Automated Digital Forensic Triage:
Rapid Detection of Anti-Forensic Tools

Thomas Laurenson

A thesis presented for the degree of
Doctor of Philosophy

Department of Information Science
School of Business
University of Otago
Dunedin, New Zealand
June, 2016
Abstract

We live in the information age. Our world is interconnected by digital devices and electronic communication. As such, criminals are finding opportunities to exploit our information rich electronic data. In 2014, the estimated annual cost from computer-related crime was more than 800 billion dollars. Examples include the theft of intellectual property, electronic fraud, identity theft and the distribution of illicit material. Digital forensics grew out of necessity to combat computer crime and involves the investigation and analysis of electronic data after a suspected criminal act.

Challenges in digital forensics exist due to constant changes in technology. Investigation challenges include exponential growth in the number of cases and the size of targets; for example, forensic practitioners must analyse multi-terabyte cases comprised of numerous digital devices. A variety of applied challenges also exist, due to continual technological advancements; for example, anti-forensic tools, including the malicious use of encryption or data wiping tools, hinder digital investigations by hiding or removing the availability of evidence.

In response, the objective of the research reported here was to automate the effective and efficient detection of anti-forensic tools. A design science research methodology was selected as it provides an applied research method to design, implement and evaluate an innovative Information Technology (IT) artifact to solve a specified problem. The research objective require that a system be designed and implemented to perform automated detection of digital artifacts (e.g., data files and Windows Registry entries) on a target data set. The goal of the system is to automatically determine if an anti-forensic tool is present, or absent, in order to prioritise additional in-depth investigation. The system performs rapid forensic triage, suitable for execution against multiple investigation targets, providing an analyst with high-level information regarding potential malicious anti-forensic tool usage.

The system is divided into two main stages: 1) Design and implementation of a solution to automate creation of an application profile (application software reference set) of known unique digital artifacts; and 2) Digital artifact matching between the created reference set and a target data set. Two tools were designed and implemented: 1) A live differential analysis tool, named LiveDiff, to reverse engineer application software with a specific emphasis on digital forensic requirements; 2) A digital artifact matching framework, named Vestigium, to correlate digital artifact metadata and detect anti-forensic tool presence. In addition, a forensic data abstraction, named Application Profile XML (APXML), was designed to store and distribute digital artifact metadata. An associated Application Programming Interface (API), named apxml.py, was authored to provide automated processing of APXML documents. Together, the tools provided an automated triage system to detect anti-forensic tool presence on an investigation target.
A two-phase approach was employed in order to assess the research products. The first phase of experimental testing involved demonstration in a controlled laboratory environment. First, the LiveDiff tool was used to create application profiles for three anti-forensic tools. The automated data collection and comparison procedure was more effective and efficient than previous approaches. Two data reduction techniques were tested to remove irrelevant operating system noise: application profile intersection and dynamic blacklisting were found to be effective in this regard. Second, the profiles were used as input to Vestigium and automated digital artifact matching was performed against authored known data sets. The results established the desired system functionality and demonstration then led to refinements of the system, as per the cyclical nature of design science.

The second phase of experimental testing involved evaluation using two additional data sets to establish effectiveness and efficiency in a real-world investigation scenario. First, a public data set was subjected to testing to provide research reproducibility, as well as to evaluate system effectiveness in a variety of complex detection scenarios. Results showed the ability to detect anti-forensic tools using a different version than that included in the application profile and on a different Windows operating system version. Both are scenarios where traditional hash set analysis fails. Furthermore, Vestigium was able to detect residual and deleted information, even after a tool had been uninstalled by the user. The efficiency of the system was determined and refinements made, resulting in an implementation that can meet forensic triage requirements. Second, a real-world data set was constructed using a collection of second-hand hard drives. The goal was to test the system using unpredictable and diverse data to provide more robust findings in an uncontrolled environment. The system detected one anti-forensic tool on the data set and processed all input data successfully without error, further validating system design and implementation.

The key outcome of this research is the design and implementation of an automated system to detect anti-forensic tool presence on a target data set. Evaluation suggested the solution was both effective and efficient, adhering to forensic triage requirements. Furthermore, techniques not previously utilised in forensic analysis were designed and applied throughout the research: dynamic blacklisting and profile intersection removed irrelevant operating system noise from application profiles; metadata matching methods resulted in efficient digital artifact detection and path normalisation aided full path correlation in complex matching scenarios. The system was subjected to rigorous experimental testing on three data sets that comprised more than 10 terabytes of data. The ultimate outcome is a practically implemented solution that has been executed on hundreds of forensic disk images, thousands of Windows Registry hives, more than 10 million data files, and approximately 50 million Registry entries. The research has resulted in the design of a scalable triage system implemented as a set of computer forensic tools.
Acknowledgements

A doctoral dissertation is solely undertaken by an individual as many others contribute and play a role in reaching the final goal. Acknowledgments must therefore be made to the people who helped me achieve the work presented in this thesis.

A heartfelt thank-you must first go to my parents who have continually supported me, not only during my PhD candidature, but in every facet of my life. Without their unwavering love and encouragement I would never have achieved my academic, career and life goals. To my older sister, Dr. Sophie Laurenson, thank-you for always looking after and inspiring me. To my partner, Victoria Alogna, we embarked on our doctoral journey together and supported each other every step of the way. We make a great team, thank-you, you are the best.

To my supervisors, I would like to express my sincere appreciation, firstly to my primary advisor, Associate Professor Hank Wolfe, for ongoing encouragement throughout my doctoral studies. Thank-you for letting me choose my research direction and supporting my research endeavour. My sincere gratitude goes to my secondary advisor, Professor Stephen MacDonell, for reassuring me during stressful times and for his immense knowledge and guidance on research methodologies, thank-you for the time and effort in supporting me throughout the research journey. Without my thesis advisors this PhD would not have been achieved.

I also wish to thank the members of the Information Science department at the University of Otago. I feel we are a small, perhaps odd, family of researchers with a shared passion. A special thanks to Professor Peter Whigham, Dr. Sherlock Licorish, Dr. Tobias Langlotz and other members of staff who all provided something unique to my graduate research experience. Furthermore, to the Information Science administration staff, whenever I had a problem, you were there to help. So, thank-you to Gail Mercer, Heather Cooper and Stephen Hall Jones.

To my office mates, thank-you for bringing out the best in my research and both putting up with me when stressed. Thank-you to Dr. Dax Roberts who shared much knowledge and stimulating discussion early on in my PhD. To Henry Gee, it was a pleasure sharing an office with you and great to enjoy the successes we achieved in collaborative work. To my more recent office mate, Mostafa Alwash, thank-you for the stimulating discussions on work and life, it was a pleasure. Also, to the all Information Science postgraduate students, thank-you for providing an interesting research community.

Acknowledgments must also be made to a number of academics and researchers in my specialist research field of digital forensics as their research and shared passion has provided me with the enthusiasm and intellectual stimulus. Simson Garfinkel, the research papers, techniques and tools you have released are second to none, thank-you for always releasing open source forensic tools and libraries. Alex Nelson, after hours of pouring over your source code, thank-you for your novel ideas and excellent code documentation. Vassil Roussev, your advanced and novel research output has inspired me to seek more elegant and comprehensive solutions. To Eric Zimmerman and Harlan Carvey, thank-you both for creating, documenting and sharing you excellent research on the cryptic Windows Registry. To all other authors
of papers in the Digital Investigation journal, this publication has entertained, provoked and sometimes baffled me while always fostering my interest in digital forensics.
## Contents

1 Introduction .................................................................................. 1
   1.1 Research Motivation................................................................. 2
   1.2 Research Methodology............................................................. 5
   1.3 System Design and Architecture............................................... 7
   1.4 Research Contribution............................................................ 8
   1.5 Research Publications............................................................ 10
   1.6 Thesis Structure..................................................................... 11

2 Background ................................................................................... 13
   2.1 Digital Forensics.................................................................... 14
      2.1.1 Digital Forensic Procedure............................................... 15
         2.1.1.1 Identification of Evidence......................................... 16
         2.1.1.2 Collection of Evidence............................................ 16
         2.1.1.3 Examination of Evidence........................................ 17
         2.1.1.4 Analysis of Evidence............................................... 17
         2.1.1.5 Reporting of Evidence............................................. 18
         2.1.1.6 Preservation of Evidence Integrity............................. 19
      2.1.2 Digital Forensic Triage...................................................... 19
   2.2 Digital Evidence..................................................................... 20
      2.2.1 Digital Artifacts................................................................ 20
      2.2.2 Digital Evidence Sources............................................... 21
         2.2.2.1 File System............................................................ 21
         2.2.2.2 Windows Registry................................................. 22
      2.2.3 Digital Forensic Data Abstractions.................................... 23
         2.2.3.1 Forensic Disk Images............................................. 24
         2.2.3.2 High-level Forensic Data Abstractions....................... 25
   2.3 Digital Forensic Research Challenges....................................... 26
      2.3.1 Data Volume & Scalability............................................... 27
      2.3.2 Case Complexity............................................................ 27
      2.3.3 Technology Advancement Challenges................................ 28
      2.3.4 The CSI Effect............................................................... 29
   2.4 Application Software.............................................................. 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>Application Software Life Cycle</td>
<td>30</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Reverse Engineering Application Software</td>
<td>31</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Forensic Analysis of Application Software</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>Anti-forensics</td>
<td>34</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Anti-forensic Techniques</td>
<td>34</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>Evidence Hiding</td>
<td>34</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Evidence Destruction</td>
<td>35</td>
</tr>
<tr>
<td>2.5.1.3</td>
<td>Evidence Elimination</td>
<td>36</td>
</tr>
<tr>
<td>2.5.1.4</td>
<td>Evidence Counterfeiting</td>
<td>37</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Anti-forensic Tools</td>
<td>37</td>
</tr>
<tr>
<td>2.5.2.1</td>
<td>Data Encryption Tools</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>Data Wiping Tools</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2.3</td>
<td>Privacy Tools</td>
<td>39</td>
</tr>
<tr>
<td>2.5.3</td>
<td>The Anti-forensics Problem</td>
<td>39</td>
</tr>
<tr>
<td>2.5.3.1</td>
<td>Anti-forensic Prevalence</td>
<td>39</td>
</tr>
<tr>
<td>2.5.3.2</td>
<td>Anti-forensic Tool Usage</td>
<td>40</td>
</tr>
<tr>
<td>2.6</td>
<td>Conclusion</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>Literature Review</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>Forensic Analysis of Anti-forensic Tools</td>
<td>44</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Evaluation of Counter-Forensic Privacy Tools</td>
<td>44</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Assessing Trace Evidence Left by Secure Deletion Programs</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3</td>
<td>An Evaluation of Data Erasing Tools</td>
<td>46</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Identifying Trace Evidence from Target-Specific Data Wiping</td>
<td>47</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Detecting Data Concealment Programs</td>
<td>47</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Anti-forensic Research Challenges</td>
<td>49</td>
</tr>
<tr>
<td>3.2</td>
<td>Reverse Engineering of Application Software</td>
<td>50</td>
</tr>
<tr>
<td>3.2.1</td>
<td>System Monitoring</td>
<td>50</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Differential Analysis</td>
<td>52</td>
</tr>
<tr>
<td>3.2.2.1</td>
<td>Snapshot Tools</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2.2</td>
<td>Disk Image and Registry Hive Tools</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Reverse Engineering Challenges</td>
<td>55</td>
</tr>
<tr>
<td>3.2.3.1</td>
<td>Lack of Standardisation</td>
<td>55</td>
</tr>
<tr>
<td>3.2.3.2</td>
<td>Experimental Testing Challenges</td>
<td>56</td>
</tr>
<tr>
<td>3.2.3.3</td>
<td>Prevalence of Unrelated Results</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>Automated Digital Forensic Analysis</td>
<td>58</td>
</tr>
<tr>
<td>3.3.1</td>
<td>File Hashing and Reference Sets</td>
<td>59</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Advanced Forensic Hashing Techniques</td>
<td>61</td>
</tr>
<tr>
<td>3.3.2.1</td>
<td>Block Hashing</td>
<td>61</td>
</tr>
<tr>
<td>3.3.2.2</td>
<td>Approximate Matching and Similarity Digests</td>
<td>63</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Performance Summary of Forensic Hashing Tools</td>
<td>65</td>
</tr>
</tbody>
</table>
3.3.4 NIST Diskprint Project ........................................ 66
3.3.5 Automated Detection of Registry Entries ....................... 68
  3.3.5.1 Windows Registry Analysis Tools .......................... 68
  3.3.5.2 Automated Windows Registry Analysis ...................... 69
3.3.6 Automated Forensic Analysis Challenges ....................... 70
  3.3.6.1 Automated Detection Performance ........................ 70
  3.3.6.2 Digital Artifact Diversity ............................... 71
3.4 Conclusion .................................................. 71

4 Research Methodology ............................................. 73
  1.1 Digital Forensic Research Methodologies ......................... 74
  1.2 Design Science Research Methodology ............................ 76
    4.2.1 Identify Research Problem and Motivation ................. 77
      4.2.1.1 Problem 1: Creating Application Software Reference Sets ... 78
      4.2.1.2 Problem 2: Storing Application Software Reference Sets ...... 79
      4.2.1.3 Problem 3: Correlating Reference Sets Against Targets ...... 80
    4.2.2 Define the Objectives of a Solution ....................... 81
      4.2.2.1 High-Level Research Objective .......................... 81
      4.2.2.2 Reverse Engineering Objectives ........................ 83
      4.2.2.3 Reference Set Objectives ................................ 83
      4.2.2.4 Target Data Set Objectives ............................ 84
      4.2.2.5 Digital Artifact Matching Objectives ................... 84
    4.2.3 Design and Development .................................. 86
    4.2.4 Demonstration .......................................... 86
    4.2.5 Evaluation ............................................ 87
      4.2.5.1 System Effectiveness .................................. 87
      4.2.5.2 System Efficiency ................................... 88
    4.2.6 Communication .......................................... 88
  4.3 Experimental Testing Method .................................. 89
    4.3.1 Operating System Selection ................................ 89
    4.3.2 Application Software Selection ............................ 89
      4.3.2.1 CCleaner .............................................. 90
      4.3.2.2 Eraser .............................................. 90
      4.3.2.3 TrueCrypt ........................................... 90
    4.3.3 Data Set Selection ....................................... 91
      4.3.3.1 Known Content Data Sets ................................ 91
      4.3.3.2 Publicly Available Data Sets ............................ 92
      4.3.3.3 Real World Data Sets ................................ 93
      4.3.3.4 Data Set Summary .................................... 94
    4.3.4 Effectiveness Metrics ..................................... 94
      4.3.4.1 Information Retrieval Metrics ........................ 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.3.1 System Design Requirements</td>
<td>127</td>
</tr>
<tr>
<td>5.4.3.2 Windows Registry Metadata Generation Solution</td>
<td>128</td>
</tr>
<tr>
<td>5.4.3.3 Software Development: HiveExtractor.py</td>
<td>128</td>
</tr>
<tr>
<td>5.4.3.4 Software Development: CellXML-Registry</td>
<td>129</td>
</tr>
<tr>
<td>5.4.4 Target Data Set Processing Summary</td>
<td>129</td>
</tr>
<tr>
<td>5.5 Digital Artifact Matching</td>
<td>130</td>
</tr>
<tr>
<td>5.5.1 Digital Artifact Matching Requirements</td>
<td>130</td>
</tr>
<tr>
<td>5.5.2 Advanced Tool Design: Vestigium</td>
<td>131</td>
</tr>
<tr>
<td>5.5.3 Digital Artifact Matching Strategies</td>
<td>132</td>
</tr>
<tr>
<td>5.5.3.1 Path Normalisation</td>
<td>132</td>
</tr>
<tr>
<td>5.5.3.2 Available Digital Artifact Metadata Properties</td>
<td>136</td>
</tr>
<tr>
<td>5.5.3.3 File System Matching Methods</td>
<td>137</td>
</tr>
<tr>
<td>5.5.3.4 Windows Registry Matching Methods</td>
<td>138</td>
</tr>
<tr>
<td>5.5.4 File System Matching</td>
<td>139</td>
</tr>
<tr>
<td>5.5.4.1 Application Profile Processing</td>
<td>140</td>
</tr>
<tr>
<td>5.5.4.2 File System Artifact Matching Process</td>
<td>141</td>
</tr>
<tr>
<td>5.5.5 Windows Registry Matching</td>
<td>142</td>
</tr>
<tr>
<td>5.5.5.1 Application Profile Processing</td>
<td>143</td>
</tr>
<tr>
<td>5.5.5.2 Registry Artifact Matching Process</td>
<td>143</td>
</tr>
<tr>
<td>5.5.6 Forensic Reporting</td>
<td>145</td>
</tr>
<tr>
<td>5.5.6.1 Log File</td>
<td>145</td>
</tr>
<tr>
<td>5.5.6.2 DFXML and RegXML Reports</td>
<td>146</td>
</tr>
<tr>
<td>5.5.7 Software Development: Vestigium</td>
<td>146</td>
</tr>
<tr>
<td>5.5.8 Digital Artifact Matching Summary</td>
<td>147</td>
</tr>
<tr>
<td>5.6 Conclusion</td>
<td>148</td>
</tr>
<tr>
<td>6 System Demonstration</td>
<td>149</td>
</tr>
<tr>
<td>6.1 Tool Operation</td>
<td>150</td>
</tr>
<tr>
<td>6.1.1 Creating Application Profiles: LiveDiff</td>
<td>151</td>
</tr>
<tr>
<td>6.1.2 Processing Application Profiles: apxml.py</td>
<td>154</td>
</tr>
<tr>
<td>6.1.3 Target Data Set Processing: CellXML-Registry</td>
<td>155</td>
</tr>
<tr>
<td>6.1.4 Digital Artifact Matching: Vestigium</td>
<td>156</td>
</tr>
<tr>
<td>6.1.5 System Implementation Summary</td>
<td>159</td>
</tr>
<tr>
<td>6.2 Experimental Testing</td>
<td>159</td>
</tr>
<tr>
<td>6.2.1 Virtual Machine Testing Environment</td>
<td>160</td>
</tr>
<tr>
<td>6.2.2 Data Generation: Recreating the Application Life Cycle</td>
<td>160</td>
</tr>
<tr>
<td>6.2.3 Data Collection: Creating Application Profiles</td>
<td>162</td>
</tr>
<tr>
<td>6.2.4 Data Collection: Creating Known Data Sets</td>
<td>163</td>
</tr>
<tr>
<td>6.2.5 Establishing Data Set Ground Truth</td>
<td>164</td>
</tr>
<tr>
<td>6.2.5.1 Ground Truth Requirements</td>
<td>165</td>
</tr>
<tr>
<td>6.2.5.2 Ground Truth: Determining Windows Version</td>
<td>165</td>
</tr>
</tbody>
</table>
6.2.5.3 Ground Truth: Determining Anti-forensic Presence ........................................ 166
6.2.5.4 Ground Truth: Determining Digital Artifacts .................................................. 166
6.2.6 System Evaluation Metrics: Score Interpretation .................................................. 167
6.2.6.1 Effectiveness Metrics ......................................................................................... 168
6.2.6.2 Efficiency Metrics .............................................................................................. 168
6.2.7 Testing Environment Summary ............................................................................ 169
6.3 System Demonstration: Creating Application Profiles ............................................ 169
6.3.1 Anti-forensic Tool Selection .................................................................................. 170
6.3.2 Overview of Created Application Profiles ............................................................... 170
6.4 Filtering Irrelevant Artifacts from Application Profiles ............................................ 171
6.4.1 Application Profile Intersection ........................................................................... 172
6.4.1.1 Application Profile Intersection Method .......................................................... 172
6.4.1.2 Application Profile Intersection Results ............................................................ 173
6.4.1.3 Analysis of Application Profile Intersection Output ........................................... 175
6.4.2 Application Profile Blacklisting ............................................................................ 177
6.4.2.1 Application Profile Blacklisting Method ........................................................... 177
6.4.2.2 Application Profile Blacklisting Results ............................................................. 178
6.4.2.3 Analysis of Application Profile Blacklisting ....................................................... 179
6.4.3 Discussion of Data Reduction Findings ................................................................. 182
6.5 System Demonstration: Recreated Application Profiles .......................................... 183
6.5.1 Revised Application Profile Generation Method ..................................................... 183
6.5.2 Overview of Recreated Application Profiles .......................................................... 184
6.5.3 Further Data Reduction: Static Blacklist Implementation ...................................... 185
6.5.4 Application Profile Creation Summary ................................................................. 186
6.6 System Demonstration: Digital Artifact Matching .................................................... 187
6.6.1 Overview of Known Data Set ................................................................................ 187
6.6.2 Experimental Testing Method .............................................................................. 189
6.6.3 Establishing Known Data Set Ground Truth ........................................................... 189
6.6.4 Known Data Set Testing ....................................................................................... 191
6.6.4.1 Matching Effectiveness: File System Entries ..................................................... 191
6.6.4.2 Matching Effectiveness: Registry Entries ......................................................... 193
6.6.5 Discussion of Findings .......................................................................................... 196
6.6.6 Design Modifications to Digital Artifact Matching ................................................. 196
6.7 Overview of Achieved Research Objectives .............................................................. 197
6.8 Conclusion ............................................................................................................... 198
7 System Evaluation ....................................................................................................... 201
7.1 Public Data Set Overview: M57-Patents Scenario ...................................................... 202
7.1.1 Overview of M57-Patents Scenario ...................................................................... 202
7.1.2 Experimental Testing Method .............................................................................. 204
7.1.3 Establishing Ground Truth for the M57-Patents Scenario .................................... 204
Chapter 7: M57-Patents Scenario

7.1 Determining Anti-forensic Tool Presence

7.1.3 Determining and Documenting Anti-forensic Tool Artifacts

7.2 M57-Patents Scenario: Effectiveness Testing and Results

7.2.1 Detecting Anti-forensic Tools using Vestigium

7.2.1.1 File System Matching Effectiveness

7.2.1.2 File System Matching Effectiveness Comparison: Hash Sets

7.2.1.3 Registry Matching Effectiveness

7.2.1.4 Registry Matching Effectiveness Comparison: Searching

7.2.2 False Positive Detection Rate using Vestigium

7.2.3 Detecting Residual Digital Artifacts using Vestigium

7.2.3.1 Application Profile Creation: XP Advanced Keylogger

7.2.3.2 Testing Methodology

7.2.3.3 Residual Digital Artifact Detection Results

7.2.4 Effectiveness of Path Normalisation Technique

7.2.4.1 Path Normalisation Results for File System Entries

7.2.4.2 Path Normalisation Results for Registry Entries

7.2.4.3 Discussion of Path Normalisation Results

7.2.5 Effectiveness of the Windows Registry Processing Technique

7.3 M57-Patents Scenario: Efficiency Testing and Findings

7.3.1 Efficiency Metrics and Testing Method

7.3.2 System Efficiency Results

7.3.3 Improving Vestigium Efficiency

7.3.3.1 Improving File System Metadata Generation Efficiency

7.3.3.2 Improving Registry Metadata Generation Efficiency

7.3.3.3 Improving Digital Artifact Matching Efficiency

7.3.4 Revised System Efficiency Results

7.4 M57-Patents Scenario: Discussion and Observations

7.4.1 System Effectiveness

7.4.2 System Efficiency

7.4.3 System Design Modifications

7.5 Real-World Data Set Overview: Second Hand Hard Drives

7.5.1 Real-World Data Set Sources

7.5.2 Selection Criteria for Real-World Data Sets

7.5.3 Overview of Selected Hard Drives from Real-World Data Set

7.5.4 Real-World Data Set Experimental Testing Method

7.6 Real-World Data Set Findings

7.6.1 Detecting Anti-forensic Tools using Vestigium

7.6.2 System Efficiency Results

7.6.3 System Efficiency Results: Disk Access Method and Analysis System

7.6.4 Discussion and Observations
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Digital forensic volume statistics from FBI RCFL</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Design Science Research Methodology (DSRM) process model</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>High-level overview of the system design architecture</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>DSRM process model with links to relevant chapters</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Digital forensic procedure</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Structure of the Windows Registry</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Application software life cycle</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Abstraction levels and layers of an HTML file</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Fingerprint generation system to detect data concealment programs</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Visual representation of block hashing</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>Throughput for 500 GB hard drive</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>Diskprint analysis workflow</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td>Design Science Research Methodology (DSRM) process model</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>High-level overview of the proposed system design architecture</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>High-level overview of the system architecture</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>Visual representation of the snapshot differencing method</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Data collection procedure implemented in LiveDiff</td>
<td>106</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of a prefix tree populated with file system entries</td>
<td>108</td>
</tr>
<tr>
<td>5.5</td>
<td>High-level overview of target data set processing</td>
<td>125</td>
</tr>
<tr>
<td>5.6</td>
<td>High-level overview of the file system matching process</td>
<td>140</td>
</tr>
<tr>
<td>5.7</td>
<td>High-level overview of the Registry matching process</td>
<td>144</td>
</tr>
<tr>
<td>6.1</td>
<td>High-level overview of the implemented system design</td>
<td>151</td>
</tr>
<tr>
<td>6.2</td>
<td>LiveDiff program help menu</td>
<td>152</td>
</tr>
<tr>
<td>6.3</td>
<td>Vestigium program help menu</td>
<td>157</td>
</tr>
<tr>
<td>6.4</td>
<td>High-level graphical overview of the application life cycle phases</td>
<td>161</td>
</tr>
<tr>
<td>6.5</td>
<td>High-level overview of the known data set generation method</td>
<td>164</td>
</tr>
<tr>
<td>6.6</td>
<td>High-level overview of the method to establish ground truth</td>
<td>167</td>
</tr>
<tr>
<td>6.7</td>
<td>Summary of application profile intersection results</td>
<td>174</td>
</tr>
<tr>
<td>6.8</td>
<td>Summary of application profile intersection results for TrueCrypt</td>
<td>175</td>
</tr>
</tbody>
</table>
6.9 Averaged application profile counts before and after dynamic blacklisting . . . 179

7.1 Longitudinal detection capabilities for XP Keylogger tool . . . . . . . . . . . . 227
7.2 Overview of actual paths versus normalised paths matching results . . . . . 234

A.1 Ethical approval granted for research involving second-hand hard drives . . . 288
A.2 Ethical approval granted to the author for this research . . . . . . . . . . . . . . 289
List of Tables

2.1 Types of electronic devices .................................................. 16
2.2 Classified levels of digital forensic analysis ............................. 18
2.3 Types of digital artifacts ....................................................... 21
2.4 Common digital forensic data abstractions .............................. 23
2.5 Common disk image file formats ............................................. 24
2.6 Classification of data wiping techniques ................................. 36
2.7 Anti-forensic tool classifications ............................................ 37
3.1 File signatures from secure deletion programs ......................... 45
3.2 Forensic analysis results of data erasing tools ......................... 46
3.3 Seraph matching results ....................................................... 48
3.4 An overview of system monitoring tools ................................... 51
3.5 An overview of differential analysis tools ............................... 53
3.6 Run-time generation efficiency of common digital forensic hash functions ............................................... 65
3.7 Generation and comparison efficiency for digital forensic hash functions ............................................... 66
3.8 List of common Windows Registry analysis tools ....................... 69
4.1 Coincidence matrix for binary classification ............................... 96
5.1 Overview of data structures for different application software artifacts ............................................... 104
5.2 List of LiveDiff source code files with a description of functionality ............................................... 111
5.3 Overview of application profile metadata properties ................. 117
5.4 Properties added to the application profile .............................. 117
5.5 Summary of application profile metadata properties ................. 119
5.6 Pre-defined XML entities and the associated LiveDiff checking function ............................................... 123
5.7 Registry value data type variability between LiveDiff and hivexml ............................................... 124
5.8 Path normalisation examples using environment variable naming conventions ............................................... 133
5.9 Overview of file system path normalisation values ................. 135
5.10 Overview of Windows Registry path normalisation ................ 135
5.11 Overview of Windows Registry base name normalisation .......... 135
5.12 Complete list of available APXML metadata properties ........... 136
5.13 List of Vestigium Python modules ........................................ 147
5.14 List of Vestigium software package dependencies ................ 147
6.1 List of command line arguments for the LiveDiff tool . . . . . . . . . . . . 153
6.2 List of command line arguments for the CellXML-Registry tool . . . . . . 156
6.3 List of command line arguments for the Vestigium tool . . . . . . . . . . . . 158
6.4 Application life cycle phases selected for experimental testing . . . . . . . . 160
6.5 Overview of the anti-forensic tools selected for application profile testing . . 170
6.6 Average digital artifact counts for created application profiles . . . . . . . . 170
6.7 Summary of application profile intersection results . . . . . . . . . . . . . . 173
6.8 Summary of blacklisting effectiveness for each anti-forensic tool . . . . . . . 178
6.9 Blacklisted data files classified on logical file system location . . . . . . . . 180
6.10 Blacklisted Registry values classified on logical Registry hive path . . . . . 181
6.11 Overview and comparison of original versus revised application profiles . . . 184
6.12 Overview and comparison of original versus revised application profiles . . . 186
6.13 Overview of the created forensic disk images from the known data set . . . . 188
6.14 Overview of the M57-Patents scenario content . . . . . . . . . . . . . . . . . 188
6.15 Overview of ground truth information for the known data set . . . . . . . . 190
6.16 Effectiveness results for file system matching for the known data set . . . . 192
6.17 Effectiveness results for Registry matching for the known data set . . . . . . 194

7.1 High-level overview of the M57-Patents scenario based on user . . . . . . . . 202
7.2 Overview of the M57-Patents scenario content . . . . . . . . . . . . . . . . . 203
7.3 Potentially interesting forensic disk images from the M57-Patents scenario . . 206
7.4 Tool versions for application profiles versus the M57-Patents scenario . . . . 206
7.5 Overview of ground truth information for the M57-Patents scenario . . . . . 207
7.6 High-level overview of Vestigium results for the M57-Patents scenario . . . . 209
7.7 Effectiveness results for file system matching for the M57-Patents Scenario . . 210
7.8 Overview of file system matching results for Eraser version 5.7.8 . . . . . . . . 212
7.9 Effectiveness results for Registry matching for the M57-Patents Scenario . . . 215
7.10 Overview of Windows Registry matching results for Eraser version 5.7.8 . . 216
7.11 Overview of Registry keyword search analysis on the M57-Patents scenario . . 219
7.12 Cumulated false positive results for file system matching . . . . . . . . . . . 220
7.13 Cumulated false positive results for Registry matching . . . . . . . . . . . . 221
7.14 Overview of the application profile created for XP Keylogger . . . . . . . . 223
7.15 Overview of file system and Registry matching for XP Keylogger . . . . . . 224
7.16 Overview of file system path detection rates using path normalisation . . . . 230
7.17 Overview of Registry path normalisation results . . . . . . . . . . . . . . . . 232
7.18 Comparison of Registry parsing capabilities . . . . . . . . . . . . . . . . . . 235
7.19 Overview of system efficiency for the M57-Patents scenario . . . . . . . . . 237
7.20 Overview of efficiency results of the four Vestigium processing steps . . . . 238
7.21 Overview of fiwalk performance for the M57-Patents scenario . . . . . . . . 240
7.22 Overview of Registry processing tools for the M57-Patents scenario . . . . . 243
7.23 Overview of revised system efficiency results for the M57-Patents scenario . . 245
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.24 Overview of the hard drives sourced for real-world data set testing</td>
<td>250</td>
</tr>
<tr>
<td>7.25 Overview of the hard drives selected for real-world data set testing</td>
<td>250</td>
</tr>
<tr>
<td>7.26 Overview of the content of hard drives selected for real-world data set testing</td>
<td>251</td>
</tr>
<tr>
<td>7.27 Effectiveness results for file system matching for the real-world data set</td>
<td>253</td>
</tr>
<tr>
<td>7.28 Effectiveness results for Registry matching for the real-world data set</td>
<td>254</td>
</tr>
<tr>
<td>7.29 Overview of system efficiency for real-world data set processing</td>
<td>255</td>
</tr>
<tr>
<td>7.30 System efficiency for real-world data set processing on field triage laptop</td>
<td>257</td>
</tr>
<tr>
<td>B.1 Overview of fileshot.c functions</td>
<td>294</td>
</tr>
<tr>
<td>B.2 Overview of regshot.c functions</td>
<td>295</td>
</tr>
<tr>
<td>C.1 Summary of application profile contents for CCleaner version 5.09</td>
<td>323</td>
</tr>
<tr>
<td>C.2 Summary of application profile contents for Eraser version 6.2.0.2970</td>
<td>324</td>
</tr>
<tr>
<td>C.3 Summary of application profile contents for TrueCrypt version 7.1a</td>
<td>325</td>
</tr>
<tr>
<td>C.4 Application Profile Intersection Results for CCleaner version 5.09</td>
<td>333</td>
</tr>
<tr>
<td>C.5 Application Profile Intersection Results for Eraser version 6.2.0.2970</td>
<td>334</td>
</tr>
<tr>
<td>C.6 Application Profile Intersection Results for TrueCrypt version 7.1a</td>
<td>335</td>
</tr>
<tr>
<td>C.7 Summary of the final application profile for CCleaner version 5.09</td>
<td>344</td>
</tr>
<tr>
<td>C.8 Summary of the final application profile for Eraser version 6.2.0</td>
<td>344</td>
</tr>
<tr>
<td>C.9 Summary of the final application profile for TrueCrypt version 7.1a</td>
<td>345</td>
</tr>
<tr>
<td>C.10 Summary of the final application profile for XP Keylogger version 2.1</td>
<td>345</td>
</tr>
<tr>
<td>C.11 Overview of file system counts for the known data set</td>
<td>350</td>
</tr>
<tr>
<td>C.12 Overview of Registry counts for the known data set</td>
<td>350</td>
</tr>
<tr>
<td>D.1 Overview of file system counts for the M57-Patents scenario</td>
<td>353</td>
</tr>
<tr>
<td>D.2 Overview of Registry counts for the M57-Patents scenario</td>
<td>356</td>
</tr>
<tr>
<td>D.3 Overview of the drives sourced for the real world data set</td>
<td>377</td>
</tr>
<tr>
<td>D.4 Overview of file system counts for the real-world data set</td>
<td>379</td>
</tr>
<tr>
<td>D.5 Overview of Registry counts for the real-world data set</td>
<td>381</td>
</tr>
</tbody>
</table>
Listings

3.1 RDS example entry ........................................... 60
3.2 Example of an ssdeep hash value ........................ 64
5.1 DFXML fileobject example ................................. 113
5.2 Example of RegXML structure ............................. 114
5.3 Example of the revised RegXML CellObject structure ... 115
5.4 Example of the Application Profile XML (APXML) structure ... 115
5.5 Example of reading multiple APXML documents ......... 141
6.1 Steps to invoke Command Prompt with administrator rights .......... 152
6.2 Functional example of the APXML API ..................... 155
6.3 Example of the APXML API statistics function .......... 155
6.4 Command line example to invoke Vestigium .............. 158
6.5 Command line example to invoke Vestigium with additional arguments ... 159
6.6 Example of method to create an immutable virtual disk ........ 162
6.7 Example of method used to clone virtual disks .......... 163
6.8 Example of LiveDiff static blacklist structure and entries ...... 185
6.9 Command line example to invoke Vestigium against the known data set .... 189
6.10 Example FileObjects with missing and incorrect metadata properties .......... 193
6.11 Example CellObjects with incomplete metadata properties .... 195
7.1 Command example to invoke Vestigium against the M57-Patents scenario .... 204
7.2 File path comparison between target and application profiles ........ 211
7.3 Example of the script used to locate MD5 hash matches .......... 213
7.4 List of additional file system paths detected using path normalisation .... 231
7.5 Command line example to invoke Vestigium against the real-world data set .... 252
B.1 Example of the Application Profile XML creator element .......... 296
B.2 Example of the Application Profile XML metadata element ........ 296
B.3 Working example of TrueCrypt APXML document ............ 297
B.4 Full source code listing for APXMLPrintStats.py ........... 298
B.5 Full source code listing for APXML2CSV.py ................ 299
B.6 Full source code listing for HiveExtractor.py .............. 302
B.7 Special character checking functions added to RegistryHive.cs ....... 307
B.8 RegXML output function added to RegistryHive.cs .......... 308
B.9 RegXML generation function added to RegistryHive.cs .......... 311
**List of Acronyms**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFF</td>
<td>Advanced Forensic Format</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APXML</td>
<td>Application Profile XML</td>
</tr>
<tr>
<td>CD</td>
<td>Compact Disc</td>
</tr>
<tr>
<td>CFTT</td>
<td>Computer Forensic Tool Testing</td>
</tr>
<tr>
<td>CF</td>
<td>Computer Forensics</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CSI</td>
<td>Crime Scene Investigation</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td>CTPH</td>
<td>Context Triggered Piecewise Hashing</td>
</tr>
<tr>
<td>DEX</td>
<td>Digital Evidence Exchange</td>
</tr>
<tr>
<td>DFXML</td>
<td>Digital Forensics XML</td>
</tr>
<tr>
<td>DF</td>
<td>Digital Forensics</td>
</tr>
<tr>
<td>DSRM</td>
<td>Design Science Research Methodology</td>
</tr>
<tr>
<td>DS</td>
<td>Design Science</td>
</tr>
<tr>
<td>EWF</td>
<td>Expert Witness Compression</td>
</tr>
<tr>
<td>EXT</td>
<td>Extended File System</td>
</tr>
<tr>
<td>FAT</td>
<td>File Allocation Table</td>
</tr>
<tr>
<td>FBI</td>
<td>Federal Bureau of Investigation</td>
</tr>
<tr>
<td>FTK</td>
<td>Forensic ToolKit</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
<tr>
<td>HFS</td>
<td>Hierarchical File System</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provided</td>
</tr>
<tr>
<td>IS</td>
<td>Information Systems</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>MAC</td>
<td>Modified, Accessed and Created</td>
</tr>
</tbody>
</table>
MB/s  Megabytes per second
MB    Megabyte
MD5   Message Digest algorithm version 5
MIT   Massachusetts Institute of Technology
MS-DOS Microsoft Disk Operating System
MSDN  Microsoft Developer Network
NFS   Network File System
NFTS  New Technology File System
NIJ   National Institute of Justice
NIST  National Institute of Standards and Technology
NSRL  National Software Reference Library
NTFS  New Technology File System
OS    Operating System
PCO   Profile CellObject
PC    Personal Computer
PFO   Profile FileObject
PGP   Pretty Good Privacy
PML   Process Monitor Format
RAM   Random Access Memory
RDS   Reference Data Set
RFCL  Regional Computer Forensic Laboratory
RegXML Registry XML
SAM   Security Account Manager
SATA  Serial AT Attachment
SHA1  Secure Hash Algorithm version 1
SHA   Secure Hash Algorithm
SP    Service Pack
SSN   Social Security Number
TB    Terabyte
TCO   Target CellObject
TFO   Target FileObject
UFS   Unix File System
UID   User Identifier
USB   Universal Serial Bus
VMDK  Virtual Machine Disk Format
VM    Virtual Machine

xxii
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMI</td>
<td>Windows Management Infrastructure</td>
</tr>
<tr>
<td>WSH</td>
<td>Windows Script Host</td>
</tr>
<tr>
<td>XIRAF</td>
<td>XML Information Retrieval Approach to Digital Forensics</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Information by its very nature needs to flow. In some regards, withholding information is trying to repeal the laws of gravity. You may succeed for a short period of time, but sooner or later it’s going to break free...

Gen. Michael Hayden (former director of the NSA and CIA)

1

Introduction

The word forensic derives from the Latin word forênsis, and translates as “of or before the forum” (Oxford Dictionary 2013); meaning something that is made public. From this origin the Oxford Dictionary (2013) then defines forensic as “scientific tests or techniques used in connection with the detection of crime” and leading to forensic evidence. In a traditional forensic investigation (e.g., a physical crime scene) a well known and relevant theory is that based on the exchange principle first proposed by Dr. Edmond Locard (1877–1966). Locard was a French pioneer in forensic science who propounded the concept that: “Any action of an individual, and obviously, the violent action constituting a crime, cannot occur without leaving a trace” (Locard 1934, p. 8). Locard’s exchange principle has become known as: every contact leaves a trace. In later research, Kirk (1953) expanded and expressed Locard’s exchange principle as:

Wherever he steps, whatever he touches, whatever he leaves, even unconsciously, will serve as a silent witness against him. Not only his fingerprints or his footprints, but his hair, the fibers from his clothes, the glass he breaks, the tool mark he leaves, the paint he scratches, the blood or semen he deposits or collects. All of these and more, bear mute witness against him. This is evidence that does not forget (Kirk 1953, p. 4).

Although Locard’s exchange principle was originally formulated based on traditional forensic science involving physical evidence, the same fundamental principle also exists in the digital world. The investigation of digital information is known as digital forensics, a branch
of forensic science which involves the examination of any digital component (Reith, Carr, & Gunsch, 2002). A digital investigation can encompass a wide variety of digital devices and evidence sources and, therefore, comprises sub-disciplines including computer forensics, network forensics, and smart phone forensics.

The proliferation of digital devices, such as Personal Computers (PC) and smart phones, generated and gave rise to the modern digital world that we live in. However, accompanying the widespread use of computer systems, crime in the digital world has also risen. This is commonly known as computer crime or cybercrime and has led to the need for forensic techniques and tools to be developed for the detection and investigation of digital evidence. Typically, cybercrime leaves digital fingerprints that investigators can analyse to determine how the crime was committed enabling a case to be built to potentially bring criminals to justice (Dahbur & Mohammad, 2011). The digital forensics procedure aims to discover digital fingerprints, or events, and generally involves the identification, collection, analysis and presentation of digital evidence (Palmer, 2001; Kent et al., 2006; Slay et al., 2009).

1.1 Research Motivation

“Since the 1980s, computers have had increasing roles in all aspects of human life—including an involvement in criminal acts” (Garfinkel, 2013a, p. 370). Digital investigations are now commonplace in computer crime cases, as well as in other criminal or civil cases where digital devices are present in almost every instance. As a result digital forensic techniques and tools have become an essential requirement. Specifically, digital forensic analysis tools are used daily within local, state and federal law enforcement, military sectors, government organisations and the private electronic discovery (e-Discovery) industry (Garfinkel, 2010).

The motivation for this research project in the field of digital forensics can broadly be divided into two main categories: 1) Investigation challenges including the increased number and size of cases; and 2) Technical challenges including the increased complexity of investigation targets.

In terms of investigation challenges, statistics highlight the increase in the number and size of cases with a digital investigation component. The US-based Federal Bureau of Investigation (FBI) operates the Regional Computer Forensic Laboratory (RCFL) which manages a total of 16 computer forensic laboratories that assist law enforcement agencies in collection and examination of digital evidence to support criminal investigations such as terrorism, intellectual property theft, fraud and other computer related crime (RCFL, 2014). According to statistics released by the FBI and RFCL program, the number of examinations performed increased from 987 in 2003 to 8,566 in 2012 (RFCL, 2003; RCFL, 2012). This is more than a 758% increase in the number of examinations conducted by the RCFL and equated to the need to examine around 23 to 24 individual cases every day in the year 2012. It is not only the exponential increase in cases requiring examination, the size of investigation targets is also dramatically increasing. However, research and development in computer forensic techniques and tools have had limited success in addressing efficient methods to analyse data.
sets of increasing size. In 2003 the total size of data examined was 82.3 terabytes (TB), while in 2012 the total size of data examined was 5,986 TB. This is an increase of over 7,100%. Figure 1.1 displays a summary of the number of cases and total size of data examined by the RFCL between 2003 and 2012.

![Figure 1.1: Digital forensic volume statistics from FBI Regional Computer Forensic Laboratory (RCFL) showing the total number of forensic examinations and data set size in Terabytes (TB) from 2003 to 2012 (Source: Statistics taken from Quick and Choo (2014, p. 278))](image)

As shown in this ten year period the RCFL has seen an increasing growth in the number of examinations performed as well as the amount of data processed. However, these statistics only include criminal investigations involving law enforcement. According to Roussev (2009), criminal investigation is just the tip of digital forensic analysis, compared to the majority of examinations which are conducted to support civil cases or internal investigations that can encompass multi-terabyte data sets. Quick and Choo (2014) state that the increased size of data sets for forensic analysis is due to three primary factors: 1) An increase in the number of devices seized per case; 2) An increase in the number of cases with a digital evidence component; and 3) An increase in the storage size of digital devices. Overall, the increasing volume of digital evidence is creating serious challenges for digital forensic practitioners, who are being expected to rapidly process and analyse a large amount of digital evidence. A key technique to combat data volume is by automated forensic analysis and this challenge has been taken up for this research project. Ayers (2009) states that automating forensic analysis will enhance repeatability and robustness while decreasing cost and potential errors that may result from manual analysis. Garfinkel (2012a) elaborates that automation will
also aid tool validation, and that the lack of automation has limited the progress of digital forensic research.

In addition to the challenges faced by the huge growth in the volume of digital investigations, there are also a plethora of technical challenges which all have the potential to frustrate digital forensic analysis. Technical challenges most often exist due to the complicated range of technological changes in the field of computing. According to Garfinkel (2010), the prevalence of flash memory, the variety of hardware interfaces, the proliferation of operating systems and the diversity of file formats all result in increased case complexity. Furthermore, new data storage and encryption technologies create challenges in acquiring digital evidence. Fahdi, Clarke, and Furnell (2013) surveyed a selection of digital forensic researchers and practitioners and discovered that the three principal future research challenges were cloud computing, encryption and anti-forensics. In terms of encryption, accessing digital evidence without knowledge of an encryption key or passphrase can create problems on modern operating systems where it is now commonplace to have strong full disk encryption available (Casey & Stellatos 2008). If used maliciously, encryption can be classified as an anti-forensic technique.

Anti-forensics is defined as “any attempt to compromise the availability or usefulness of evidence to the forensics process” (Harris 2006, p. S45), and is a fundamental problem in digital forensics. Anti-forensics has the potential to remove essential digital evidence that may aid in determining a digital event. Anti-forensics may involve a variety of techniques and tools ranging from encrypting data to block an investigator’s access (evidence hiding) to securely wiping digital evidence to remove evidence availability (evidence destruction). A key example of evidence destruction is overwriting the contents of illicit digital images thereby removing the potential for a forensic analyst to recover the original data.

Opinion regarding anti-forensic prevalence in digital investigations varies, but the largest American telecommunications company Verizon (2012), states that they often identify anti-forensic techniques in the field, with an estimation of $\frac{1}{3}$ of all cases incorporating some form of anti-forensic element. To further complicate the problem, “the number of scholarly papers on protecting against anti-forensic methods is greatly outnumbered by the number of websites about how to exploit the forensic process” (Harris 2006, p. S48). It was this fact that alerted the need for further academic research to address the problem of anti-forensic practices. The other realisation was that the number of tools available to perform anti-forensic techniques greatly outnumbers the forensic analysis tools able to detect anti-forensic tool usage. Some operating systems even have tools included to perform anti-forensic techniques; for example, modern versions of Microsoft Windows, Apple OS X and Linux-based operating systems all have built-in functionality to securely wipe data without the need for a third-party tool. Ultimately, criminals can maliciously use anti-forensic tools to thwart digital forensic investigations and remove the availability of essential digital evidence.

In broad terms the key phase of digital forensics targeted in this research is forensic analysis of digital evidence, determining an appropriate conclusion based on the available evidence (Kent et al. 2006). Thereafter, the motivation for the research project was inspired
and driven by a combination of the investigative challenges (increase in the number and size of data sets) and the technical challenges (increase in the complexity of digital investigations). The discussion of research and real-world challenges faced by forensic analysts has, therefore, led to the identification of a specific research topic, the primary scope of which can be summarised as **performing detection of anti-forensic tools**.

### 1.2 Research Methodology

Digital forensics falls within the Information Systems (IS) research discipline, where the goal is to produce knowledge that encompasses the application of Information Technology (IT) for organisational purposes. A common research methodology in the realm of IS is Design Science (DS). “Design Science seeks to create innovations, or artifacts, that embody the ideas, practices, technical capabilities, and products required to efficiently accomplish the analysis, design, implementation, and use of information systems” (Hevner & March 2003, p.111). In IS research, design science aims to create an innovative and purposeful IT artifact which addresses an important problem domain. There have been a variety of proposed DS research methodology models for IS research. This research project specified the use of the Design Science Research Methodology (DSRM) process model by Peffers, Tuunanen, Rothenberger, and Chatterjee (2007). Figure 1.2 displays a high-level overview of the DSRM process model.

![Design Science Research Methodology (DSRM) process model](image)

The first element of the DSRM process model is to **identify the research problem and motivation** which is covered in Section 1.1. The second element of the DSRM process model
is to **define the objectives of a solution.** One high-level research objective has been proposed for this research project:

**High-level Research Objective:** To enable effective, automated detection of relevant digital artifacts to identify application software presence on a target data set. The solution must adhere to digital forensic triage requirements including high system efficiency and initial forensic examination output.

The high-level research objective specifies an effective and efficient **system design** centred around automated forensic analysis of application software. In this research the application software of interest is anti-forensic tools, as they present obstacles for digital investigations as well as being a relevant and interesting case study. The system architecture can be classified into two main stages: 1) Automated creation of an application software reference set; and 2) Automated correlation between the reference set and a target data set. The high-level research objective has therefore been further broken down to a group of more specific lower-level objectives, thus dividing the proposed system design into a variety of sub-components, each with their own objectives.

The third element of the DSRM process model is the existent **design** of the proposed artifact (system). Through design the process seeks attempts to solve the research problem based on the specified research objectives. To aid this process, design requirements are established followed by the application of relevant scientific and technical knowledge to design a solution. In this research the process produces a system design to perform automated forensic analysis of application software, implemented in the form of computer software to provide a real-world applicable solution to the problem domain.

The fourth element of the DSRM process model is the **demonstration** of the implemented system design which usually involves experimentation, simulation or a case study to aid in showing that the implemented system can solve the specified problem. In this research a controlled laboratory testing environment is used to test the functionality of the implemented system to both automate the creation of reference sets for application software and subsequently test the output against a selection of known-content data sets authored specifically for the project.

The fifth element of the DSRM process model is the **evaluation** of the system, in terms of effectiveness and efficiency. This phase involves performing scientific evaluation of the implemented system design to validate functionality and utility in a real world context. This includes observing and measuring how well the designed artifact supports the objectives of the solution using analysis and metrics. In this research two realistic data sets are tested: 1) A publicly available data set designed specifically for digital forensic research; and 2) A real world data set comprised of second-hand hard drives with unknown and unpredictable data content. The outcome of evaluation is evidential proof of system utility and overall performance.

The sixth and final element of the DSRM process model is the **communication** of research. The major goal of communication is to convey knowledge and findings to the target
audience including researchers and practising professionals. This document communicates a large portion of the research as conducted including in-depth specification of the designed and implemented system. Furthermore, a portion of this research has been communicated via peer reviewed academic publication. Finally, communication is also achieved by openly publishing source code and documentation for the implemented system design in the form of computer software that has been authored and evaluated during the course of this research project.

1.3 System Design and Architecture

The design of an IT artifact is a significant component of a DS research methodology and involves synthesis of a proposed solution to provide an innovative and purposeful solution to the specified research objective \([\text{Nunamaker \\& Chen} \ (1990), \text{Peffers et al.} (2007)]\) state that the artifact can be any designed object with the contribution embedded in the design and where the design process involves determining the required functionality, architecture and creation of an actual artifact. The IT artifact produced in this research is in the form of a **system design.** The required system functionality is to perform effective and efficient automated detection of application software using a reference set populated with known content that can be compared against a target data set. A full chapter is dedicated to the system design which outlines the requirements, design and implementation (see Chapter 5). However, to provide clarity a high-level overview of the system design and architecture is briefly outlined here, as displayed in Figure 1.3.

![Figure 1.3: High-level overview of the system design architecture](Image sources: Mozilla Firefox icon taken from The Mozilla Foundation (2013), Mozilla Thunderbird icon taken from The Mozilla Foundation (2011), Chromium logo taken from The Chromium Authors (2015), TOR icon taken from The Tor Project (2011). All other images are public domain or the author’s original work.)

There are two inputs to the system architecture: 1) An application (e.g., Firefox, Chrome, TrueCrypt and CCleaner); and 2) A target data set (e.g., a perpetrator’s hard drive). The application is reverse engineered to determine the digital artifacts (e.g., data files, Registry entries) that are uniquely associated with the application. Identified digital artifacts are
populated into a reference set. This research uses the term application profile for a reference set designed for storing, distributing and processing digital artifacts specifically from application software. The application profile contains known content, represented by metadata, that can be used to aid detection of the same digital artifacts on a target data set.

The target data set is the second input to the system design. The target is processed and a metadata representation of the original evidence source generated. The application profile can then be compared to the metadata representation of the target data set and digital artifact matching performed. A set of prescribed matching methods are performed for each entry and various metadata properties are correlated to determine if a match can be achieved. Any matching entry is reported to the investigator. Finding such a match is digital evidence that a digital artifact (e.g., data file) resides on the target data set which is known to be uniquely associated with a known application. Essentially, this provides robust evidence that the selected application is present on the suspect’s system and can also provide evidence of the user who installed or ran the software.

1.4 Research Contribution

Numerous theoretical and practical contributions to the field of digital forensics have resulted from this research. The overall research contribution is the design, implementation, demonstration and evaluation of a system to automate application software reference set creation and subsequent correlation against a target data set. The following list outlines the major methodological contributions made in this research project.

1) Improved application software reference sets for digital forensics: This research advanced system-level reverse engineering methods to incorporate digital forensic requirements. An automated data collection method was designed, removing time-consuming manual analysis and post-processing requirements. A scalable blacklisting method was designed to automate filtering of irrelevant digital artifacts that are not unique to the application. Although implemented for Microsoft Windows, the data collection method, differencing strategy and blacklisting design are portable to other operating systems.

2) Application profile data abstraction: An application profile is a reference set for application software which is populated with known content. A data abstraction was specifically designed in this research to address digital forensic requirements. The Application Profile XML (APXML) abstraction provides functionality to store, distribute and automate processing of application profiles. The data abstraction provides functionality to store multiple digital artifact types with accompanying detailed metadata.

3) Advanced digital artifact matching methods: Previous digital forensic reference sets are limited to performing known file filtering using only cryptographic hash values. This research designed advanced matching methods to include file system and Windows Registry entries and correlation using multiple metadata properties; for example, data
file matching is performed using the file name, file size, logical file system location (absolute path), cryptographic hash value and allocation status (deleted or not). The created matching methods solved the following complex matching scenarios:

a) **Different software versions:** The implemented matching method proved effective at detecting digital artifacts from application software from a different version than that used to create an application profile; for example, TrueCrypt version 6.3a versus TrueCrypt version 7.1a. This is important as creating and maintaining an application profile for every software version is not feasible.

b) **Different operating system versions:** The implemented matching methods proved effective at detecting digital artifacts on different operating system versions than that used to create the application profile; for example, Windows XP versus Windows Vista. Similar to different software version problems, this is important as creating an application profile for the same application on different operating system versions is time consuming.

A variety of applied contributions have been made in the form of a data abstraction implementation as well as a selection of computer forensic tools to aid in performing a variety of tasks in the overall system design. The tools were designed based on the objectives of a solution, implemented in the form of computer software, demonstrated to convey functionality and evaluated in terms of an overall system design. The applied contributions that have resulted from the conducted research are as follows:

1) **apxml.py:** An Application Programming Interface (API) is provided to automate processing of APXML documents using an object oriented model with built-in type checking to create objects that represent the original information in an APXML document. The API includes functionality to read, process, modify and write APXML documents, as well as built-in functions to accomplish common processing tasks. The API is distributed with a collection of scripts to automate common processing tasks.

2) **LiveDiff:** A portable live differential forensic analysis tool to automate profile creation that is designed specifically to address digital forensic requirements. It performs system snapshots before and after a single action (e.g., installing an application), identifies file system and Registry changes (new, modified and deleted entries) and populates all results in an APXML document.

3) **Vestigium:** A forensic analysis framework to automate the correlation of application profile(s) entries against target data set entries. The tool is designed to detect known content from application profiles that reside in a target data set. Matching is performed by correlating digital artifact metadata, in contrast to previous solutions which only match using hash values. Any detected entries are reported to an investigator in two formats: 1) a human-readable; and 2) a machine-readable data abstraction.

All authored tools are licensed under open source licenses (including GNU GPL and The MIT License). It is hoped that this will promote tool usage by other researchers and practitioners,
as well as further development of the tools produced in this research project. All source code for each tool listed above is freely available from the author’s GitHub repositories.

1.5 Research Publications

A collection of academic publications have resulted from the research conducted. The first paper was published early in the author’s research journey and focussed on the initially proposed research area, namely file carving. Laurensen (2013) outlined a performance analysis of popular file carving tools to effectively perform data recovery in a variety of complex data file structures and file types. Although the topic presented in this paper eventually changed, the tools, metrics and data abstractions used were also utilised in this thesis research.

Another paper published was a collaborative study with a fellow postgraduate student. Although not directly related to the thesis research presented, the paper utilised a variety of underlying methods taken from this research; for example, the use of the DFXML forensic data abstraction to process forensic analysis results. Gee, Laurenson, and Wolfe (2015) outlined on detection and validation of Inland Revenue numbers, a variation of the American Social Security Number (SSN) used for taxation purposes. The publication won best paper in the Digital Forensics Conference stream, as well as the best overall paper at the 2015 SRI Security Congress (out of a total of 46 papers).

In the later stages of the author’s candidature, a paper was published based on one stage of the system design and approach presented in this research. Laurenson, MacDonell, and Wolfe (2015) presented the design and implementation of the application profile component of the system design, including the data collection method (the LiveDiff tool) and application profile data abstraction design (the APXML document structure). All published academic articles are listed below:

Published academic articles:


Although not yet produced, it is envisioned that two additional academic papers will be authored based on the research presented in this thesis. The listing below specifies the section numbers from which the research is to be sourced and the accompanying paper titles:

1See: https://github.com/thomaslaurenson/
1) Sections 5.3, 6.3 and 6.4 Filtering irrelevant digital artifacts from differential forensic analysis using set intersection and dynamic blacklisting

2) Sections 5.5, 6.6 and 7.2 Advanced matching methods using rich metadata to correlate file system and Registry artifacts

1.6 Thesis Structure

Chapter 1 has provided an introduction to the proposed topic of this research project. The incentive to conduct additional research in the discipline of digital forensics was largely motivated by the number of existing research and practical problems that prevail in this field. The research methodology outlined the DSRM process model to function as the guide for this research project and a high-level research objective was identified; an intent to design, implement and evaluate a system to perform automated reference set creation and subsequent automated correlation against a target data set. Numerous theoretical and practical research outcomes achieved throughout the duration of the project have contributed relevant and useful resources to the field of digital forensics.

The structure for the remainder of this thesis is outlined below. As the DSRM process model has been chosen as the blueprint for the project, the chapter structure of the thesis should be based on the different phases of the model as represented in Figure 1.4.

Chapter 2 presents background information regarding the field of digital forensics, including documented procedures, techniques, tools and digital evidence sources. Many current digital forensic research challenges are discussed but in particular the focus is concentrated on anti-forensics, the techniques and tools that are available which have the capability to jeopardise digital investigations. A selection of anti-forensic problems are then identified.

Chapter 3 provides an in-depth review of literature of specific relevance to the research conducted and reported here. A review of previous academic research regarding anti-forensic tool detection is presented. A thorough analysis of reverse-engineering techniques and tools is conducted and associated limitations of current approaches specified. Finally, various automated digital forensic analysis techniques and associated tools are investigated and the associated research challenges specified.

Chapter 4 outlines the complete research methodology. Various frequently used digital forensic research methodologies are first reviewed supporting and justifying the use of design science for this research project. Each phase in the DSRM process model is then described in detail. The research problem is reiterated from the background and literature review chapters.
The objectives of a solution are defined with one high-level objective and various lower-level research objectives to aid in building a robust solution. An overview of experimental testing is provided including data sets for testing and various metrics to measure system effectiveness and efficiency.

Chapter 5 specifies a system design to be implemented with the required functionality to address the particular research objectives. Firstly, the system architecture is introduced with the necessary specification and divided into the required components. The actual design process is then conducted for each distinct component. Information relevant to software development is outlined that assists in the application of the specified design in a selection of computer forensic tools and the end result is an implemented design ready to be executed against a target data set.

Chapter 6 covers the use of the implemented system as it is demonstrated against a selected data set to test functionality in a laboratory controlled environment. This process validates the system design, confirming it can solve the specified research problems. The operation of the authored tools is presented followed by a summary of the experimental testing method for both demonstration and evaluation purposes. The system is then demonstrated by performing testing against a data set authored specifically for this research which contains known content. Refinements and modifications to the system design and the associated implemented tools are finalised, rechecked and performed before proceeding to the evaluation stage.

Chapter 7 contains a thorough evaluation of the implemented system against two different data sets on which to assess performance. A public data set designed specifically for digital forensic research is implemented and a methodical review of effectiveness and efficiency is carried out. System design refinements and modifications are presented from the further lessons learned from evaluation. Finally, a real world data set comprised of hundreds of second-hand hard drives with unknown data is tested. This data set was especially selected as it provides more unpredictable and diverse data, resulting in more robust research findings and potential software errors in a variety of non-laboratory controlled data. In closing, a discussion summarises the overall system performance in terms of solving the research problem and associated research objectives.

Chapter 8 presents the conclusions of the research conducted including future research avenues. Appendices contain supplementary material not covered in the main text of the thesis including additional tables and figures as well as some smaller pieces of programming code not associated with the software tools released as a result of this research.
Chapter 1 provided an introduction to this research project. An escalation of current investigation and technical challenges faced by digital forensic researchers and practitioners has led to clear-cut motivations for research in the field of automated forensic analysis with specific regard to the detection of anti-forensic tools. The Design Science Research Method (DSRM) process model was specified to guide the project and the high-level research objective was stated as the design, implementation and evaluation of an effective and efficient system to automate the detection of anti-forensic tool activity on a target data set.

This chapter presents background information pertinent to the underlying foundations of this research. An exhaustive background is not intended; rather the goal is to provide an overview of the relevant technologies, processes and tools to inform the reader. Section 2.1 is an overview of digital forensics, and more general digital investigations, including the digital forensic procedure while Section 2.2 outlines digital evidence sources and common forensic data abstractions. Section 2.3 presents a collection of digital forensic research challenges including data volume, tool scalability, case complexity, technology advancements and the CSI effect. Section 2.4 provides an overview of application software and the techniques and tools commonly used to perform manual and automated forensic analysis of application software. Section 2.5 introduces the important aspect of anti-forensics including the techniques and tools followed by a summary of the challenges surrounding anti-forensic usage.
2.1 Digital Forensics

The investigation of digital information is known as digital forensics, a branch of forensic science involving the investigation of any digital or electronic component (Reith et al., 2002). Digital forensics encompasses a wide variety of digital evidence sources and, therefore, has sub-disciplines including computer forensics, network forensics and Internet forensics.

Computer forensics usually refers to the forensic examination of computer components and their contents such as hard drives, compact disks, and printers. However, the term is sometimes used more loosely to describe the forensic examination of all forms of digital evidence, including data travelling over networks (a.k.a. network forensics). (Casey, 2011, p. 3).

A variety of definitions have been proposed for the digital forensic procedure. However, these definitions all outline similar key principles and investigation phases. The Digital Forensic Research WorkShop (DFRWS) technical report, A Road Map for Digital Forensic Research, defines digital forensics as:

The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations (Palmer, 2001, p. 16).

The American National Institute of Standards and Technology (NIST) Special Publication: Guide to Integrating Forensic Techniques into Incident Response, defines digital forensics as:

Digital forensics, also known as computer and network forensics, has many definitions. Generally, it is considered the application of science to the identification, collection, examination, and analysis of data while preserving the integrity of the information and maintaining a strict chain of custody for the data (Kent et al., 2006, p. 9).

Although there are minor differences between the definitions and specified investigation phases, the same overall principle applies; that is the use of scientific methods to determine digital events based on reliable digital evidence. Similar to traditional forensic science, digital forensics involves conducting an investigation.

A digital investigation is a scientific process where hypotheses are developed and tested in an attempt to answer specific questions about digital events (Carrier, 2005). The digital evidence obtained can be used to provide information to determine particular events that have occurred to either support or refute the proposed hypotheses. There is not always a criminal element associated with such an investigation which may simply involve investigation
of any electronic event or information. A similar investigation scope is a digital forensic investigation. “The main difference between a digital investigation and a digital forensic investigation is the introduction of legal requirements” (Carrier, 2005, p.13), ensuring that a chain of custody is documented for all collected evidence. Ultimately, the outcome of both investigation types provides evidential proof of specific digital events. However, the additional requirement of producing viable and sound evidence which can be made available and is admissible in civil or criminal legal proceedings is the definable distinction of a digital forensic investigation.

2.1.1 Digital Forensic Procedure

Digital forensics requires a rigorous and tested procedure so that the resultant digital evidence can withstand contested scrutiny in a court of law. Specific standardised guidelines have therefore been developed so that the investigation meets the necessary criteria. Figure 2.1 displays the overall digital forensic procedure.

![Digital forensic procedure](image)

Figure 2.1: Digital forensic procedure (Source: Figure adapted from Palmer (2001), Kent et al. (2006) & Slay et al. (2009))

As displayed in Figure 2.1 the digital forensic procedure is comprised of the identification of potential sources of digital evidence, the collection of identified digital evidence, the examination and analysis of evidence to determine specific digital events and lastly the reporting and presentation of digital evidence. Underlying all of these phases is the preservation of digital evidence that must be maintained throughout the entire digital forensic procedure. The following subsections outline and discuss high-level information regarding the processes conducted at each phase.
2.1.1.1 Identification of Evidence

The initial digital forensic phase is to identify potential sources of evidence. McKemmish (1999) states that the identification of evidence involves determining the presence of any electronic device capable of storing digital data including establishing the type and format in which the data is stored. Table 2.1 displays a list of electronic devices, specified by the American National Institute of Justice (NIJ), that are commonly encountered at a crime scene. Associated examples are included.

<table>
<thead>
<tr>
<th>Electronic Device</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer systems</td>
<td>Desktop computers, laptops, tablets, servers</td>
</tr>
<tr>
<td>Storage devices</td>
<td>Hard Disk Drive (HDD), external HDD, memory cards and other removable media</td>
</tr>
<tr>
<td>Handheld devices</td>
<td>Mobile phones, smart phones, Personal Digital Assistant (PDA), Global Positioning Systems (GPS)</td>
</tr>
<tr>
<td>Peripheral devices</td>
<td>Keyboards, webcams, microphones</td>
</tr>
<tr>
<td>Network devices</td>
<td>Internet modems, network hubs, network switches, wireless Access Points (AP), wireless cards/adapters</td>
</tr>
<tr>
<td>Other devices</td>
<td>Surveillance equipment, digital and video cameras, audio recorders and players, video game consoles, card readers</td>
</tr>
</tbody>
</table>

There are also a variety of other sources of digital evidence which may not be physically present at a crime scene; for example, a PC identified at a crime scene may contain records of a user’s Internet history. However, an Internet Service Provider (ISP), not located at the physical crime scene, may also retain a log of a user’s Internet history. Additionally, when a user visits a web site, the server hosting the web site may also retain logs of the user’s activity on the site. These examples illustrate that digital evidence may be geographically dispersed and that digital investigators need to be aware of this fact.

2.1.1.2 Collection of Evidence

According to Kent et al. (2006) the collection, or acquisition, of evidence should be performed using a three step process: 1) Developing a plan to acquire the data; 2) Collecting the data from identified evidence sources; and 3) Verifying the integrity of the collected data.

As there are multiple potential sources of digital evidence a plan is formulated at the outset to first, identify the sources of evidence then, how best to acquire the identified evidence. Kent et al. (2006) state that two major factors influence the preparation of an evidence collection plan: 1) Estimating the likely value of evidence; and 2) Determining the volatility of the data to be collected. The likely value of evidence from various sources is based on a practitioner’s understanding and previous experience involving similar investigative scenarios. In terms of volatility, it is important to recognise that certain digital evidence may be irretrievable if
the power source is removed from an electronic device. Volatile data therefore, needs to be collected from a running system.

It is essential that digital evidence is collected using tested, reliable and proven techniques. [Palmer (2001)] states that collection of evidence from identified sources involves using approved methods together with hardware and software solutions to acquire the data. The *Computer Forensic Tool Testing Handbook*, authored by the NIJ, specifies a list of disk imaging, software write blockers and hardware write blockers that have been certified under the Computer Forensics Tool Testing (CFTT) program [NIJ (2015)]. Disk imaging tools provide the capability to produce a forensic (bit-by-bit) copy of a target hard drive, while software and hardware write blockers provide the functionality to ensure that no data can be written to a target hard drive during evidence collection. The use of these tested and verified tools ensures that data integrity is maintained during the process of evidence collection.

2.1.1.3 Examination of Evidence

[Palmer (2001)] states that examination of digital evidence involves filtering techniques, pattern matching, hidden data discovery while all the time maintaining traceability and validation of original evidence. [Kent et al. (2006) p. 2-2], states a similar yet expanded definition of evidence examination which involves “forensically processing large amounts of collected data using a combination of automated and manual methods to assess and extract data of particular interest, while preserving the integrity of the data”. Furthermore, the examination of evidence requires performing targeted investigation based on the criteria of the case. [Reith et al. (2002)] states an in-depth systematic examination is required with specific emphasis relating to the suspected crime; for example, in an electronic fraud investigation the relevant evidence might include the discovery of credit card and bank account numbers.

These definitions of the examination phase of digital forensics relate to the extraction of relevant data from the original data set that may be useful to the digital investigation, yet still requires further processing and additional analysis. Most digital forensic procedures specify a separate examination and analysis phase, “typically, an analyst examines data and performs analysis of that data, then conducts additional examination and analysis based on the results of the initial analysis” [Kent et al. (2006) p. 3-6]. The examination and analysis phases are thus inherently intertwined.

2.1.1.4 Analysis of Evidence

Essentially, the resultant output from the above examination phase is extracted information from the initially collected evidence which now requires further processing. The additional analysis of this digital evidence serves to draw an appropriate conclusion based on the available information, or otherwise to determine that no conclusion can be drawn (Kent et al. 2006). N. Beebe and Clark (2005b) state that the questions needing to be put forward during the digital forensic analysis phase include the who, what, when, where, why and how type questions by surveying, extracting and reconstructing evidence. In turn, the answered questions should
help identify people, places, items and events, and determine the relation of these elements to draw an appropriate conclusion (Kent et al., 2006).

Casey (2013) states that there are three main levels of forensic analysis: 1) Survey/triage forensic analysis; 2) Preliminary forensic analysis; and 3) In-depth forensic analysis. Each level has a specific objective and investigation scope as outlined in Table 2.2. The variety of levels used in forensic analysis is dependent on the type of analysis being conducted where a variety of techniques and tools may be implemented by an investigator.

<table>
<thead>
<tr>
<th>Analysis Phase</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triage forensic analysis</td>
<td>Triage forensic analysis is a targeted review of all media involved in an investigation to discover the most useful evidence sources for further processing</td>
</tr>
<tr>
<td>Preliminary forensic analysis</td>
<td>Forensic examination of evidence identified from the triage phase which helps investigators determine information to aid the digital investigation</td>
</tr>
<tr>
<td>In-depth forensic analysis</td>
<td>Comprehensive examination of evidence that requires additional investigation to provide a more complete understanding of the evidence in question in order to help answer the investigation hypothesis</td>
</tr>
</tbody>
</table>

### 2.1.1.5 Reporting of Evidence

The final phase of the digital forensic procedure is the reporting, or presentation, of digital evidence. The main purpose is to convey the results of the detailed forensic analysis including documenting the actions performed and how different techniques and tools were used (Kent et al., 2006). The summarised explanation and conclusions conveyed from the analysis phase should be written in non-technical English, clearly expressed and using terminology that is easily understood (Reith et al., 2002).

The reporting phase can vary greatly depending on the case or scenario of the digital investigation. Nevertheless, “writing a report is one of the most important stages of the process because it is the only view that others have of the entire process” (Casey, 2013, p. 175). Additionally, unless the procedures, findings and conclusions are well articulated it is unlikely that the evidence will be considered viable or significant.

Closely related to reporting evidence is presentation evidence which has variations to solely reporting. N. Beebe and Clark (2005b) state that the presentation of evidence involves communicating findings (either written, oral or both) to a diverse audience including law enforcement, management, technical, judiciary and legal personnel. Ultimately, the goal of evidence presentation is to convey the findings of a digital forensic investigation in a succinct and detailed manner to relay the information and the conclusions drawn from analysis.
2.1.1.6 Preservation of Evidence Integrity

It is a priority that digital evidence is preserved throughout the entire digital forensic procedure including the identification, collection, examination, analysis and reporting of digital evidence.

In order to be able to prove that the evidence presented in a court of law is forensically sound, the integrity of the evidence has to be preserved, and the complete documentation trail (referred to as the Chain of Custody) has to be provided (Flaglien, Mallasvik, Mustorp, & Årnes, 2011, p. 122).

According to Roussev, Chen, Bourg, and Richard III (2006), hashing is a fundamental tool in digital forensics to ensure data integrity. One-way cryptographic hash functions can be used to verify the integrity of acquired evidence by generating a hash value during the collection of evidence and using that generated hash value to compare at a later date. If the hash values match it is assured that evidence integrity has been maintained.

2.1.2 Digital Forensic Triage

Digital forensic triage can be considered as a starting point to forensic examination and analysis of evidence. In generalised terms, “triage is a term widely used to denote the prioritization of work according to a quality inherent in the objects being acted upon” (Garfinkel, 2013b, p. 57). A well known example is the sorting of medical patients based on the urgency of care required. Digital forensic triage follows the same principles, applying precedence in the examination and analysis of investigation targets to determine potentially useful evidence sources; for example, a triage examination to discover potential electronic fraud consisting of multiple digital devices in an investigation (e.g., a desktop computer, laptop and smartphone) might yield results that dictate that one device (e.g., the laptop) has a higher number of credit card numbers and warrants further investigation before other devices.

Numerous definitions have been proposed for digital forensic triage. Roussev, Quates, and Martell (2013) state that triage is a fast initial screen of (potential) investigative targets in order to estimate their evidentiary value. Garfinkel (2013b) states triage also includes rapid and largely automated analysis performed when evidence is first encountered and can reveal the intelligence value of media and prioritisation of further in-depth analysis. The scope and methods used to perform triage in digital investigations has been debated amongst researchers and practitioners. Some state it is completely independent of the forensic procedure, while others emphasise that it is linked with the forensic procedure as part of the examination of evidence. The definition promulgated by Roussev et al. (2013), argue that triage can be included in the examination phase of the digital forensic procedure and can be defined as:

**Triage:** A partial forensic examination conducted under (significant) time and resource constraints (Roussev et al., 2013, p. 160).
Others, such as Casey (2011) include triage as being the starting level for analysis of evidence and as such has been adopted for this research project. Also, for the sake of clarity, the examination and analysis phases in this research will refer to the generic term forensic analysis, while digital forensic triage is referred to simply as triage. The difference between the two methods is only the time constraints of triage when compared to forensic analysis.

The overall digital forensic procedure aims to identify, collect, examine, analyse and report useful digital evidence to prove, or disprove, a particular electronic event. The procedure is inseparable from digital evidence as the source of information which is now discussed further in the following section.

2.2 Digital Evidence

The National Institute of Justice (NIJ) defines digital evidence as “information stored or transmitted in binary form that may be introduced and relied on in court” (NIJ 2008, p. 52). Digital data is essentially a collection of binary data that represent higher-level information such as text, audio, video and images. This section outlines information pertaining to digital evidence starting with a generalised overview of digital artifacts and their use in digital investigations. A discussion of evidence sources used in this research includes file systems and the Windows Registry. Finally, a summary of digital forensic data abstractions cover forensic disk images used to store collected digital evidence followed by high-level data abstractions used to enhance forensic examination and analysis.

2.2.1 Digital Artifacts

Using a practice similar to archaeologists, who seek to understand past human behaviour by studying artifacts, digital investigators similarly study digital artifacts to understand past behaviour in the digital world. N. Beebe (2009) states that digital artifacts are a function of the physical media, operating system, file system and user-level applications which impact on what digital evidence is created when a user interacts with a computer system. Digital artifacts can include a wide variety of different types such as data files, system configuration settings, information from volatile computer memory and network traffic packets. Table 2.3 displays common digital artifact types that are encountered in digital investigations.

During the course of a digital investigation, especially the examination and analysis of digital evidence, an investigator seeks to extract these digital artifacts, analyse the contents and associated metadata to reconstruct a digital event. An example of a common digital artifact is a word processing document (e.g. report.doc) which is a data file. The contents of the document can be examined to determine what it stores; for example, opening the document using a native application (e.g., Microsoft Word) to determine the contents. In addition, data files also have associated file metadata; for example, timestamp information including the modified, accessed and created (MAC) times. Thus, the information obtained during the analysis of digital artifacts serves to determine past behaviour on a system.
Table 2.3: Types of digital artifacts

<table>
<thead>
<tr>
<th>Digital Artifact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data files</td>
<td>Structured data created and used by user-level software such as word processors and image editors; for example, documents (report.doc), photos (image.jpg), multimedia files (video.mpg).</td>
</tr>
<tr>
<td>System configuration</td>
<td>Structured data stored by an operating system for configuring the system such as device drivers, services or interface options; for example, Windows Registry and Apple Property Lists.</td>
</tr>
<tr>
<td>Network traffic</td>
<td>Digital data communicated between two devices in a computer network; for example, Internet Protocol (IP) packets.</td>
</tr>
<tr>
<td>Volatile memory</td>
<td>Volatile information stored in RAM; for example, encryption keys, details of running processes.</td>
</tr>
</tbody>
</table>

2.2.2 Digital Evidence Sources

Digital artifacts are a valuable source of evidence in digital investigations. This section comments further on evidence sources and associated digital artifacts of interest to this research; in particular the Microsoft Windows operating system platform. The file system as an evidence source is first outlined, followed by information pertaining to the Windows Registry both of which are intended to provide a generalised overview relating to the scope of this research. References are provided to direct the reader to additional material if required.

2.2.2.1 File System

In the field of computing, a file system defines how digital data is controlled including storage, naming conventions, organisation and access (Kent et al., 2006). The file system itself is located on a partition of the physical digital storage device and are primarily used on digital storage devices to provide a structure to store data and an index to point to the data stored. A large variety of file systems exist and each has differences in structure, properties, functionality, security and intended application. The remainder of this section provides a brief overview of file systems for the scope of research presented in this project. Further information regarding physical hard drives, other digital storage devices, data encoding principles, in-depth information on partitions and specific information regarding forensic analysis of different file system types can be found in *File System Forensic Analysis* by Carrier (2005).

According to Carrier (2005), common examples of file systems include the File Allocation Table (FAT) and New Technology File System (NTFS) primarily for Microsoft Windows systems. The Unix File System (UFS) and Extended (EXT) file system are prominent on Linux systems, while the Hierarchical File System (HFS) on Apple systems. Most file systems can be classified based on the type of digital media they were originally intended for; for example, the FAT file system was designed for block devices starting with floppy disks and later for Hard Disk Drives (HDD). The Network File System (NFS) is a network protocol that was designed and is used for file system access over a network, while the International Organization for Standardization (ISO) 9660 file system is designed specifically for Compact Discs (CD).
Although many types of file systems exist with unique structures and functionality, all share common characteristics such as the concept of directories and files to store and organise digital data. Directories, also called folders, are organisational structures that act as a catalogue provide the functionality for the operating system and allow the user to store and group files into logical collections; for example, on a Microsoft Windows systems the Windows directory stores almost all operating system files\(^1\) while the My Documents directory stores a user’s documents. According to [Kent et al. (2006)](kent2006), a file, also called a data file or computer file, is a collection of digital information that is logically grouped into a single entity and referenced by a unique file name. A huge variety of data file types exist including common examples such as documents, images, and video files.

### 2.2.2.2 Windows Registry

According to [Microsoft (2002)](microsoft2002), the Windows Registry is a central hierarchical database used to store configuration information for users, applications and hardware devices. The Registry was introduced in Windows 3.1 and replaced the text-based configuration files (.ini files) used in MS-DOS. The Registry is an exceptionally valuable forensic resource containing rich user and application information, especially when attempting to establish a timeline of system activity (Carvey, 2011). This section provides a general commentary of the Windows Registry for the scope of this research. Additional information regarding the internal structure of the Registry, a summary of common Registry analysis tools and case studies of Registry analysis can be found in *Windows Registry Forensics* by [Carvey (2011)](carvey2011).

Although the Registry is presented as a hierarchical database the different components of the Registry are stored in a collection of local data files referred to as hives. “A hive is a logical group of keys, subkeys, and values in the registry” [Microsoft (2016b)](microsoft2016b) para. 1). When the system starts the required hives are loaded; for example, the NTUSER.DAT hive file is loaded into the HKEY_USERS (HKU) and HKEY_CURRENT_USER (HKCU) Registry root keys. The core system hive files (SAM, Security, Software and System hive files) are located in the Windows directory (Windows\system32\config), while each user has an individual NTUSER.DAT hive file in their associated user profile directory (Carvey, 2011). Registry hives have a binary structure comprised of different cells, sometimes referred to as record types\(^2\). Registry hives are stored as data files by the file system, and similarly to file systems they have unallocated (slack) space (Carvey, 2011). Previous research has investigated recovery of deleted Registry entries by investigation and extraction of information from the slack space in a Registry hive file (Morgan, 2008).

Figure 2.2 displays a visual representation of different Windows Registry elements including hive files, keys, subkeys, values, value types and actual value data. The interface is displayed from the Windows Registry Editor, or regedit.exe, utility. As seen, the Reg-

---

\(^1\)The Windows directory is a default setting that can be changed based on user preference.

\(^2\)A discussion of the internal Registry structure goes beyond the scope for this discussion. Additional information can be found in Chapter One of *Windows Registry Forensics* by [Carvey (2011)](carvey2011) and *The Windows NT Registry File Format* by [Morgan (2009)](morgan2009).
Registry is organised into a tree structure very similar to that of a file system (Morgan, 2008). Registry keys are similar to file system directories, while Registry values are analogous to data files that store discrete portions of raw data in a single entity. Registry keys are different from Registry values in that the only timestamp information retained in the Registry is the LastWriteTime (modification time) which is only stored by keys, not values. However, Registry values have additional information including a value name, value data type and value data. The data type can be specific and is used to store strings, numbers and binary data.

### 2.2.3 Digital Forensic Data Abstractions

In the field of digital forensics, data abstractions are primarily used to store and represent digital evidence. This includes storing collected digital evidence while maintaining the integrity of the original source, as well as transforming digital evidence into different representations to ease forensic examination, analysis and reporting. Table 2.4 displays five widely used data abstractions with accompanying examples of the type of digital evidence they can store.

<table>
<thead>
<tr>
<th>Data Abstraction Type</th>
<th>Evidence Contained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Images</td>
<td>Digital data storage device; for example, hard drives or flash memory cards</td>
</tr>
<tr>
<td>Data Files</td>
<td>Structured data files usually to present digital evidence or transfer evidence between investigators; for example, documents and images</td>
</tr>
<tr>
<td>Reference Sets</td>
<td>A document containing known files represented by Message Digest version 5 (MD5) &amp; Secure Hash Algorithm (SHA) hash values</td>
</tr>
<tr>
<td>Extracted Name Entities</td>
<td>Information extracted from an investigation target including email addresses and credit card numbers</td>
</tr>
<tr>
<td>Packet Capture Files</td>
<td>Intercepted network traffic usually archived in data files</td>
</tr>
</tbody>
</table>

The following subsections briefly outline the digital forensic data abstractions used in the
course of this research for the storage of digital evidence. A summary of forensic disk images is first given followed by a brief overview of high-level data abstractions designed specifically to advance digital forensic research and development.

### 2.2.3.1 Forensic Disk Images

Forensic disk images, also known simply as disk images or evidence files, are the most common forensic data abstraction. Disk images are a bit-by-bit (or block-by-block) copy of a digital data storage device such as a Hard Disk Drive (HDD). They provide the ability to store and preserve the integrity of digital evidence by generating an exact duplicate of the source media ([Altheide & Carvey, 2011](#)). Table 2.5 displays three of the most frequent disk image formats used in digital investigations.

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Extension</th>
<th>Integrity</th>
<th>Compression</th>
<th>Software Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>.RAW</td>
<td>None</td>
<td>None</td>
<td>Extensive</td>
</tr>
<tr>
<td></td>
<td>.001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert Witness</td>
<td>.E01</td>
<td>Entire MD5</td>
<td>Zlib</td>
<td>libewf library</td>
</tr>
<tr>
<td>Format (EWF)</td>
<td>.EX01</td>
<td>Block CRC</td>
<td></td>
<td>Multiple toolkits</td>
</tr>
<tr>
<td>Advanced</td>
<td>.AFF</td>
<td>Entire MD5</td>
<td>Zlib</td>
<td>afflib library</td>
</tr>
<tr>
<td>Forensic Format (AFF)</td>
<td>.AFM</td>
<td>Block MD5</td>
<td></td>
<td>The Sleuth Kit</td>
</tr>
</tbody>
</table>

A raw disk image is commonly referred to as the `dd` image format because the file is derived from the output of the `dd` tool found on UNIX and UNIX-like operating systems. The primary purpose of the `dd` tool is to convert and copy a file ([IEEE Std 1003.1, 2008](#)) and has previously been used to populate data in a disk image. However, since the file format was not designed to store digital evidence it lacks essential digital forensic functionality; for example, the capability to store an acquisition hash value to ensure evidence preservation. Hashing limitations have been addressed by forensic software developers and modified versions of the `dd` tool made available; for example, `dcfldd`[^3] is a forked version of the `dd` tool, while `dc3dd`[^4] is a patched version of the `dd` tool. Both tools provide additional functionality to better support forensic use such as the inclusion of hashing for the purpose of evidence integrity preservation ([Altheide & Carvey, 2011](#)).

Expert Witness Compression Format (EWF) is a proprietary disk image format maintained by Guidance Software. EWF is considered to be the de-facto standard as it has, in practice, been validated and approved to be forensically sound ([Flaglien et al., 2011](#)). The EWF file format is also commonly referred to as `E01`, owing to the file extension it

[^3]: Dcfldd was created for the Defence Computer Forensics Laboratory by Nick Harbour and is available from [http://dcfldd.sourceforge.net/](http://dcfldd.sourceforge.net/)
[^4]: Dc3dd was created for the Department of Defense Cyber Crime Center by Jesse Kornblum and is available from [http://sourceforge.net/projects/dc3dd/](http://sourceforge.net/projects/dc3dd/)
uses. EWF provides the means to store disk, partition and memory images and supports data compression and splitting disk images into multiple segments (Metz, 2006). Guidance Software introduced EWF version 2 with new functionality which includes support for data encryption, additional compression algorithms and increased data acquisition performance (Guidance Software, 2012). Since EWF is a proprietary format, the libewf project was developed to allow ease of integration of the EWF disk image format into other forensic tools. The libewf project is an open source library which provides: 1) The ability to add EWF support to forensic tools; and 2) A selection of command line tools to work with EWF files including the ability to acquire, mount and verify EWF disk images.

The Advanced Forensic Format (AFF) disk image format was developed as an alternative to the existing proprietary disk EWF format. According to Garfinkel, Malan, Dubec, Stevens, and Pham (2006), AFF has three significant benefits over other forensic disk image formats: 1) Open source implementation and support for multiple operating systems; 2) Additional flexibility provided by the ability to store extensive metadata; and 3) Decrease in required disk space by implementation of better data compression techniques. Similar to the libewf project, the afflib project has a selection of command line tools to work with AFF files including the ability to acquire, mount, convert and compare AFF evidence files.

2.2.3.2 High-level Forensic Data Abstractions

Table 2.4 displayed a selection of data abstractions that are frequently used in digital forensics. However, these have challenges as there are no standardised techniques for forensic tools to ingest input or store output. Almost every tool implements a unique output format, resulting in tool output incompatibility. The lack of standardised and widely accepted high-level data abstraction decreases forensic tool composibility; that is, the output from one tool cannot directly be used as input to another forensic tool. Furthermore, research and technological advances have been constrained due to the limitations and variability of existing formats. Garfinkel (2010) states that a key component to improving digital forensic research is the adoption of high-level data abstractions for storage, processing and documenting digital evidence. There have been numerous attempts to develop abstraction methods, especially for different digital artifacts types. A variety of approaches have used an Extensible Markup Language (XML) approach to satisfy the storage and processing requirements of a digital investigation. The use of XML is advantageous as it is easy to create structured documents, is human-readable, machine-readable and can be validated through the use of an XML Schema (Yergeau, Bray, Paoli, Sperberg-McQueen, & Maler, 2004).

Recently, a diverse range of XML-based data abstractions designed for digital forensics have emerged. Alink, Bhoedjang, Boncz, and de Vries (2006) outlined an XML Information Retrieval Approach to Digital Forensics (XIRAF) which described a framework for feature extraction and forensic analysis using XML database technology for flexible and powerful querying. By storing extracted features in XML, the system design provides a clear separation.
between examination and analysis, while providing a common output format to enable better tool integration. Levine and Liberatore (2009) outlined the Digital Evidence Exchange (DEX) data abstraction which aimed to provide digital evidence provenance, the ability to reproduce evidence from the XML description and the ability to enable tool comparison and validation. The DEX standard was released together with the initial publication of the corresponding paper, but has not been updated since 2009.

A more successful effort at implementing a high-level forensic data abstraction using XML is Digital Forensic XML (DFXML). Garfinkel (2012a) states that DFXML is an XML language designed to represent a wide range of forensic information, specifically used to describe digital artifacts and also to help simplify advanced forensic processing and automated analysis. Initial development of DFXML was conducted by Garfinkel (2009) when implementing a tool, dubbed fiwalk, which has the ability to produce detailed XML output to describe information about a target evidence file (e.g. sector size), partition information (e.g. file system type), file information (e.g. file name or size) and information about the tool which created the XML file (e.g. tool name and version). According to Garfinkel (2012a), DFXML has the following goals: 1) To complement and augment existing forensic formats; 2) Be easy to add support for generation or incorporation of DFXML into existing tools; 3) Provide human readability; 4) Be free, open and extensible; 5) Provide scalability to allow large and small scale investigations; and 6) Adhere to existing forensic practices and standards. The official DFXML project is available in both C and Python programming languages to allow ease of integration into new and existing tools. Even though DFXML is a relatively new development a number of forensic tools already provide the functionality to output to the DFXML format; for example, the bulk_extractor, hashdeep and photorec tools all provide the ability to output to DFXML reports.

2.3 Digital Forensic Research Challenges

Although there has been steady research and development regarding the techniques and tools used in the field of digital forensics, there are numerous research challenges that still exist. Garfinkel (2010) states that both academic researchers and practitioners are currently facing a digital forensic crisis where technology advances are being diminished, or even lost, due to fundamental changes in the computer industry. The following subsections discuss a range of current digital forensic research challenges including data volume and scalability, case complexity, the CSI effect and legal challenges. The potential of anti-forensic technique presence in investigations is another serious research challenge and, as it is an important aspect of this research, an entire section is dedicated to this material (see Section 2.5).

---

6Additional information and the source code for the Digital Evidence Exchange project can be found at http://forensics.umass.edu/research.php
7The DFXML project is available from https://github.com/simsong/dfxml
2.3.1 Data Volume & Scalability

Data volume is a fundamental challenge in modern digital investigations. Moore’s law states that the number of transistors on integrated circuits approximately doubles every two years (Moore, 1965). Moore’s law is strongly linked to the growth of digital data storage device capacity and the corresponding increasing size of target devices in forensic investigations. “The targets that digital forensic investigations must process continues to grow and the current generation of digital forensics tools is already struggling to deal with even modest-sized targets” (Marziale, Richard III, & Roussev, 2007, p. S80). Garfinkel (2010) states that the growing size of storage devices means that there is insufficient time to create a forensic image of a target device, or to process all of the data found on the device; for example, a 2 terabyte (TB) hard drive is relatively inexpensive but requires approximately 7 hours of processing to capture a forensic disk image.

Large-scale digital forensic investigations present at least two fundamental challenges. The first one is accommodating the computational needs of a large amount of data to be processed. The second one is extracting useful information from the raw data in an automated fashion. Both of these problems could result in long processing times that can seriously hamper an investigation (Roussev, Richard III, & Marziale, 2007, p. 105).

Scalability of current forensic techniques and tools to large investigations is a fundamental problem in digital forensic research and investigations. The exponential growth of data volume is increasing case processing time and in turn the cost of conducting digital investigations. N. Beebe and Clark (2005a) state that digital investigation tools and techniques are constrained by simplistic data reduction and mining algorithms, are not scalable to large data sets, and produce an enormous volume of results which need to be manually analysed. Practitioners are attempting to perform digital investigations on large scale target data sets using first-generation forensic analysis tools. In order to combat these challenges more advanced forensic analysis techniques are required to harness more efficient computational processing methods.

2.3.2 Case Complexity

The case complexity of digital investigations is increasing due to a the rise of technological changes in the field of computing. The prevalence of flash memory, the variety of hardware interfaces in devices and the proliferation of operating systems and file formats all add to an increasing difficulty to perform digital investigations and digital forensic research and development (Garfinkel, 2010). The correlation of digital evidence between multiple electronic devices has also become a growing challenge.

In terms of flash memory, the massive growth of electronic devices with flash-based storage and a plethora of possible hardware interfaces make forensic acquisition of such devices difficult to perform. In addition, the variety of operating system platforms and
versions and the variety of file formats encountered in modern digital investigations all create ongoing challenges. Take the example of smart phones:

Mobile forensics requires procedures that are very specific to device manufacturer and/or model for both collection and analysis. Not only do mobile phones employ a diversity of cables, interfaces and form factors, but the devices also have unique software, memory layouts and storage techniques (Vidas, Zhang, & Christin, 2011, p. S14).

Another technology that increases case complexity is the prevalence of cloud computing; a modern computing service which provides users with remote storage and processing capabilities, usually via the Internet. Cloud computing has the potential to create issues in digital investigations due to the challenge of acquiring digital evidence that is stored at a remote location. “Cloud service providers and customers have yet to establish adequate forensic capabilities that could support investigations of criminal activities in the cloud” (Ruan, Carthy, Kechadi, & Crosbie, 2011, p. 35). Therefore, the ramifications of cloud computing in digital investigations have still to be fully addressed. However, types of digital artifacts that can be useful in these digital investigations are being explored; for example, Quick and Choo (2013) investigated the data remnants from the cloud computing provider Dropbox.

The requirements of correlation between multiple targets in digital investigations is becoming increasingly important. Digital investigations have progressed from forensic analysis of a single device to the need to perform correlation of digital evidence from a large collection and variation of devices from a single investigation target; for example, it is common that the target of an investigation has a laptop, a smart phone and a potential plethora of other electronic devices. However, current forensic analysis techniques are designed to independently examine single sources of evidence and limited support is available to automate the process of correlating evidence between multiple devices (Garfinkel, 2006); for example, it is difficult for investigators to perform correlated analysis of information such as credit card numbers and email addresses between multiple devices. Research and development of automated analysis techniques are required plus next-generation digital forensic techniques and tools in order to keep pace with the expanding digital world in which we live.

2.3.3 Technology Advancement Challenges

Garfinkel (2012b) states that temporal diversity is a major digital forensic research challenge centred around technology advancement challenges of continual software updates which creates a never-ending forensic tool upgrade cycle. The problem is it is essential that computer forensic tools are required to support all new software versions and operating system platforms; for example, whenever new operating system or application software versions are released by vendors, researchers are required to update the techniques and tools used to perform forensic analysis. The inability to update in a timely manner means investigators are unable to perform analysis of new software. Furthermore, “not only must tools support new
targets within days of their release, but they must continue to support old target releases forever” (Garfinkel, 2012b, p. S82). Support must also be provided for all applications on all major operating systems which creates even more challenges.

As a further obstacle, digital forensics must provide support for technology that is constantly changing; for example the advent of Internet services such as Internet applications provided by Google which may change the system or format of services they provide. Such a change results in the need to update the techniques required so that analysis can take place for newly updated systems. An example of temporal diversity can be illustrated using the Mozilla Firefox web browser. Firefox version 3 introduced a completely new database system to store Internet history, bookmarks, form field information and browser cookies (Pereira, 2009). The dramatic change in the database storage method required digital forensic researchers to reverse engineer the new technology in order to develop new techniques and tools for forensic investigation of the same application.

The ongoing evolution of digital technology requires that digital forensic researchers and tool vendors must repeatedly update and add support for new software, as well as new operating systems, file systems and devices. Any new digital technology must be able to be evaluated so that when encountered during an investigation the practitioner has the capability to perform forensic acquisition and analysis of the identified evidence.

2.3.4 The CSI Effect

A unique digital forensic research challenge is the effect that Crime Scene Investigation (CSI) style television shows may have on forensic evidence presented in a court of law. This phenomenon has been referred to as the CSI Effect. “In recent years, the television program CSI and its spin-offs have portrayed forensic science as high-tech magic, solving crimes quickly and unerringly” (Schweitzer & Saks, 2006, p. 357). The specific problem is based on the proposed hypothesis that CSI type shows have raised the expectations of forensic science evidence and that jurors are disappointed by conventional evidence presented in court. To convey this, a survey conducted in 2006 of 1,027 jurors in Michigan discovered that 46% of jurors expected to see scientific evidence in every criminal case (Shelton, 2008).

Shelton, Kim, and Barak (2006) state that the increase in juror’s expectation of forensic evidence is caused by a broader tech effect rather than just crime based television shows. The tech effect is characterised by the advancement of modern technology which has created a heightened reliance on evidence from forensic science to be used in a court of law. In terms of digital forensics, computer-based crime investigations have become a staple of many CSI type shows.

On the screen nearly every DF investigator is trained on every tool; correlation is easy and instantaneous; there are never false positives; overwritten data can frequently be recovered; encryption can frequently be cracked; it is all but impossible to delete anything; and tools never crash. Reality is not so kind (Garfinkel, 2012b, p. S82).

29
Whatever the root cause of the problem, it can be concluded that scientific evidence is an expected requirement for civil and criminal legal proceedings. Therefore, digital forensics requires continual research and development to advance the procedures, techniques and tools used to perform investigations in order to produce even more thorough and reliable digital evidence.

The following section introduces and provides background material specifically regarding application software in the realm of digital forensics. Following this, a section is dedicated to the introduction of anti-forensics with emphasis given to anti-forensic tools, a type of application software that may be maliciously used to obstruct the digital forensic procedure and remove the availability of digital evidence.

2.4 Application Software

Application software is a computer program (or set of programs) that perform specific end-user tasks; for example, web browsers, word processors, image editors, and multimedia players. The digital investigation of application software is a very active research and development area because different types of software allows a user to perform certain tasks that are of forensic interest; for example, using a web browser to access illegal content on the Internet, using file sharing software to distribute copyrighted material or using hacking tools to illegally access a computer system.

Digital investigation of application software commonly involves the analysis of the digital artifacts (see Section 2.2.1) that are created, modified or deleted when a user interacts with an application. Digital artifacts of interest include file system entries (e.g., data files) and system configuration information (e.g., Windows Registry), as well as data remnants left after application usage or uninstallation. Digital artifacts provide a robust source of digital evidence regarding application software usage in specific scenarios and gives insight into the actions conducted by a perpetrator; for example, digital artifacts are frequently used to perform event reconstruction to create a timeline of digital events.

This section first presents a summary of the application life cycle and the different phases in the life span of an application on a computer system. A synopsis on the use of reverse engineering follows to determine the digital artifacts associated with an application and finally, an introduction is made to the digital forensic techniques and tools commonly used to perform the forensic analysis of application software.

2.4.1 Application Software Life Cycle

Application software that is executed on a host operating system is dynamic. This means that the impact on the system that software leaves is dependent on the actions performed by a user; for example, a user might install an application but never use it; this leaves different digital artifacts than if the user executed the software and performed a variety of tasks over a long time span. To understand the complexity of forensic analysis of application software it is
important to outline how digital artifacts are introduced or removed from the host operating system. This process is known as the application software life cycle.

Each application has a life cycle that follows a determinate path and includes phases such as distribution, installation, execution and uninstallation (Davis, Kennedy, Pyles, Strickler, & Shenoi, 2006). During each phase of the application life cycle the program makes changes to the host operating system in the form of digital artifacts that are created, modified and/or removed; for example, when installing an application, various folders, files, and configuration settings are created which provide the resources to run the specific application. When an application is uninstalled the digital artifacts are removed but some residual information may remain.

![Diagram of the application software life cycle]

The application software life cycle illustrated in Figure 2.3 only displays a high-level overview of the phases that application software may perform. An application is distributed to the user; this generally involves the user downloading an installer from the Internet. The user then installs the application, which may involve selecting configuration options or packages to install for additional functionality. The application is then executed by the user to perform a specific task; for example, browsing the Internet, creating documents, playing videos or viewing images. The application may be further configured with settings specified by the user. The application itself may create additional digital artifacts such as creating documents or editing and saving digital images. Finally, the application is removed by uninstalling, a feature designed to remove all application components from the user’s system.

In a real world environment, application software can perform numerous iterations of a specific task (e.g., execution) and the range of potential application scenarios are endless. Furthermore, not all potential phases are covered in Figure 2.3; for example, if an application is somehow corrupted it may require reinstallation or a different version may be installed. However, the phases of the basic life cycle of an application are generally constant.

2.4.2 Reverse Engineering Application Software

“Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made” (Eilam, 2011, p. 3). In the context of computing, reverse engineering
is the process of analysing a subject to create a representation of the system at a higher level of abstraction (Chikofsky & Cross, 1990). Reverse engineering can be seen as a hostile process; for example, when used to illegally replicate software. However, reverse engineering also has a variety of legitimate uses; for example, inspecting malware to determine how it operates. “Reverse engineers attempt to infer the behaviour of malware by comparing the contents of a hard drive before the malware is introduced with the hard drive captured after the malware infection” (Garfinkel, 2012a, p. S5). In the context of the forensic analysis of application software, system-level reverse engineering is required to determine the design, operation and/or behaviour of the application.

Behavioural analysis is the process of reverse engineering the inner workings of applications by examining their effects on the system they operate in. Because source code is not always readily available, behavioural analysis is the major technique for determining what an application does and how it manipulates data (Seifert, Steenson, Welch, Komisarczuk, & Endicott-Popovsky, 2007, p. S23).

Reverse engineering is required when performing forensic analysis of application software due to: 1) Unavailable source code; 2) Source code complexity; and 3) Limited developer documentation. Therefore, reverse engineering is implemented to deconstruct application software and determine how it operates and what digital artifacts are created, modified or deleted in different scenarios. This can include created, modified or deleted file system entries (e.g., data files and directories) and system configuration entries (e.g., Windows Registry keys and values). The information aids in providing reliable knowledge about digital artifacts with regard to application software usage on a system.

Previous research has performed reverse engineering of a large variety of application software to support digital investigations; for example, instant messaging software such as Digsby (Yasin & Abulaish, 2013), cloud storage software such as Dropbox (Quick & Choo, 2013), and anti-forensic tools (Geiger & Cranor, 2006) have all been investigated using reverse engineering techniques. In these examples the following method is conducted: 1) Manual analysis of application software using reverse engineering and a variety of different tools; and 2) Documentation of the analysis method, the discovered findings, and sharing of knowledge (usually via academic publication).

2.4.3 Forensic Analysis of Application Software

The terms (and phases) forensic examination and forensic analysis are intertwined. Forensic examination of application software aims to determine the existence, or absence, of specific software on an investigation target; for example, to determine if a BitTorrent application

In terms of reverse engineering application software, in some scenarios the process may involve taking the program’s binary code and attempting to trace back to assembly language or original source code. In this research, the generic term reverse engineering is used to refer to system-level reverse engineering, where the method attempts to determine system-level changes (files or Registry entries) that the application creates, modifies or removes.
is on a target system. Forensic analysis aims to make sense of the discovered evidence; for example, what was the BitTorrent application used to download. The following discussion uses the generic term **forensic analysis** to refer to both phases.

Although forensic analysis of application software is a fundamental requirement in digital investigations there are few tools designed specifically to perform examination of user-level programs on a target data set. Instead, practitioners implement a variety of conventional analysis techniques and tools to investigate of all types of application software which may range from rudimentary to complex analysis.

Carrier (2003) described the purpose and goals of digital forensic analysis tools using abstraction layers to represent each level of a target digital storage device. An overview of how the four prescribed abstraction layers interact with each other is shown in Figure 2.4 for the scenario of analysing a HyperText Markup Language (HTML) file of a hard disk drive. As depicted, the physical hard drive is comprised of a head and cylinders which is further divided into sectors (or blocks) of data. In the media management layer the sectors store the partition table and are amalgamated to form a partition. Each partition has a file system which stores a boot sector, a file allocation table (FAT) and logically grouped data in files. In the example, the location of the logical grouping of data is stored by the file system. Finally, the application layer is the highest level of abstraction which translates data from the file system into the custom application format. In this example, the HTML could be rendered in a web browser, or source code displayed in a text editor.

![Abstraction levels and layers of an HTML file](Carrier.png)

Forensic analysis can take place on any of the prescribed abstraction layers. Application software analysis can be performed on the sector level using block-based hashing and sub-file forensics to identify chunks of data files associated with an application (Garfinkel, Nelson, White, & Roussev, 2010). On the file system level, application software can be analysed using file system metadata (Kälber, Dewald, & Idler, 2014) or by using file hashing (Roussev, 2009). On the application level, proprietary file formats can be natively analysed using the application which created the file; for example, viewing documents in a word processor.
2.5 Anti-forensics

In response to the procedures developed for digital investigations, criminals and hackers have become aware of investigation methods and, in turn, have developed their own techniques and tools in an attempt to thwart the digital forensics procedure. Such techniques are known as anti-forensics, or counter-forensics, which is defined as “any attempt to compromise the availability or usefulness of evidence to the forensics process” (Harris, 2006, p. S45). This section outlines and discusses anti-forensic techniques and tools that may be maliciously used to hinder digital investigations. Following this, the anti-forensic problem is also outlined and discussed to establish the key research and real world digital investigation challenges.

2.5.1 Anti-forensic Techniques

Anti-forensic techniques can be classified based on the method used to manipulate the original data. Harris (2006) classified anti-forensic techniques into four categories: 1) Hiding; 2) Destruction; 3) Elimination; and 4) Counterfeiting of digital evidence. The following subsections provide a brief discussion of each anti-forensic technique.

2.5.1.1 Evidence Hiding

“Hiding evidence is the act of removing evidence from view so that it is less likely to be incorporated into the forensic process” (Harris, 2006, p. S46). A physical world example would be discarding a firearm into the ocean where it is less likely, or more difficult, for an investigator to discover. In the digital world the primary evidence hiding technique is data encryption. However, steganography can also be implemented to hide digital information as well as more rudimentary techniques including hidden partitions.

Data encryption is the process of disguising a message in such a way as to hide its substance (Schneier, 1996). Evidence hiding using encryption can be achieved on almost all electronic devices or digital communication. Data encryption can range from encrypting individual files to encrypting an entire disk. The integration of strong Full Disk Encryption (FDE) into modern operating systems has created a problem for investigators as encryption cannot normally be circumvented without a key or passphrase (Casey & Stellatos, 2008). Data encryption is also prevalent in network traffic; for example, Transport Layer Security (TLS) provides encrypted network communication, and onion routing provides anonymous encrypted Internet access (Dingledine, Mathewson, & Syverson, 2004). Schneier (1996, p. 16) states “steganography serves to hide secret messages in other messages, such that the secret’s very existence is concealed”. Steganography is commonly described as security through obscurity as the hiding process involves concealing the original information. A routine example is hiding illicit digital images inside another container such as a word processing document.

In addition to the main methods of evidence hiding there is also a variety of other evidence hiding techniques. One example is the Slacker tool which is part of the Metasploit
framework. Slacker can hide files in the slack space, or unallocated area, of a digital storage device with the Microsoft Windows New Technology File System (NTFS) (Kessler, 2007). Another example is by manipulating digital storage devices:

Users may create hidden partitions by altering the partition table to disrupt disk management and prevent applications from seeing that the data area exists. Hidden data can also be found within ADSs on NTFS volumes, in the end-of-file slack space and free space on a medium, and in the Host Protected Area (HPA) on some hard drives, which is a region of a drive intended to be used by vendors only (Kent et al., 2006, p. 42).

Morgan (2008) outlined a variety of techniques that could be implemented to hide data in the Windows Registry; for example, a user could hide data in the registry file header which contains reserved space that is ignored by the operating system and most Registry tools. Even the simple technique of “an obscure key or value placed deep within a little-known configuration tree would likely avoid notice in most cases due to the poorly documented nature of the registry” (Morgan, 2008, p. S40). Evidence hiding presents a daunting problem for digital investigations. As evidence can be removed from plain sight it introduces difficulties in discovering information to provide supporting evidence of specific digital events.

2.5.1.2 Evidence Destruction

Harris (2006) states that evidence destruction involves dismantling or otherwise making digital evidence unusable to the investigative process as data is partially or completely obliterated. An example of evidence destruction in the physical world is wiping fingerprints off a gun or removing gun powder residue from hands, both of which destroys evidence that can link a perpetrator to a firearm. Digital evidence destruction can be either physical or logical based on the technique used to destroy the data.

Physical evidence destruction will most likely result in an unusable device after evidence destruction measures have been conducted as it involves physically destroying the storage device. Usually, this is accomplished by applying brute force; for example, disassembling the device or the use of hammers and drills. However, more sophisticated methods can be implemented. NIST recommends physical destruction as the ultimate form of media sanitization including methods such as disintegration, incineration, pulverization and melting of digital media devices (Kissel, Scholl, Skolochenko, & Li, 2006). A specialised means of hard drive destruction is the use of degaussing which is the process of reducing the magnetization of a storage device to zero by applying a reverse magnetizing force, rendering any previously stored data unreadable and unintelligible (Peron & Legary, 2005).

Logical evidence destruction involves destroying digital data by performing some form of logical data manipulation. Data wiping is the most common logical evidence destruction technique, also referred to as: data overwriting, secure deleting, zeroing, cleaning, sanitising, scrubbing or nuking. Data wiping renders logical data unavailable when compared to normal
file system entry deletion. When a file is deleted the disk cluster link in the directory entry of a file system is set to zero (0) indicating that the clusters have been unallocated. However, the actual contents of the drive are still present (Pal & Memon, 2009). Therefore, in order to avoid data file recovery a malicious user has to actively wipe the contents of the file. There are a variety of standards, algorithms and techniques that can be implemented to perform data wiping. Nevertheless, there has been a great deal of controversy regarding data wiping and the ability to recover data that has been overwritten. According to Joukov, Papaxenopoulos, and Zadok (2006), a single overwrite of data will greatly enhance security and will make any software-based data recovery impossible. Wright, Kleiman, and Sundhar (2008) also state that although there is a good chance of recovering a single bit of data from an overwritten hard drive, using physical based recovery techniques, the possibility of recovering even a small size of data is negligible. A perpetrator can maliciously implement tools to perform data wiping on an entire device, or specific areas of a device, in an attempt to remove the availability of digital evidence. Table 2.6 displays three data wiping techniques which each target a specific area of a digital storage device.

Table 2.6: Classification of data wiping techniques (Source: Table populated from Garfinkel (2007) & Bassett et al. (2006))

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire device wiping</td>
<td>Overwriting the entire contents of a digital storage device with zeros or random data</td>
</tr>
<tr>
<td>Free space wiping</td>
<td>Overwriting unallocated (free) space of a digital storage device so that previously deleted files and file fragments are overwritten</td>
</tr>
<tr>
<td>Digital artifact wiping</td>
<td>Overwriting specific files using automated or manual tools. Common examples include wiping user-generated operating system artifacts (e.g., recently used software list) and Internet artifacts (e.g., web browser cache)</td>
</tr>
</tbody>
</table>

If implemented correctly there is little chance to recover potential evidence after physical device destruction has taken place, as the medium itself has been rendered unusable. Similarly, logical evidence destruction has the potential, if implemented correctly, to provide little chance to recover data. However, improper logical evidence destruction may leave traces of evidence that still remain in the form of tool logs of incorrectly deleted files.

2.5.1.3 Evidence Elimination

According to Harris (2006), eliminating sources of evidence involves neutralising the creation of evidence so that it is never produced. A physical world example is wearing gloves when handling a firearm which means that fingerprints are not left on the object. A simple example of evidence elimination in digital investigations is the perpetrator disabling system log files to remove any future event logging; for example, disabling audit logon events so that no entries are stored when a user logs on to a system. Another example is the use of a live Linux operating system that can be temporarily used as a secondary system. Since the system is live, all changes are stored in volatile memory and lost when the system is shut down.
2.5.1.4 Evidence Counterfeiting

Harris (2006) states that evidence counterfeiting is a method which involves manipulating digital evidence to produce a faked version of the original data, modified to appear different in some aspect. A physical world example is planting another person’s registered firearm with their fingerprints at a crime scene to make it appear the other person was responsible for the crime. A simple example in the digital world is creating legitimate looking e-mails or documents on purpose to mislead an investigator. A more technical example is changing the created timestamp of a file to make it appear that the file was created at a different date and time then it actually was.

2.5.2 Anti-forensic Tools

“Encryption, steganography and erasing tools are increasingly used by malicious individuals to hinder forensic investigation” (Davis et al., 2006, p. 171). Therefore, the problems caused by anti-forensic techniques and the removal of digital evidence is further accentuated by the high availability of anti-forensic tools capable of achieving the task. These tools usually require minimal technical knowledge to operate but can cause significant loss of digital evidence; for example, privacy suite tools “claim to expunge all traces of information about specific computer usage, including documents and other files created, records of websites visited, images viewed and files downloaded” (Geiger & Cranor, 2006, p. 16). However, while anti-forensic tools are relatively easy to use, the underlying principle of how the tool operates is usually too advanced for an average user to understand.

What also must be understood is that anti-forensic tools are not always developed for unlawful practices and many have legitimate uses; for example, encryption is a primary technique to perform data hiding, but encryption tools also have legitimate uses such as protecting confidential data in government and medical sectors. To avoid confusion, tools that are for specific anti-forensic use must be classified based on the malicious intent to manipulate the availability of digital evidence, rather than the technical functionality of the tool. Table 2.7 displays three groups of anti-forensic tools that may be used for illicit purposes with details of the tool functionality for each classification.

<table>
<thead>
<tr>
<th>Tool Classification</th>
<th>Tool Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Encryption</td>
<td>Perform data encryption on an entire disk, partition or file using a variety of encryption algorithms depending on the implementation and application</td>
</tr>
<tr>
<td>Data Wiping</td>
<td>Overwrite data on digital data storage devices which may include overwriting individual files, partitions, whole disks or free space</td>
</tr>
<tr>
<td>Privacy Suites</td>
<td>Automate the removal (either deletion or overwriting) of digital artifacts associated with data created by various system or application activity</td>
</tr>
</tbody>
</table>
Anti-forensic tools can be categorised into two main sources, including: 1) Tools that are built-in to an operating system; and 2) Tools that are purpose built third-party applications. Although many anti-forensic tools are stand-alone applications, there are also a collection of built-in operating system utilities to perform anti-forensic techniques; for example, in terms of logical evidence destruction, most modern operating systems have built-in data wiping utilities: The \texttt{shred} utility is available on most Linux-based operating systems, the \texttt{cipher.exe} utility is available on most Microsoft Windows systems, and the \texttt{rm} utility (with the \texttt{-P} argument) can overwrite data files on most Apple OS X systems. In addition, a plethora of third party tools are available to perform the previously specified anti-forensic techniques. The following subsections provide a brief summary of the three classifications of anti-forensic tools with usage examples and a discussion on the effect they have on evidence.

### 2.5.2.1 Data Encryption Tools

Data encryption tools are able to encrypt data in a range of scenarios, from encrypting a single file to encrypting an entire hard disk. To heighten the problem, data encryption tools are readily available. “Advances in communications technologies, such as the Internet, have made complex encryption products widely accessible, presenting the forensic computer examiner with a significant barrier” (McKemmish 1999, p. 6).

Programs such as PGP and TrueCrypt enable file-level encryption, as well as encrypted containers that may be mounted as a volume and used to store data. Some encryption systems make an effort to support plausible deniability, making it difficult to determine whether a disk contains encrypted versus random data (Casey & Stellatos 2008, p. 93).

Furthermore, modern operating systems are now shipped with built-in disk encryption capability; for example, Microsoft Windows BitLocker Drive Encryption or Apple FileVault. Casey, Fellows, Geiger, and Stellatos (2011) also state that many applications exist for disk or volume encryption including open source (TrueCrypt) or third party (McAfee’s Safeboot, WinMagic’s SecureDoc, Symantec’s PGP and GuardianEdge) options.

### 2.5.2.2 Data Wiping Tools

According to Berghel and Hoelzer (2006), the advancement of modern computer forensics has prompted disk wiping to become increasingly important to protect proprietary, confidential and private information, ultimately resulting in a plethora of readily available disk and data wiping utilities. However, the potential malicious usage of data wiping tools can be devastating to forensic investigations.

Many existing programs claim to properly sanitize a hard drive, including $1,695 commercial offerings that boast government certifications, more than 50 tools licensed for a single computer system, and free software/open-source products that seem to offer largely the same features (Garfinkel & Shelat 2003, p. 22).
Data wiping tools can be broadly classified based on functionality into three categories: 1) Disk wiping tools; and 2) Free space wiping tools; 3) File wiping tools. However, some tools may include support for two or even all three functions. As the name implies, disk wiping overwrites all logical data on a digital storage device, or disk. Data wiping tools that overwrite free space sanitise unallocated, or slack, space on a digital storage device so that previously deleted (unlinked) files and file fragments are securely removed (Garfinkel & Malan, 2006). Finally, file wiping tools perform secure removal of data files by overwriting the data contained in the file.

2.5.2.3 Privacy Tools

Privacy tools operate on the same premise as file wiping tools, by securely deleting files by overwriting the original file. However, “unlike tools designed to wipe a whole disk or filesystem, counter-forensic privacy tools are designed to locate activity records scattered across the computer filesystem and erase them irretrievably, while leaving the system otherwise fully functional” (Geiger & Cranor, 2006, p. 16). Basically, privacy tools automate the discovery and secure wiping of data files that may be unwanted. Common examples include temporary files, Internet history, previously deleted files (e.g., Recycling Bin), search history and a list of recently used applications or files. There are no built-in privacy tools which are bundled with operating systems, rather privacy tools are stand-alone third-party applications.

2.5.3 The Anti-forensics Problem

It is important to develop a better understanding of the problems surrounding the use of anti-forensic techniques and tools because of the impact they have on digital investigations. It has already been discussed how anti-forensic techniques and associated tools have the potential to adversely affect the availability of digital evidence. However, the anti-forensics problem is exacerbated by a variety of other factors including the prevalence, increased sophistication and availability of anti-forensic tools encountered in real world digital investigations.

2.5.3.1 Anti-forensic Prevalence

According to Sartin (2006), anti-forensics is a trend that is on the rise in both the law enforcement, government and private sectors. In terms of the prevalence of anti-forensics, there is some difference in opinion on the occurrence of anti-forensics techniques encountered in real world digital investigations. However, Verizon publish an annual Data Breach Investigations Report since 2007 which presents detailed information and statistics compiled from the cases handled by the digital forensic investigation team of the company. “The fact of the matter is that for the entire period that we have been studying breaches, we have seen consistent signs of anti-forensics” (Verizon, 2011, p. 61). Additionally, although anti-forensic techniques are hard to identify if effective, Verizon (2012, p. 55) states that they often identify anti-forensic techniques in the field, with an estimation that 1/3 of all cases incorporate some form of anti-forensic element. Additionally, Verizon (2009, p. 40) states that in cases where anti-forensics
was encountered, data wiping was discovered in 31% and data hiding techniques accounted for 9% of all cases reported.

2.5.3.2 Anti-forensic Tool Usage

The techniques and tools used to remove the availability of evidence are becoming increasingly sophisticated and, thus, presenting a growing challenge for investigators. “Computer forensics presumed usefulness against anyone with computer savvy is minimal because such persons can readily defeat forensics techniques” (Caloyannides 2009, p. 18). The ability to compromise potentially essential digital evidence presents a daunting problem for the digital investigation process.

Unfortunately, with people becoming more mindful of information security, the further introduction of security within operating systems (e.g. full disk encryption within Microsoft and Apple Mac), the promotion of anti-forensic technologies and wiping of media, the ability for investigators to locate relevant evidence will become increasingly challenging (Fahdi et al. 2013, p. 7-8).

A further challenge is the availability of anti-forensic tools. “The number of scholarly papers on protecting against anti-forensic methods is greatly outnumbered by the number of websites about how to exploit the forensic process” (Harris 2006, p. S48). A quick Google search emphasises the availability of data wiping tools, producing approximately 12 million links using the term data wiping tool as a keyword. A brief investigation provides a perpetrator with numerous potential data wiping tools for any operating system platform. On the first page of the performed Google search, a Wikipedia page was found named List of data-erasing software with a total of 16 different data wiping tools (Wikipedia 2016). Additionally, another list entitled 42 Free Data Destruction Software Programs was also found on the first page of links (Fisher 2016). The same problems are also prevalent for encryption programs and other similar anti-forensic tools. This readily available information illustrates that anti-forensic tools and associated information on how to use them is widely distributed and available to anyone with an Internet connection.

2.6 Conclusion

Chapter 2 has presented the pertinent background information required to understand the content covered during the course of this research project. An overview of digital forensics was presented which included various phases of the investigation procedure. Digital evidence was defined and outlined including the use of digital artifacts to aid forensic analysis, evidence sources of digital artifacts used in this research, as well as a summary of commonly used forensic data abstractions. Various digital forensic research challenges were identified. A review of anti-forensics provided a summary of techniques and associated tools used to remove the availability of digital evidence, followed by the real world challenges faced by anti-forensic
presence in digital investigations. This background information has led to knowledge of the subject area and direction of this research project.

Chapter 3 continues on from the background information presented in this chapter with the purpose of reviewing further literature to aid in identification of gaps in existing research. Limitations of the technology, techniques and tools used to perform forensic analysis of application software, specifically anti-forensic tools has, thus far, been clearly highlighted as a viable research topic.
Chapter 2 presented an overview of relevant background material for this research project in which a summary of digital forensics, digital evidence and digital forensic research challenges were summarised. An introduction to application software was presented coupled with typical forensic analysis techniques used to investigate applications. Anti-forensics was then outlined with a summary of the techniques and tools which can be maliciously used to remove the availability of digital evidence creating problems that challenge both research and real world investigations.

This chapter presents an in-depth review of literature that is pertinent to the selected research topic; specifically, literature that focusses on either forensic analysis of anti-forensic tools or automated forensic analysis. Firstly, a study of research papers surrounding the forensic analysis of anti-forensic tools is presented. This aims to summarise previous research in the detection of anti-forensic tools as well as to provide an overview of digital artifacts associated with anti-forensic tool usage. A detailed evaluation of reverse engineering techniques and tools is then undertaken; the goal being to identify current reverse engineering practices to aid identification of digital artifacts from application software, and more specifically, anti-forensic tools. Next, various approaches to performing automated digital artifact detection are critiqued to determine what techniques are available to identify digital artifacts from application software on an investigation target with a summary of the problems and issues discovered.
3.1 Forensic Analysis of Anti-forensic Tools

Anti-forensic tools were previously introduced (see Section 2.5) and shown to have the potential to remove the availability of digital evidence. As such, academic research has analysed the techniques these tools implement to determine how they operate and how they can be defeated. Specifically, attempts to determine digital artifact remnants left after anti-forensic tool usage has been a key focus. Research has also investigated techniques to detect anti-forensic tool usage on an investigation target. In order to provide a better understanding of preceding research, a body of similar case studies will be reviewed in the following subsections. Where available, information will be discussed regarding what types of digital artifacts are potentially available and what methods and tools were used by researchers to reverse engineer the tested anti-forensic tools.

3.1.1 Evaluation of Counter-Forensic Privacy Tools

Geiger (2005), and Geiger and Cranor (2006), both reviewed the performance of six commercial anti-forensic privacy tools in terms of the ability to purge a variety of operating system activity including Internet history, recently used files, search history, temporary files, previously deleted files and unallocated space. The research papers outlined that the experimental testing platform was Windows XP Professional with the New Technology File System (NTFS). Additional software was installed, including Microsoft Office. Eight days of typical user activity was performed and documented using specific marker phrases to later identify content. This process was repeated for each of the six anti-forensic tools tested on a separate testing system. Each system was subjected to a forensic bit-by-bit copy and analysis performed using a common forensic analysis tool, namely Forensic ToolKit produced by Access Data. Additionally, a baseline system was created prior to anti-forensic tool usage.

Geiger (2005) documented the successes, errors and limitations for the six selected anti-forensic tools. Notable tool errors included incomplete wiping of unallocated space, failure to wipe targeted user and system files, failure to remove Registry records (e.g., a list of recently used files), failure to wipe data retained in the actual file system table and failure to remove system restore points. In addition, each anti-forensic tool left a variety of tool operation information including a record of tool configuration and activity (e.g., files selected for deletion) and distinct file signatures (e.g., wiped files were renamed with a unique suffix). This information is exceptionally useful for a practitioner to discover; that is, what files were targeted by the user and what files were securely removed by an anti-forensic tool.

Selectively purging sensitive data on a filesystem - as opposed to a blanket wipe of the filesystem - is a challenging task. All of the commercial counter-forensic tools tested left data of potential value to an investigation of activity on the computer system (Geiger 2005, p. 11).

1The typical user activity included browsing arbitrary websites, registering new accounts for websites, posting comments to online forums, saving HTML pages, performing instant messenger chatting and retrieving and composing emails.
Nevertheless, each anti-forensic tool did succeed in wiping the majority of targeted data, having the potential to destroy evidence and hinder the forensic analysis process. Geiger (2005) proposed future research to carry out additional analysis of anti-forensic tools to document operation weaknesses and to expand a catalog of file signatures that are created by anti-forensic tools when wiping data files. Full details of discovered weaknesses and discovered file signatures are available from Geiger (2005), and Geiger and Cranor (2006).

### 3.1.2 Assessing Trace Evidence Left by Secure Deletion Programs

Burke and Craiger (2006) published a paper entitled: Assessing Trace Evidence Left by Secure Deletion Programs, which investigated trace evidence from a collection of Windows based anti-forensic tools to perform secure deletion (data wiping). The overall findings indicate that the majority of secure deletion programs left identifiable signatures and file system metadata when deleting files. In contrast to the previous research study by Geiger (2005), Burke and Craiger (2006) investigated data wiping tools in the context of securely deleting data files rather than evaluating privacy tools which automate data wiping of specific digital artifacts.

Burke and Craiger (2006) implemented a test environment using a sanitised floppy disk and copied the target data file to the disk. Each of the five secure deletion programs were then used to wipe the file on a separate testing system. “As expected, all the programs completely deleted the file contents. However, several programs did leave digital signatures within the file’s root directory entry” (Burke & Craiger 2006 p. 191). Table 3.1 displays a summary of the identifiable file signatures and file system metadata information that remained after performing secure deletion of a file. All file signatures are displayed using the short (MS-DOS) file name convention.

Table 3.1: File signatures from secure deletion programs (Source: Table derived from Burke and Craiger (2006, p. 191-195))

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Identifiable File Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evidence Eliminator 5.0</td>
<td>Did not rename deleted file, did not remove file system metadata and also created temporary file, named “?E--- 1.TMP”</td>
<td></td>
</tr>
<tr>
<td>UltraSentry 2.0</td>
<td>Attempted to rename deleted file name to “?PPPPP.FPP”, where new file name is the same length as the original, also no file system metadata associated with the deleted file was removed</td>
<td></td>
</tr>
<tr>
<td>CyberScrub 4.0</td>
<td>File name and metadata were overwritten with random characters and values, deleted file was set to the “hidden” attribute and directory entries were renamed with the “.WIP” extension</td>
<td></td>
</tr>
<tr>
<td>East-Tec Eraser 2005</td>
<td>File name and metadata were overwritten with random characters and values, deleted file was set to the “hidden” attribute and directory entries were renamed with the “.WIP” extension</td>
<td></td>
</tr>
<tr>
<td>Eraser 5.3</td>
<td>File name renamed with random characters, file creation and modification dates set to the FAT file system epoch (January 1, 1980)</td>
<td></td>
</tr>
</tbody>
</table>
"Tests of five popular Windows-based programs demonstrate that each program leaves unique signatures, which could assist examiners in determining whether a secure deletion was performed and in identifying the program used to perform the deletion" (Burke & Craiger, 2006, p. 194-195). Interestingly, the findings determined that various file system metadata information, including creation and modification time stamps, was left unaltered by most programs which could assist in determining the date and time that secure deletion took place.

Future research suggested extending the research to other file systems such as FAT32, NTFS and EXT2 and other operating systems including Linux, UNIX and Apple OS X. Additionally, Burke and Craiger (2006) also noted that all of the tested programs had numerous options which could be optimised to extend the effectiveness of the secure deletion process.

### 3.1.3 An Evaluation of Data Erasing Tools

Martin and Jones (2011) performed similar research to Burke and Craiger (2006) but carried out a more extensive test using twelve different data wiping tools and a much larger data set, comprised of hundreds of thousand data files. Overall, the number of anti-forensic tool failures were observed to be high; in particular it was detected that a variety of residual data in the System Volume Information folder present on NTFS file systems.

The testing method first involved creating a baseline data set comprised of a collection of more than 140,000 data files of different file types (e.g., archives, images, documents and videos). The testing environment used a sanitised physical 80 GB hard drive to ensure no residual data remained, copying the file data set to the hard drive, wiping all files using each data wiping tool, and finally creating a forensic image of the drive. A range of common forensic analysis tools were used, two of which were Forensic Toolkit, md5deep to perform file hashing and scalpel to perform file carving. Table 3.2 displays an analysis of the results including the tool name, version, number of files recovered by Forensic Toolkit (FTK), the number of files recovered by scalpel and the data wiping pattern.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>FTK Recovered</th>
<th>Scalpel Recovered</th>
<th>Overwrite Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCWipe</td>
<td>4.01.23</td>
<td>28,326</td>
<td>170</td>
<td>Zeroes</td>
</tr>
<tr>
<td>DPWiper</td>
<td>1.1</td>
<td>91</td>
<td>126</td>
<td>Random</td>
</tr>
<tr>
<td>Eraser</td>
<td>6.0.8.2273</td>
<td>3,255</td>
<td>40</td>
<td>Random</td>
</tr>
<tr>
<td>File Eraser</td>
<td>5.7</td>
<td>3,075</td>
<td>6,333</td>
<td>Random</td>
</tr>
<tr>
<td>File Shredder</td>
<td>2.0</td>
<td>145,695</td>
<td>59</td>
<td>Random</td>
</tr>
<tr>
<td>Freeraser</td>
<td>1.0.0.23</td>
<td>145,687</td>
<td>12</td>
<td>Random</td>
</tr>
<tr>
<td>Hard Drive Eraser</td>
<td>2.0</td>
<td>58</td>
<td>34</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

The overall results discovered that seven of the twelve data wiping tools left recoverable information. Forensic Toolkit was able to recover a large number of data files, while the scalpel file carver was able to extract fragments of files, as well as some entire files.
3.1.4 Identifying Trace Evidence from Target-Specific Data Wiping

Carlton and Kessler (2012) performed an investigation of five privacy tools that securely wipe targeted files and evidence of selected system activity including Internet history and Registry entries. The goal of the research was to identify trace evidence which data wiping programs left after usage. The research conducted is similar to that previously covered but evaluation was performed on more recent software including the Microsoft Windows 7 operating system and data wiping tools that support Windows 7 and have a Graphical User Interface (GUI).

Carlton and Kessler (2012) prepared sample data on fresh Windows 7 installation and inserted a collection of sample data on the system. A total of 57 sample data files were added to the system, made up of common file formats including documents and images. Files were added using a variety of scenarios such as downloading files from the Internet, creating documents using a word processor and taking and saving screenshots. This resulted in a variety of generated data in various evidence sources. Each anti-forensic tool was executed on an identical system and configured to wipe specific files and to automate discovery and wiping of Internet history and Registry entries.

The findings stated that all five data wiping tools left some form of trace evidence that may be of value to forensic analysts (Carlton & Kessler, 2012). Specifically, the findings showed that trace digital artifacts remained in web browser history and in the Windows Registry while some data wiping tools created log files which retained information regarding tool usage.

3.1.5 Detecting Data Concealment Programs

Davis et al. (2006) published a paper entitled: Detecting Data Concealment Programs Using Passive File System Analysis, outlining a method to detect trace evidence left by data wiping tools. The previously reviewed studies in this section analysed the failures of data wiping tools which left traces of evidence. This research differs in that the premise is: “having established the presence of a specific data concealment program on a seized computer, an investigator can attempt to exploit known vulnerabilities in the program to recover concealed or erased data” (Davis et al., 2006, p. 181–182). The vulnerabilities discussed are anti-forensic tool errors; for example, the tool has not been successful in completely removing the data (as described in the previous studies).

Davis et al. (2006) first described the application software life cycle (see Section 2.4.1), stating that an application has various phases of operation including installation, execution and uninstallation. Providing an investigator with information of where the application is found in the life cycle is valuable evidence to determine the actions of the perpetrator. The application life cycle was recreated for a selection of data wiping tools and an application fingerprint generated which contained metadata about data files. Figure 3.1 displays an overview of the implemented system to perform fingerprint generation of a specific program. It is important to note that Davis et al. (2006) constructed a single fingerprint for each program and did not attempt to identify the difference between installed and executed digital artifacts.
Figure 3.1: Fingerprint generation system to detect data concealment programs (Source: Figure adapted from Davis et al. (2006, p. 177-178))

After identification of the digital artifacts associated with a specific program, the conducted research developed and tested a tool, dubbed Seraph, which aimed to assist forensic investigators by automating the detection of digital artifacts associated with various data wiping applications. According to Davis et al. (2006), digital artifact matching was performed by correlation of the file name, full file system path, or MD5 file hash. In terms of matching effectiveness, file name matching produced a significant number of false positives due to a common file name; for example, most applications include a readme.txt file. When using the full path the false positive rate was greatly reduced, especially when applications were installed in default directories. Additionally, using file hashing was effective, but the results illustrated a drop in effectiveness as different versions of the same software produced different file hash values. Table 3.3 displays an overview of the results from the experimental testing of the Seraph tool including the data wiping tool tested and the corresponding software state, the number of matched entries compared to the entries in the fingerprint profile, as well as the related confidence rating.

<table>
<thead>
<tr>
<th>Program</th>
<th>State</th>
<th>Matched Entries</th>
<th>Fingerprint Entries</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Cleanup</td>
<td>Installed</td>
<td>39</td>
<td>42</td>
<td>92.86%</td>
</tr>
<tr>
<td>Complete Cleanup</td>
<td>Run</td>
<td>43</td>
<td>46</td>
<td>93.48%</td>
</tr>
<tr>
<td>Eraser</td>
<td>Installed</td>
<td>41</td>
<td>48</td>
<td>85.42%</td>
</tr>
<tr>
<td>Eraser</td>
<td>Run</td>
<td>40</td>
<td>49</td>
<td>81.63%</td>
</tr>
<tr>
<td>TrackEraser</td>
<td>Installed</td>
<td>53</td>
<td>55</td>
<td>96.36%</td>
</tr>
<tr>
<td>TrackEraser</td>
<td>Run</td>
<td>53</td>
<td>56</td>
<td>94.64%</td>
</tr>
</tbody>
</table>

Overall, the research conducted by Davis et al. (2006) was a new technique to generate reference sets for data wiping tools. However, a number of limitations are present, including generating only digital artifacts for the install and run phases of the application life cycle. The uninstallation phase would also be an important phase to test as perpetrators might uninstall a tool to potentially remove any traces of activity.
3.1.6 Anti-forensic Research Challenges

The previous subsections introduced and discussed research papers that focussed on advancing research and knowledge on the topic of anti-forensic techniques and tools. The reviewed academic research papers can be categorised into the following three primary research areas:

1) Evaluating the performance of anti-forensic tool functionality; for example, the ability of a data wiping tool to remove all evidence (Geiger, 2005; Geiger & Cranor, 2006; Burke & Craiger, 2006; Martin & Jones, 2011)

2) Determining residual remnants from anti-forensic tool operation; for example, log files which retain tool operation information (Geiger, 2005; Martin & Jones, 2011; Carlton & Kessler, 2012)

3) Development and evaluation of systems to aid detection of anti-forensic tools; for example, automating analysis of data files associated with a known anti-forensic tool (Davis et al., 2006)

While all the mentioned research papers have presented useful and interesting findings for the digital forensic community, serious research challenges remain. The problem is that, although much research has been conducted, the current methods ultimately result in researchers having to test every anti-forensic tool (and every tool version) on every operating system in every scenario. Furthermore, the research output distributes knowledge and practitioners must read every paper to determine the effect of a specific anti-forensic tool which they may encounter in an investigation. If documentation is not available, investigators must themselves conduct time consuming research to establish such knowledge. Overall, this method results in research output that produces no technological advancement to the fundamental methods, techniques and tools with which digital investigation of anti-forensic tools are performed.

Research requires that the tested anti-forensic tools be reverse engineered to determine the digital artifacts created, modified or removed. Furthermore, each project investigates anti-forensic tools in an endless variety of scenarios; for example, wiping a file, wiping Internet related files, deleting Registry keys or wiping an entire disk. Each of these scenarios is also affected by the experimental testing environment. Basically, no official set of procedures exist for analysing each tool. Instead of each researcher setting an arbitrary number of experiments, a method is needed to specify important scenarios to test. The overall outcome is that no tangible advancement of the underlying technology is made from the research topics outlined in the previous subsections. Research is required in the area of anti-forensic tool detection to narrow the scope of the investigation, allowing the practitioner to manually analyse a specific tool to find trace evidence and establish digital events.

Forensic tools will have to become less focused upon identifying evidence that is criminal in its own right and become more intelligent in identifying artifacts that highlight misuse. For example, analyses will need to be developed that are capable of picking up trace evidence to the application of anti-forensic techniques. Whilst not evidence of the crime itself, it will provide invaluable intelligence to the examiner to how to best proceed (Fahdi et al., 2013, p. 7).
In summary, previous research in the realm of anti-forensics has provided a strong definition of scope; knowledge regarding effectiveness of anti-forensic tools and knowledge regarding remnant digital artifacts and unique file signatures. The following section investigates the use of reverse engineering to aid in deconstructing anti-forensic tools and application software in a general sense. The outcome of the process would be discovery of digital artifacts associated with an anti-forensic tool which may later be used to aid detection on an investigation target.

3.2 Reverse Engineering of Application Software

The use of system-level reverse engineering to aid digital forensics has already been briefly introduced and defined (see Section 2.4.2). To reiterate, reverse engineering is the process of analysing a subject to create a representation of that subject at a higher level. For clarity, this research uses the generic term reverse engineering to refer to system-level reverse engineering. In digital forensics reverse engineering application software is commonly used to determine the digital artifacts (e.g., data files or Registry entries) created on a system; for example, malware is reverse engineered to determine how it operates.

In terms of application software, reverse engineering is usually accomplished by performing system-level reverse engineering. “System-level reversing requires a variety of tools that sniff, monitor, explore and otherwise expose the program to being reversed” (Eilam 2011, p. 15). Such tools have the functionality to obtain system-level information about applications and track program input and output; for example, when an application is installed, tools can track what file system and Windows Registry additions, modifications or deletions have occurred. Two primary techniques are commonly implemented for system-level reverse engineering of application software: 1) System monitoring; and 2) Differential analysis. Each technique has certain advantages and disadvantages but both have the functionality to monitor and determine the system-level changes that an application makes during the application life cycle.

3.2.1 System Monitoring

System monitoring has a broad scope depending on the context in which it is implemented; it can range from monitoring the resources (CPU usage, running processes, RAM utilization) of a single desktop to monitoring an entire network of computers using a distributed system. However, system monitoring is not solely for performance measurements, rather, it also includes monitoring system activity and tracking system changes. In digital forensics, system monitoring encompasses the latter scope, where tools are commonly used to track system activity including events such as file system and Windows Registry changes.

System monitoring tools range in scope, complexity and functionality; for example, the Microsoft Windows built-in tool, Task Manager, is a type of system monitoring tool which displays the programs, processes and services that are running on a system. However, Task Manager provides no functionality to determine file system or Windows Registry changes, or the ability to record such events, which are both necessary functions for reverse engineering.
of application software. This section will therefore outline system monitoring tools used in
digital investigations to track and record operating system events with a specific emphasis on
detecting file system and Windows Registry changes. Table 3.4 displays a selection of system
monitoring tools commonly used in digital forensics to reverse engineer application software.

Table 3.4: An overview of system monitoring tools to track file system and Windows Registry changes. Operating system (OS) support is checked if the tool natively supports Windows XP, Vista, 7 and 8. File system and Registry columns indicate support for each data source. Logging indicates the functionality to output discovered system changes to the following log file formats: TXT = Text based log file, PML = Native Process Monitor Format, CSV = Comma-Separated Values, XML = Extensible Markup Language

<table>
<thead>
<tr>
<th>Tool</th>
<th>License</th>
<th>OS</th>
<th>File System</th>
<th>Registry</th>
<th>Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaptureBAT</td>
<td>Open</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ TXT</td>
</tr>
<tr>
<td>FileMon</td>
<td>Freeware</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ PML, CSV, XML</td>
</tr>
<tr>
<td>RegMon</td>
<td>Freeware</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓ PML, CSV, XML</td>
</tr>
<tr>
<td>Process Monitor</td>
<td>Freeware</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ PML, CSV, XML</td>
</tr>
</tbody>
</table>

Seifert et al. (2007) developed a system monitoring tool specifically targeted for digital forensics called CaptureBAT. The tool provides the functionality to monitor Microsoft Windows operating system state changes at a low kernel level and has the ability to capture both file system and Windows Registry changes, as well as monitoring process information. All events that CaptureBAT detects are logged to standard output in a delimited text format including the timestamp (date and time), the source (file system, Registry or process), the event (e.g., write, create), the process that caused the event and the full path of the file that was acted upon. CaptureBAT includes portable exclusion lists that are designed to omit detected events from being logged; this being included to filter normal system events such as system services that are constantly running and creating system noise. Seifert et al. (2007) demonstrated CaptureBAT capability by analysing and collecting evidence of the payload from a malicious Microsoft Word document being invoked on a test system. A key point of CaptureBAT is it was designed for digital forensic requirements.

According to Russinovich and Margosis (2011), FileMon captures information about file system activity, while RegMon captures information about Registry activity. However, according to Microsoft (2016a) and Microsoft (2016c) both tools are out-dated with the last release being in 2006. Process Monitor was first released in 2006 to replace the obsolete FileMon and RegMon tools. Process Monitor provides a unified view of file system, Registry and process activity while offering non-destructive filtering and a detailed log file format (Russinovich & Margosis, 2011). The functionality to log changes is essential to perform forensic analysis of application software. Process Monitor includes support to log system changes to the Process Monitor Format (PML) and export options to Comma-Separated Value (CSV) and Extensible Markup Language (XML).

2Available from: [https://www.honeynet.org/node/315](https://www.honeynet.org/node/315)
Many forensic and digital investigators consider the tool [Process Monitor] the de facto advanced monitoring tool for Windows systems due to its advanced features, including live filtering and the ability to save session details in Process Monitor Format (PML), which further allows the data to be loaded into Process Monitor for subsequent analysis (Halsey & Bettany, 2015, p. 25).

Tools such as Process Monitor and CaptureBAT determine file system and Registry changes using common operating system Application Programming Interfaces (API). For example, the CaptureBAT tool monitors file system changes using kernel drivers and callback functions available in the Windows API (Seifert et al., 2007). Examples provided are ReadDirectoryChangesW() and FindFirstChangeNotification() functions which have the capability to monitor a directory (or entire volume) for changes (Microsoft, 2017a). Another example is the FileSystemWatcher class available in the .NET framework which also monitors a directory for file system changes (Microsoft, 2017b). The stated APIs have the capability to be easily implemented to track both file system and Registry changes in Windows environments. In addition, similar solutions exist for Linux and OS X operating systems.

In summary, system monitoring tools provide the functionality for forensic analysts to reverse engineer application software by detecting and reporting system-level changes. All of the reviewed system monitoring tools operate in real time on a running operating system. However, while all tools collect detailed information on running processes, this particular functionality is not required for this research project. Rather, the goal is to determine the digital artifact changes that application software creates. Another technique that can be implemented to reverse engineer application software, however, is differential analysis.

### 3.2.2 Differential Analysis

Garfinkel, Nelson, and Young (2012) define differential analysis as an analytical process that compares two objects and reports the differences between them. Differential analysis is a type of delta algorithm which can be defined as a function that computes the difference between two files or strings (Hunt, Vo, & Tichy, 1996). Delta algorithms are commonly used for version management systems that track revisions of documents or computer programs. A simple example is the `diff` utility that is included in most Linux operating systems. Differential analysis, specifically differential forensic analysis, operates on the same principles as delta algorithms but varies depending on the nature of input. While delta algorithms commonly take text files as input, differential forensic analysis performs differencing on objects such as forensic disk images, Registry hive files, or network traffic captures.

The differential forensic analysis formula developed by Garfinkel et al. (2012) can be expressed as: \[ A \xrightarrow{R} B. \] “If \( A \) and \( B \) are disk images and the examiner is evaluating the installation footprint of a new application, then \( R \) might be a list of files and Registry entries that are created or changed” (Garfinkel et al., 2012, p. S51). The objects that are compared using differential analysis can be any two common objects; for example, two forensic
disk images of hard drives, two data files, or two network traffic capture files. The result from
differential analysis is the reporting of changes between the two objects; for example, in the
case of two forensic disk images the differences are the additions, modifications, or deletions
to the file system and Windows Registry that occur. Table 3.5 displays a group of various
differential analysis tools supported on the Microsoft Windows operating system.

Table 3.5: An overview of differential analysis tools to determine file system and Windows Registry changes made by an application. Data source indicates the input data requirements. File system and Registry columns indicate support for each data source. Logging indicates the functionality to output discovered system changes to the following log file formats: TXT = Text based log file, PML = Native Process Monitor Format, CSV = Comma-Separated Values, XML = Extensible Markup Language, DFXML = Digital Forensic XML, RegXML = Registry XML

<table>
<thead>
<tr>
<th>Tool</th>
<th>License</th>
<th>Data Source</th>
<th>File System</th>
<th>Registry</th>
<th>Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regshot</td>
<td>Open</td>
<td>Live Differencing</td>
<td>✓</td>
<td>✓</td>
<td>✓ TXT</td>
</tr>
<tr>
<td>WhatsChanged</td>
<td>Freeware</td>
<td>Live Differencing</td>
<td>✓</td>
<td>✓</td>
<td>✓ TXT</td>
</tr>
<tr>
<td>idifference.py</td>
<td>Open</td>
<td>Disk Image</td>
<td>✓</td>
<td>✗</td>
<td>✓ TXT, DFXML</td>
</tr>
<tr>
<td>idifference2.py</td>
<td>Open</td>
<td>Disk Image</td>
<td>✓</td>
<td>✗</td>
<td>✓ TXT, DFXML</td>
</tr>
<tr>
<td>rdifference.py</td>
<td>Open</td>
<td>Registry hive</td>
<td>✗</td>
<td>✓</td>
<td>✓ RegXML</td>
</tr>
</tbody>
</table>

The tools are categorised based on the input data source, either: 1) Snapshot tools which are run on a live operating system and use a live file system and Registry hives to perform differential analysis; and 2) Disk image and Registry hive tools which use post-mortem data sources such as a forensic disk image of a hard drive and offline Registry hive files to perform differential analysis.

3.2.2.1 Snapshot Tools

Snapshot tools, also referred to as live differencing tools or simply diff tools, are similar to system monitoring techniques but function using a different method to determine system-level changes. Both techniques are executed on a live running operating system. System monitoring tools operate by constantly monitoring the system and reporting changes, while snapshot tools collect a snapshot before and after a specific scenario, compare them and report the results; for example, a snapshot of file system entries is collected before and after an application installation, compared and the differences between them reported.

The Regshot tool is a differencing utility available for Microsoft Windows. Regshot provides the functionality to take a system snapshot of the file system and Windows Registry on a live system and compare them to determine the changes that have occurred (Carvey, 2011). Regshot has been used by in a number of digital forensic research studies to reverse engineer applications and determine the digital artifacts created or left after performing different actions. Studies where Regshot was used for data collection include the forensic analysis of digital artifacts left by virtual disk encryption tools (Lim, Park, s. Lim, Lee, & Lee, 2010), investigation of Registry artifacts produced by BitTorrent applications (Lallie & Briggs, 2011), review of cloud storage client applications (Malik, Shashidhar, & Chen, 2015).
and an analysis of timestamps when inserting and removing USB devices (Deb & Chetry, 2015). Furthermore, as stated by Kang and Srivastava (2011), Regshot is commonly used to perform dynamic malware analysis; a technique that monitors malware on a system.

There are a variety of similar tools, such as WhatsChanged, but none are prominent in digital forensics practice or research. According to He, Duan, Luo, Wang, and Zhang (2011), WhatsChanged is a system snapshot tool that can scan the file system and Registry, save the snapshot and compare multiple snapshots. A variety of other similar solutions exist including InCtrl4, TrackWinstall5 and SysTracer6. It is important to note that none of these tools were designed specifically for digital forensics in spite of having been adopted by forensic researchers in the absence of a better solution.

3.2.2.2 Disk Image and Registry Hive Tools

In contrast to snapshot tools, there are a selection of differential analysis tools that have been specifically designed for digital forensic research. These tools were developed by Garfinkel et al. (2012) when outlining a general strategy to perform differential forensic analysis and are built on specifically authored high-level forensic data abstractions (e.g., Digital Forensic XML (DFXML)) developed by the same researchers (see Section 2.2.3.2).

Garfinkel et al. (2012) authored tools to perform differential analysis of: 1) Forensic disk image contents, specifically, file system differences; and 2) Windows Registry hive files. Both tools leverage the DFXML data abstraction, associated DFXML Application Programming Interface (API) and are written in the Python programming language. The idifference.py tool takes two forensic disk images as input (or two DFXML reports generated by fiwalk) and reports the differences between them. The rdifference.py tool takes two offline Registry hive files as input (or two RegXML reports generated by the regxml_extractor tool) and reports the differences between them. Both tools are relatively new, being released in 2012, and designed specifically for digital forensics research.

Nelson, Steggall, and Long (2014) performed additional research and development of DFXML differencing capabilities resulting in a completely rewritten file system differencing tool, idifference2.py10. They also wrote and released a new DFXML API, Objects.py11 to provide additional functionality.

In summary, this section has explored academic literature resources and provided examples of reverse engineering techniques and tools used to aid digital forensic research. The express goal of reverse engineering in this research project is to determine the digital artifacts that are uniquely associated with application software. However, a number of challenges exist that hinder the effective and efficient use of reverse engineering in digital forensics research.
3.2.3 Reverse Engineering Challenges

A variety of techniques and an assortment of tools to perform system-level reverse engineering of application software has highlighted a collection of fundamental problems when performing reverse engineering as an aid to digital forensics. Challenges include a lack of standardisation surrounding reverse engineering techniques and tools, problems with experimental testing environments and methods, as well as a high prevalence of unrelated results from reverse engineering caused by operating system noise.

3.2.3.1 Lack of Standardisation

Although reverse engineering tools, specifically differential analysis tools, are widely used in digital forensics to achieve a variety of reverse engineering tasks, there is no standardisation of the procedures, techniques or tools that should be used. The following direct quote eloquently summarises the current situation:

Many of today’s DF [digital forensic] engineering resources are dedicated to reverse engineering hardware and software artifacts that have been developed by the global IT economy and sold without restrictions into the marketplace. But despite the resources being expended, researchers lack a systematic approach to reverse engineering. There is no standard set of tools or procedure. There is little automation. As a result, each project is a stand-alone endeavor, and the results of one project generally cannot exchange data or high-level processing with other tools in today’s forensic kit (Garfinkel, 2010, p. S68).

In later research, Garfinkel et al. (2012) specified a general strategy for differential forensic analysis which provides an abstract differencing procedure to suit a variety of evidence sources (disk images, network traffic and volatile memory captures). Although such research helps to standardise the technology implemented to perform reverse engineering, there are still no standardised tools available to reverse engineer application software. Instead, researchers and practitioners implement techniques lacking digital forensic requirements and tool that lack digital forensic functionality.

Commonly used tools, such as Regshot, were never designed for use in digital forensic research and lack the functionality required for the specialist area of forensic analysis; for example, Regshot provides limited metadata when reporting system-level changes and does not include important information such as cryptographic hash values for identified files. Additionally, the text-based output of most differencing tools is not easily parsed and usually requires manual analysis or an API authored and tested to handle each output format. Furthermore, each tool generates a different output syntax, requiring a different method to parse the results from each tool. It is also difficult to share the output from reverse engineering tools when each tool generates different output formats.
3.2.3.2 Experimental Testing Challenges

Two foremost challenges still exist when performing reverse engineering in the realm of digital forensic research: 1) Contamination of the testing environment; and 2) Non-standardised experimental testing methods.

Although the discussed reverse engineering tools are useful at detecting and recording system-level changes, they can also introduce unrelated data into a testing environment. “The use of a system monitoring tool to collect data during experimentation can introduce errors into an experiment” (Casey, 2013, p. 167). Quick and Choo (2013) agree with this position, stating that system monitoring tools can introduce unintended changes to the monitored system. The same problem exists for any system monitoring or differential analysis tool that is being executed on a running system where the use of the tool creates additional digital artifacts on the system. Therefore, tools should only be used on a system which is independent to the actual system used during testing. A simple method to achieve this is to implement Virtual Machines (VM) and clone the same virtual hard drive used for testing to an independent system. This creates an exact replica of the test system which can be used for system monitoring purposes and provides a controlled and contaminant free test system environment. However, the previously mentioned idifference.py and rdifference.py tools do not suffer from this issue because the tools are not executed on a live system. This means that unintended changes cannot be made to the target system being analysed. In comparison, if live system monitoring tools need to be executed on the same testing system it is necessary to test the tools to determine the digital artifacts that have been created. The identified digital artifacts can then be documented so that the entries can be deleted from any results.

A further challenge with experimental testing using reverse engineering is the method or procedure of how an application is reverse engineered. There are a large number of variables that exist when reverse engineering of an application software is performed:

1) There exists a variety of different application life cycle phases which can be recreated and reverse engineered.

2) There exists a variety of different operating systems on which to perform reverse engineering of application software. Since each operating system introduces changes to what digital artifacts are produced the output will be different. This is true for different operating system platforms (e.g., Windows versus Linux) and even in the same family of operating systems (e.g., Window XP versus Windows 7).

3) There exists a variety of different application software versions to perform experimental testing. Similarly to different operating system versions, different application software versions can produce different results.

Each research effort commonly analyses a single application software version on a single operating system platform or version and can produce dramatically different results when compared to a different experimental testing environments. The challenge of a variable testing
environment culminates in one research study being directly linked to a single operating system (platform and version) and a single application version. Due to the extensive resources required to perform reverse engineering research to aid digital forensics, it is currently not feasible to test every application version on every potential operating system. As also stated, each research study can pick and choose which of the application life cycle phases to investigate. To combat this, a generic testing procedure needs to be implemented.

3.2.3.3 Prevalence of Unrelated Results

Another challenge when performing reverse engineering of application software is that a high number of unrelated results are commonly detected and reported. This is because modern operating systems are highly complex and even when the system is idle (not actively running any user-level applications or tasks) there can be services, processes and network activity that creates system noise.

To determine the behaviour of software on a complex operating system is a difficult endeavour. Many system events occur even when an operating system is idle. Thousands of events are generated that would overwhelm an analyst if one would simply listen to all events. On a clean, idle Windows XP SP2 installation, we observed 530 registry events and 60 file events within 1 min (Seifert et al., 2007, p. S24).

To support this argument, Garfinkel et al. (2012) performed a differential forensic analysis on the M57-Patents scenario and discovered that realistic usage of a Windows XP system over one (1) day generated approximately 5,000 new files, 200 deleted files and 700 files with modified content. Garfinkel et al. (2012) documented that the majority of file system changes appeared to be the result of non-interactive processes (system services), Windows updates and application software updates. The high number of results that are not directly related to an application makes it difficult to determine what digital artifacts are actually associated with, or unique to, the application. This leaves the challenge of filtering unrelated results. However, various solutions have been proposed.

Carvey (2011) states that when using differencing tools to perform Windows Registry analysis the best practice is to collect a snapshot before and after a single task, defined as an atomic action; for example, collecting a system snapshot before and after installing the application. By using atomic actions the time taken between collecting the two system snapshots should reduce possible extraneous information that is not specifically related to the application being investigated. By using a shorter time frame or smaller task the system activity between snapshots should result in a reduced number of differential analysis results. Kälber, Dewald, and Freiling (2013) proposed a consensus building approach when reverse engineering application software by performing multiple runs for each atomic action to build metadata profiles. For each application being profiled they performed each atomic action (e.g., installing the application) a total of 100 times, the returned results were compared and only the digital artifacts that were present in all 100 profiles were selected. This technique
is similar to the mathematical principle of set intersection, where the intersection of two sets is the objects that reside in both input sets. The premise is that specific irrelevant entries may be present in some test results, but not all. Therefore, only entries that are unique to the application will be present every reverse engineering report. (James, Gladyshev, & Zhu, 2011) performed a similar consensus building approach, by running Process Monitor 400 times per test and selecting only entries that were present in every test. Using this approach the number of detected Registry entries was reduced from approximately 11,000 to 4,000. However, “consistent noise was still found to be present” (James et al., 2011, p. 99). A similar technique to using a consensus building approach could be to use multiple tools to reverse engineer the same scenario on the same test system and compare the output from different tools.

Seifert et al. (2007) state that normal system events are constantly generated and therefore, adopted the use of exclusion lists to omit results from the final report. Exclusion lists are also commonly referred to as blacklists and contain content that should be ignored. However, this requires knowledge of the digital artifacts that a specific application creates before performing reverse engineering.

The sources of literature consulted in reverse engineering of application software has featured not just the various techniques and procedures used, but also identified and highlights some shortcomings. Potential scope for improvements with an emphasis on standardisation, functionality and digital forensic requirements are a few of the challenges to be addressed by future research.

### 3.3 Automated Digital Forensic Analysis

The previous section discussed the use of reverse engineering as a resource to aid digital forensic analysis. The generated outcome of this process is the identification of the digital artifacts associated with application software. This information is classified as known content that can be linked to a specific application. In order to establish the presence, or absence, of application software on a target data set, the previously known content can be searched for. If a match is detected it can be concluded that the application is, in fact, present on the system. Likewise, if not, it can be concluded that the application is not present on the system.

The importance of digital artifacts was previously discussed (see Section 2.2.1) but to reiterate, digital artifacts are a fundamental source of evidence in digital investigations which is simply digital data that is logically grouped; for example, file system entries (e.g., data files and directories) and Windows Registry entries (e.g., Registry keys and values). Digital forensic analysis commonly involves searching electronic data for digital artifacts of interest within each of these entities. This provides the information, and ultimately the evidence of the actions that a suspect has performed.

This section discusses literature that is relevant to automating forensic analysis, specifically automated detection of digital artifacts from application software. File hashing techniques of
data files coupled with the implementation of reference sets is first outlined followed by a summary of alternative hashing algorithms to enhance data file matching, including block-based hashing and similarity digests. An in-depth review of the National Institute of Standards and Technology (NIST) Diskprint project then provides a wider scope for reference set contents. Finally presented is a brief appraisal of Windows Registry analysis to aid application software detection including the techniques, tools and available data abstractions, culminating in the lack of automated processes available to perform Registry investigation.

3.3.1 File Hashing and Reference Sets

According to Roussev (2009), hashing is a fundamental tool in digital forensics to aid identification of data files. A cryptographic hash function (or algorithm) takes an arbitrary length input and produces a fixed length output, usually called a digest or hash value (Ferguson & Schneier, 2003). Hash functions are useful to identify data files that are exactly the same because they are collision resistant, which means that finding two different inputs with the same output is not computationally feasible. Therefore, two data files with different content could never have the same hash value – this statement is only valid if a secure and reliable cryptographic hash function is implemented. Given this property, hash functions are commonly used to perform identification of data files by hashing every file on a target data set and comparing the calculated hash values against a database of known file hashes.

The use of reference sets in conjunction with hash values is a fundamental digital investigation technique used to automate detection of known content. Reference sets, commonly referred to as reference databases or hash sets, are a document populated with metadata about known data files, including the file name, file size and a cryptographic hash value.

The general approach is to construct a reference set of known data and then use a forensic tool to automatically search the target to filter irrelevant or relevant artifacts as the case may be. Since the common unit of interest is a file, this technique is often referred to as known file filtering. The standard known file filtering approach is to use cryptographic hashes, such as MD5 and SHA1, and construct tables of known hash values (e.g., of operating system and application installation files) (Roussev, 2012, p. 19-20).

A populated reference set is, therefore, a document containing known files and the associated hash values. To perform forensic analysis, the target device is processed, each file hashed and then compared to the hash set. If a match is found in the hash set, it can be concluded that the files are the same. Reference sets have two primary uses: 1) To identify relevant content (e.g., illicit digital images); or 2) Filter irrelevant content (e.g., known operating system files). A well known example of a reference set for application software is maintained and distributed by the National Institute of Standards and Technology (NIST) which operates the National Software Reference Library (NRSL). The NSRL distributes a Reference Data Set (RDS) comprised of hash values for a variety of commercial software (NIST, 2015b). In the RDS version 2.41, there are a total of 14,503 different profiled applications available.
The RDS is used by law enforcement, government, and industry organizations to review files on a computer by matching file profiles in the RDS using an automated system. The reference data is used to rapidly identify files on computer systems, based solely on the content of the files. In most cases, NSRL file data is used to eliminate known files, such as operating system and application files, during criminal forensic investigations. This reduces the number of files that must be manually examined and thus increases the efficiency of the investigation (Lyle, White, & Ayers, 2008a, p. 1).

The RDS, therefore, is commonly used to perform data reduction by filtering known files that are not of forensic interest. However, the RDS can also be used to detect relevant files; for example, to determine which applications are present on a system. Such information regarding the presence of an application may provide insight of how the system was being used and where potential evidence may reside (Mead, 2006); for example, if a data wiping tool is detected on an investigation target it is possible that the suspect may have attempted to securely remove files. This provides a lead for the investigator to further analyse the available evidence in an attempt to discover additional or corroborating evidence including application log files, configuration files, or pertinent Registry entries. Listing 3.1 displays a single example of a file entry taken from the RDS. The file (WORD.EXE) is displayed with associated metadata including MD5 and SHA1 hash values, file name, file size and other product and operating system information.

```
"SHA-1","MD5","CRC32","FileName","FileSize","ProductCode","OpSystemCode","SpecialCode"
"AC91EF00F33F12D491CC91EF00F33F12D491CA","DC2311FFDC0015FCCC12130FF145DEE78","14CCE9061FFDC001","WORD.EXE",1217654,103,"T4WKS",""
```

Listing 3.1: RDS example entry (Source: Listing taken from NIST (2009))

As seen in Listing 3.1, the hash values included are Message Digest version 5 (MD5) and Secure Hash Algorithm version 1 (SHA1). Both of these are commonly used hash functions in digital forensics for both preservation integrity and for performing known file filtering. However, there is debate about the validity and reliability of both MD5 and SHA1 hash collisions that has been presented in various research papers. Nevertheless, the requirements of hash function usage in digital forensics differs from information security requirements. Thompson (2005) states that potential MD5 collisions in digital forensics in unlike that in computer security and does not pose a threat in the same context. Furthermore, “it can be clearly stated that the use of MD5 and SHA1 hashing within the field of digital forensics remains a valid scientific practice” (Schmitt & Jordaan, 2013, p. 42).

Although the use of the MD5 and SHA1 hash functions are still widely accepted and reliable in digital forensics there still exists an underlying problem with traditional file hashing techniques. Small modifications to a file (even modifying a single bit in a file or one word in a document) creates a completely different hash value resulting in the inability to detect similar
files using MD5 or similar traditional hash functions. This is problematic as two almost identical files cannot be matched. This difficulty is caused by the fragility of hash functions and led to the digital forensic research community focussing on improvements to file hashing technology.

3.3.2 Advanced Forensic Hashing Techniques

The major drawback to the use of traditional file hashing in digital forensics is that only exact matching can be performed, because it is infeasible to modify a message without changing the hash, meaning that a secure hash function will produce completely different output if even one bit of the input message (or data file) is changed. Nevertheless, “files with one-bit changes are almost entirely identical and share a large ordered homology” ([Kornblum] (2006, p. S92). A real world example would be when a single word in a document is changed, resulting in a completely different hash value even though both documents share a majority of similar content. As a consequence further advancement in digital forensics has attempted to develop hashing algorithms specifically to detect files with only slight differences. The design and use of block hashing to perform sub-file forensics as well as approximate matching using similarity digests are outcomes of recent research.

3.3.2.1 Block Hashing

File hashing computes a hash value for an entire file. In contrast, block hashing is a technique that calculates hash values for individual blocks of data using a constant block size ([Young, Foster, Garfinkel, & Fairbanks] (2012). Block hashing has a number of different terms including: block-based hashing, piecewise hashing, and sector hashing. The generic term block hashing is used in this research. Block hashing splits a data file, disk image or another form of structured data into separate fixed sized blocks (e.g., 512 byte blocks) which are then used as input to a hash function. Figure 3.2 displays a visual representation of how block hashing operates on a file. As illustrated, the file is divided into blocks of equal size, used as input to a hash function and result in each block having a unique hash value.

![Figure 3.2: Visual representation of block hashing performed on a data file (Source: Figure adapted from [Kornblum] (2011, p. 10); Image sources: file icon is public domain)
The first implementation of block hashing for digital forensics usage was developed by Nick Harbour who created `dcfldd`, a modified version of the `dd` tool to adhere to forensic requirements when performing data collection. “Harbour subsequently modified `dcfldd` to compute hashes over segments of the disk image so that if it was inadvertently modified, a chain of custody could be maintained for at least part of the image. He called this piecewise hashing” (Young et al., 2012, p. 30). This technique aided preservation of evidence integrity, but was not yet implemented to aid file detection. Block hashing received additional development when Jesse Kornblum released the `hashdeep` tool. The main advancement was that the functionality to compute block hashes to aid detection of data file fragments was also achieved. A reference set can be created using `hashdeep` which has the ability to compare the created reference set against a target forensic disk image or collection of files.

**Sector hashing** is an extension of block hashing logic which uses specifically selected block sizes that are commonly encountered sector sizes in file systems. Foster (2012) first proposed the use of sector hashing to perform rapid content identification by hashing sector size blocks (e.g., 512 or 4096 bytes) and comparing each sector hash to a database of known sector hash values. A major advantage to sector-level processing (see Figure 2.4) is that it is file system independent and, therefore, requires no knowledge of the overlying file system. This is useful as the file system may be either unavailable or corrupt. Furthermore, sector-level processing can be highly parallelised as it is possible for data to be sequentially read and processed from a target device. This does not apply to file hashing, as files are not sequentially stored and each file needs to be located before being hashed. “File-based data access tends to generate a non-sequential disk access pattern, which can seriously degrade throughput” (Roussev, 2009, p. 50). Figure 3.3 displays an example of hard drive throughput which compares the average speed in Megabytes per second (MB/s) at which a 500 Gigabyte (GB) hard drive can be read at different sequential levels.

As shown in Figure 3.3 reading a hard drive 100% sequentially results in an average throughput of 68 MB/s. In contrast, reading the same hard drive 98% sequentially the average throughput lowers to 48 MB/s (therefore, 2% of data is read using random disk seeks). This is a 30% performance penalty when compared to sequential disk reading and illustrates the performance advantages of sector-level hashing.

Another advantage of both block and sector hashing is the potential identification of not just entire files, but also file fragments (Garfinkel et al., 2010). Detection of file fragments provides the capability to detect previously deleted files where fragments may still exist in unallocated (free) space when complete file recovery is not possible. Block hashing is also used to detect files where small changes have been made, but where a large proportion of the file remains the same. As with other hashing techniques, block or sector hashes can be used to create a reference set of known content, the only difference from traditional hash sets is that instead of storing one hash value for each file (e.g., one MD5 hash value), multiple hashes are stored for each known file. The number of hashes stored varies based on the size of the file and the specified block size.

---

12See: https://github.com/jessek/hashdeep
Block hashing remains an active digital forensic research area with numerous contributions being made to the underlying technology and practical application in the form of computer forensic tool development. Garfinkel and McCarrin (2015) developed an advanced algorithm to aid hash-based carving using block hashing. Taguchi (2013) leveraged the use of sector hashing to perform rapid triage of large investigation targets. Furthermore, the practical application of block hashing has been advanced through a variety of robust forensic analysis tools. The hashdeep tool creates reference sets comprised of block hashes and compares them to a target forensic disk image or set of files. The bulk_extractor tool has the functionality to perform highly parallelised sector hashing and comparison against a target. Finally, the hashdb tool is a purpose block hashing database built for fast comparison of block hashes.

The use of block hashing and sector hashing enables detection of files and file fragments. However, a crucial file detection challenge arises when a perpetrator inputs just one bit of data into a file. “If we insert a single character into a file, all the block hashes following the change will also change” (Roussev, 2009, p. 52). To combat this problem and address the shortcomings of both file hashing and block hashing, research has been conducted in the field of approximate matching.

3.3.2.2 Approximate Matching and Similarity Digests

According to NIST (2013, p. 1), “approximate matching is a technique for identifying similarities between or among digital objects, such as files, storage media, or network streams, which..."
do not match exactly”. In terms of data file matching, it can be seen that previous forensic analysis techniques have primarily relied on performing **exact matching** using traditional hash functions such as **MD5** and **SHA1**. Either the hashes match, and the file is known, or the hashes don’t match and the file is not known. **Approximate matching** improves on previous matching techniques by specifying a comparison based on a degree of similarity between the two files ([Breitinger & Roussev, 2014]).

[NIST, 2013] state that an approximate matching algorithm should perform at least one of the following four matching functions: 1) Object similarity detection (e.g., two different versions of a document that has been slightly modified); 2) Embedded object detection (e.g., an image in a document); 3) Fragment detection (e.g., a cropped image); and 4) Cross correlation (e.g., detecting files in different targets). Furthermore, approximate matching algorithms can be used for enhancing black or white listing using reference sets and to perform object grouping. NIST (2013) specify that the ssdeep and sdhash tools are both being implemented in the NSRL project. The following paragraphs provide a brief discussion of each tool.

Kornblum (2006) first proposed the use of approximate matching using a hashing technique called **context triggered piecewise hashing** (CTPH). According to Kornblum (2006, p. 91), CTPH is a “technique for constructing hash signatures by combining a number of traditional hashes whose boundaries are determined by the context of the input”. The technique is based on an e-mail spam detection tool, dubbed spamsum, which compares the similarity of two e-mails by generating and testing the signature of an email compared to known spam. A similarity metric of the two e-mails with a value between 0 and 100 is provided (Tridgell, 2009). The implementation of CTPH resulted in the release of the ssdeep proof-of-concept tool to compute and compare hashes. Listing 3.2 displays an example hash value created by the ssdeep tool. Line 1 displays the execute command, line 2 displays the ssdeep header, and line 3 displays the actual signature where each component is separated by a colon. As illustrated, the ssdeep hash value is dramatically different from a traditional hash value (e.g., **MD5**).

Listing 3.2: Example of an ssdeep hash value (Source: Listing taken from Kornblum (2006))

<table>
<thead>
<tr>
<th>$ ssdeep Msdosdrv.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdeep,1.0 --blocksize:hash:hash, filename</td>
</tr>
<tr>
<td>384:6A:A46SBSZHJEi4gMOzscKThLXmdolp72mzdldM72132fMENY2PDr20ypz installs:KQx+</td>
</tr>
<tr>
<td>AecKumvlAN20sYyX5uR,&quot; Msdosdrv.txt &quot;</td>
</tr>
</tbody>
</table>

Roussev et al. (2007) proposed the use of similarity hashing to produce an efficient and scalable technique to identify similar objects. In later research, Roussev (2010) released the sdhash tool to generate and compare similarity digests. According to Breitinger and Roussev (2014), the sdhash tool addressed the short-comings of ssdeep by selecting statistically improbable features to represent each object instead of simply dividing a file into chunks (as implemented in ssdeep). Similar to all approximate matching algorithms, similarity digests can be compared on a value of 0-100 to highlight the similarity between two objects. Roussev
(2011) performed a thorough evaluation that compared the `ssdeep` and `sdhash` tools in a variety of forensic analysis scenarios to determine capabilities of each tool.

The use of advanced forensic hashing techniques to aid in automated detection of digital artifacts, specifically data files using `ssdeep` and `sdhash` has made progress. However, both techniques and associated tools have never been tested for detecting data files and similarities in the realm of application software. Nevertheless, performance evaluations in terms of computational efficiency have been published and is discussed in the following subsection.

### 3.3.3 Performance Summary of Forensic Hashing Tools

Tool efficiency is critical in digital forensics due to the challenge of data volume (see Figure 1.1 and Section 2.3.1). Furthermore, performance is exceptionally important when conducting digital forensic triage which attempts to execute a partial forensic examination under significant time and resource constraints (see Section 2.1.2). Therefore, it is prudent to investigate the performance of available hashing techniques so that the selection of hash functions for the system design in this research can be later made.

Hash function efficiency can be divided into two phases: 1) Hash function generation (e.g., computing the hash value of a file); and 2) Hash value comparison (e.g., comparing two previously generated hash values). Table 3.6 displays the efficiency of hash value generation for the `MD5`, `SHA1`, `ssdeep`, and `sdhash` functions. The run-time efficiency (displayed in seconds) is given for a 500MB test file filled with random data (generated from `/dev/urandom`). Row 2 displays a performance comparison using `MD5` as a benchmark, that is, the run-time generation efficiency of each hash function compared to the `MD5` algorithm. Similarly, row 3 displays another performance comparison using `SHA1` as a benchmark.

Table 3.6: Run-time generation efficiency of common digital forensic hash functions (Source: Table adapted from Breitinger and Petrov (2013, p. 112))

<table>
<thead>
<tr>
<th></th>
<th>MD5</th>
<th>SHA1</th>
<th>ssdeep</th>
<th>sdhash</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime efficiency</td>
<td>1.35</td>
<td>2.33</td>
<td>6.48</td>
<td>22.82</td>
</tr>
<tr>
<td>algorithm (MD5)</td>
<td>1.00</td>
<td>1.72</td>
<td>4.80</td>
<td>16.90</td>
</tr>
<tr>
<td>algorithm (SHA1)</td>
<td>0.58</td>
<td>1.00</td>
<td>2.78</td>
<td>9.87</td>
</tr>
</tbody>
</table>

As displayed in Table 3.6, `MD5` is the most efficient hash function, taking 1.35 seconds to compute a hash value from the 500MB file. `SHA1` takes almost twice as long, while `ssdeep` takes approximately 5 times as long and `sdhash` takes almost 17 times as long. Similar trends are seen when comparing `SHA1` to the `ssdeep` and `sdhash` tools. Simply put, traditional hash functions are much more efficient when compared to approximate matching algorithms.

The second efficiency phase is correlation of previously computed hash values. "Fast fingerprint comparison is part of the runtime efficiency as an approach is only useful if it has a..."
fast comparison function” (Breitinger, Stivaktakis, & Baier, 2013, p. S53). Table 3.7 displays results from fingerprint generation and comparison for SHA1, ssdeep and sdhash algorithms using the t5 corpus\(^\text{16}\) a publicly available data set comprised of 4,457 files equating approximately 1.78 GB. Results are displayed for the average hash generation time for each tool, total hash generation time for each tool and hash value correlation time. Similar to the information displayed in Table 3.6 a benchmark of hash value correlation efficiency is included using the SHA1 hash function as the benchmark.

### Table 3.7: Generation and comparison efficiency for common digital forensic hash functions
(Source: Table adapted from Breitinger et al. (2013))

<table>
<thead>
<tr>
<th>Tool</th>
<th>Average</th>
<th>Total</th>
<th>Hash value correlation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sha1sum</td>
<td>0.0013</td>
<td>5.632</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>ssdeep -s</td>
<td>0.0089</td>
<td>39.789</td>
<td>18.217</td>
<td>7.06</td>
</tr>
<tr>
<td>sdhash</td>
<td>0.0167</td>
<td>74.278</td>
<td>356.730</td>
<td>13.19</td>
</tr>
<tr>
<td>sdhash -p4</td>
<td>0.0066</td>
<td>29.382</td>
<td>346.902</td>
<td>5.22</td>
</tr>
</tbody>
</table>

Once again, as seen in Table 3.7, traditional hash functions (e.g., SHA1) are much more efficient compared to approximate matching algorithms (e.g., ssdeep and sdhash). SHA1 fingerprint comparison is not displayed as it is a simple string comparison and requires minimal time to compare two hash values. The ssdeep tool required 18 seconds to perform a comparison, compared to approximately 350 seconds for sdhash. These results highlight the inefficiency of approximate matching algorithms compared to traditional hash functions.

In summary, knowledge has been gained from the comparison of traditional hashing functions with advanced hashing algorithms. It is evident that traditional hash functions are much more computationally efficient, yet lack the capability to detect similar files. Comparatively, advanced approximate matching algorithms can detect similar files, but are much more computationally intensive. This is true for both fingerprint generation and correlation.

The following sections continue to investigate literature relevant to automated detection of digital artifacts. A new research direction at the National Institute of Standards and Technology (NIST) is to create reference sets specifically tailored for detection of digital artifacts from operating systems and application software.

### 3.3.4 NIST Diskprint Project

According to NIST (2015a), the NSRL RDS contains file-based metadata of application software which is generated from installation media; for example, the files that make up the software distribution (e.g., application installer) and the actual files created during installation (e.g., executable, library and configuration files). However, this information only informs a practitioner if an application has been installed on an investigation target. Recent research at NIST has focussed on advancing the technology and techniques used to automate forensic

\(^{16}\)See: [http://roussev.net/t5/t5.html](http://roussev.net/t5/t5.html)
analysis of data files. The new project, dubbed **diskprints**, attempts to progress the use of reference data sets to aid forensic analysis of application software by collecting digital artifacts from the entire application life cycle.

[Tebbutt (2012)] initially introduced the project as **application footprints** with the aim of tracking the lifetime of application software. The project aims to determine the file system, Registry and volatile memory changes that occur at different stages of the application life cycle. To achieve this Virtual Machines (VM) are specified as the **state** of the system can be captured at any time by simply pausing the VM and copying the evidence sources (virtual hard drive, memory contents). [Tebbutt (2012)] referred to the collected data as **slices**, because they represent a slice of the overall application life cycle. One key detail is that the life cycle is manually created by an investigator who is required to perform and document the process. Figure 3.4 displays an example of the diskprint analysis workflow for the installation of the Mozilla Firefox web browser. The first **slice** is a fresh installation of Microsoft Windows, while the second is after the installation of Firefox. Differencing is achieved by comparing the two **slices**, specifically, two DFXML reports. The result is a **delta** DFXML report (\(\Delta\).dfxml) which documents the file system changes that have occurred as a result of the installation of Firefox.

![Diskprint analysis workflow](image)

Figure 3.4: Diskprint analysis workflow (Source: Figure adapted from [Nelson, Laamanen, et al. (2014, p. 24)]); Image sources: Mozilla Firefox icon taken from [The Mozilla Foundation (2013)], other images are public domain.)

Later research outlined by [Laamanen & Nelson, 2014] specified target operating systems including Microsoft Windows XP, Vista, 7 and 8 in 32-bit and 64-bit variants. Additionally, diskprints are being created for a variety of popular web browsers, network tools, archivers and messenger applications. Furthermore, details were included for the collection of network traffic at each phase and that automated collection was being developed. Additional research by [Nelson, Laamanen, et al. (2014)] outlined information regarding Windows Registry detection utilising the method specified and techniques implemented in the diskprint project.
To date, 33 diskprints are available for a total of three operating systems and 16 applications. Only one research presentation has been published regarding performance and evaluation of the diskprint project. Jones et al. (2015) performed an analysis of past-activity of application software using block hashing to detect files and file fragments. They were able to detect file fragments (blocks) that remained after uninstallation of an application even after normal system activity continued. Additionally, a weighted score was developed to determine the percentage of matched artifact fragments. Jones et al. (2015) specified that future research should include enhancement of computation, sector differencing and noise reduction at collection and extension of the method to malware, mobile and memory artifacts.

Overall, the NIST diskprint project has advanced the fundamental technology that is used to create reference sets for application software. As an ongoing research project, it presents a much more comprehensive technique to automate the process of detecting the presence of application software on an investigation target compared to previous approaches. In addition is the functionality to classify the application life cycle phase (e.g., execution) of detected digital artifacts. This is exceptionally useful information to provide an investigator regarding application software usage.

3.3.5 Automated Detection of Registry Entries

A brief introduction to the Windows Registry was previously outlined including the potential evidential value, an overview of hive files and types as well as a brief summary of the Registry structure (see Section 2.2.2.2). To reiterate, the Windows Registry is a hierarchical database used to store configuration information for users, applications and hardware devices. Given the information that the Registry stores, it is an exceptionally useful resource to aid digital investigations. This section provides a summary of literature and software available to investigate the Windows Registry to aid detection of digital artifacts directly relevant to application software. Potential Registry analysis tools are first presented, followed by previous research which has focussed on automating Registry content analysis.

3.3.5.1 Windows Registry Analysis Tools

Due to the evidentiary value of the Windows Registry there are a wide range of computer forensic analysis tools designed specifically to parse, process, examine and analyse different aspects of the Registry. Available tools range from simple command line applications to search the Registry, to advanced solutions with Graphical User Interfaces (GUI) to discover forensically interesting Registry entries and generate reports to document potentially useful evidence. Carvey (2011) provided a list of various tools available to perform Windows Registry forensics including live analysis tools, monitoring tools and post-mortem forensic analysis tools. Table 3.8 displays a list of common Windows Registry tools used to aid forensic analysis. The tool name is provided followed by classification of operation on either live or offline Registry hive files and an associated description of tool functionality.

See: http://www.nsrl.nist.gov/dskprt/sequence.html
Table 3.8: List of common Windows Registry analysis tools (Source: Table populated from Carvey (2011, §§ 2.1-2.2))

<table>
<thead>
<tr>
<th>Tool</th>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg.exe</td>
<td>Live</td>
<td>A native command line tool included with Microsoft Windows to read, write, modify or delete Registry entries</td>
</tr>
<tr>
<td>regedit.exe</td>
<td>Live</td>
<td>A native Registry tool to read, write, modify and delete Registry entries</td>
</tr>
<tr>
<td>Autoruns</td>
<td>Live/Offline</td>
<td>GUI tool which classifies Registry entries based on potential information including Windows Explorer information, scheduled tasks, system services and logon information</td>
</tr>
<tr>
<td>Scripting</td>
<td>Live/Offline</td>
<td>A variety of programming languages provide live and offline access to Registry hive files including Windows Management Infrastructure (WMI), Windows Script Host (WSH), Windows PowerShell, Perl and Python</td>
</tr>
<tr>
<td>MiTeC Registry</td>
<td>Offline</td>
<td>Provides a GUI interface similar to regedit.exe to browse and search Registry entries</td>
</tr>
<tr>
<td>RegRipper</td>
<td>Offline</td>
<td>A GUI tool to import offline Registry hive and scan using a variety of plugins to extract forensically interesting information including autorun and logon entries</td>
</tr>
</tbody>
</table>

3.3.5.2 Automated Windows Registry Analysis

Very few forensic techniques and associated tools are available to perform automated analysis of the Windows Registry. However, this section will outline a collection of semi-automated analysis techniques including string searching and Registry entry detection using high-level forensic data abstractions.

A common analysis technique involves **string searching** the Registry using keywords in an attempt to discover evidence. According to [N. L. Beebe and Clark (2007, p. 49)], “digital forensic text string searches are designed to search every byte of the digital evidence, at the physical level, to locate specific text strings of interest to the investigation”. However, in many scenarios a string search of the Registry may not yield many results as the data is stored in a binary format and, in some cases, subjected to various encodings; for example, is the use of the ROT-13 basic encryption algorithm for UserAssist Registry value entries (Carvey, 2005). Mee, Tryfonas, and Sutherland (2006) state that most Registry viewers have search functions which an investigator can leverage to discover application software; for example, in an investigation involving a specific peer-to-peer application, named KaZaA, the application name can be used to search for potentially interesting Registry entries. This is possible because a Registry viewer transforms the binary structure of the Registry, thus, allowing searching strings that are not in the native binary format. However, string searching for applications using keywords is only successful when the desired application has an uncommon name. Applications including Google Chrome may yield numerous entries if the keyword *chrome* was used. Furthermore,
string searching has minimal automation to both discover potential keywords and to analyse search output.

In terms of advanced automated Registry analysis it is essential to implement a high-level data abstraction to provide easier mechanisms to search and process Registry entries. An example of this is the Registry XML (RegXML) data abstraction. Nelson (2012) developed RegXML to represent Windows Registry hive files by documenting the logical structure and location of data found within hive files. The research was released with supporting software, named regxml_extractor, designed to extract Registry hive files from a forensic disk image and then parse and document entries in the RegXML data abstraction. An associated programming library was also developed which allows an investigator to write simple scripts to read all Registry entries and perform searches. Aside from this example, there are very few research papers or forensic analysis tools available to automate processing and analysis of the Registry. In particular, minimal research is available that has focussed directly on Registry analysis to aid application software detection.

### 3.3.6 Automated Forensic Analysis Challenges

This section has outlined and discussed literature, forensic analysis techniques and the de facto industry tools to perform automated detection of digital artifacts. Two primary digital artifact types were investigated: data files and Registry entries. A number of automated analysis techniques and associated tools are available for detecting data files. The use of reference sets and file hashing is prominent in digital investigations and in scenarios where traditional file hashing fails, researchers have developed more flexible hashing approaches using block hashing and approximate matching. Both are suitable to create reference sets to represent known content used in automated detection of file system entries. However, in terms of automated Registry analysis, there are very few solutions available. This section highlights specific challenges centred around performing automated detection of digital artifacts with specific relevance to identifying application software on a target data set.

#### 3.3.6.1 Automated Detection Performance

Due to data volume challenges in digital forensics, rapid automated solutions are required. The performance of file hashing was investigated (see Section [3.3.3](#)) which highlighted that traditional file hashing (e.g., MD5 and SHA1) present viable and computationally efficient solutions for known file identification. However, they lack the flexibility to detect files which are similar. Block-based hashing and approximate matching were developed to address this challenge. However, the performance overhead is high, even when factoring in performance gained for sequential disk processing. These problems culminate with the challenge of developing a solution that is both flexible enough to detect similar files while also being computationally efficient to perform rapid application software detection on a target data set.
3.3.6.2 Digital Artifact Diversity

As previous research has noted, the types of digital artifacts produced by application software are diverse. According to Tebbutt (2012), digital artifact sources from application software may encompass file system entries, system configuration information, volatile memory information and network traffic. Garfinkel (2010) agrees with this position, stating that an application software reference set could be populated with all of the above digital artifact sources. However, no such solution exists to determine and subsequently correlate such a diverse range of digital artifacts.

From the reviewed literature, the only robust solutions to automate detection of digital artifacts are data file centric; for example, reference sets of known data files with an associated hash value. Such solutions only differ based on the hash function or algorithm to generate the associated hash value. These solutions do not even include file system directory (folder) detection. This is because directories lack the logical file contents required to generate a hash value. The same problem exists for Registry entry detection, where no reference set solution exists as Registry entries have no content and are not applicable to hash generation or comparison. This culminates in research challenges, of application software that creates a diverse range of digital artifact types, where only data files have a robust automated detection solution. There is a wealth of potentially useful evidence that an investigator has to manually analyse. Given the digital investigation data volume challenge an alternative automated analysis solution is paramount.

3.4 Conclusion

This chapter has presented and discussed literature that is pertinent to the research project being conducted. A summary of previous research regarding anti-forensic tools was outlined. Research challenges were discovered from this review, specifically that previous research has evaluated tool performance and determined residual evidence after anti-forensic tools usage. The research challenge that persists is the inability to detect anti-forensic tool usage in an automated and timely manner. The technique of reverse engineering was then outlined including system monitoring and differential forensic analysis techniques and associated tools. Research challenges were discussed including a lack of standardisation, experimental testing problems and a prevalence of unrelated results. Finally, a thorough review of automated digital artifact detection methods to perform forensic analysis was covered. The outcome is that there are limited robust solutions to aid detection of digital artifacts from applications, where most solutions are data file centric. The research challenge exists to develop a solution that is built to aid automated software discovery while ensuring computational efficiency.

Chapter 4 leads on from the background information presented in Chapter 2 and literature reviewed in this chapter. Several severe research and real world challenges surround the automated detection of application software, specifically anti-forensic tools. The next chapter specifies the research methodology implemented for this project, a system architecture and forms a set of objectives for a new robust solution.
Chapter 2 provided a background to digital forensics including a discussion on the procedure and techniques for conducting digital investigations as well as digital forensic research problems. The challenges of anti-forensics was introduced with an overview of the techniques and tools that may be used to maliciously remove the availability of digital evidence. Chapter 3 presented and discussed research relating to the forensic analysis of application software, automated forensic analysis, forensic analysis of anti-forensic tools and an appraisal of the current investigation techniques to discover digital artifacts from application software. The aim of this research is to design, implement, demonstrate and evaluate a system to perform automated detection of digital artifacts from application software. Specifically, anti-forensic tools are selected as the application software of choice for this research as they present an interesting and relevant case study.

A research methodology is essential to specify a strategy when undertaking a research project so that reliable knowledge and useful output is achieved. The purpose of this chapter then, is first to select a suitable and sound research methodology for implementation based on various digital forensic research practices and requirements. The outcome is that Design Science (DS) has been selected as the research methodology of choice for this project. The remainder of the chapter then implements the Design Science Research Methodology (DSRM) process model and specifies the following phases: identification of the problem, stating research objectives, the design and development of a solution, demonstration of the proposed system, the evaluation of the system design and communication of research. The experimental testing method is outlined which specifies data sets and metrics for evaluation.
4.1 Digital Forensic Research Methodologies

According to [N. Beebe (2009)], traditional forensic sciences (e.g., toxicology and ballistics) resulted from academic research, thus, allowing science to precede forensic science application. However, digital forensics is a relatively new discipline which has had an unusual emergence into the scientific community.

Originating in the late 1980s as an ad hoc practice to meet the service demand from the law enforcement community, computer forensics has recently developed into a multi-domain discipline crossing the corporate, academic and law enforcement fields ([Guo, Slay, & Beckett (2009), p. S12]).

Palmer (as cited in [N. Beebe (2009, p. 18-19)]) agrees with this premise, stating that digital forensics emerged from the practitioner community where crime investigators and computer forensic tool developers carried out research as a means to solve real world problems. [Nance, Hay, and Bishop (2009)] support this position, stating that digital forensics is a largely practitioner driven field where technology advances are made in a bottom-up approach in reaction to a specific incident. In contrast a top-down approach first identifies a research challenge and then formulates potential solutions. A top-down approach promotes a more robust scientific research methodology to better advance technologies, processes and techniques.

In the field of digital forensics, a variety of methodologies have been used to carry out research. [Baggili, BaAbdallah, Al-Safi, and Marrington (2013)] categorised and analysed a sample of 500 digital forensics academic publications to determine research trends. Of the papers reviewed in the study the findings showed there was an approximate 30% division between empirical, exploratory and constructive research methodologies, while implemented research methods were made up of qualitative (80%) and quantitative research (20%).

In considering the general context of Information Systems (IS), which includes digital forensics, [Nunamaker, Chen, and Purdin (1990)] state that basic research, also called pure research, involves developing and testing theories and hypotheses that are the endeavour of the researcher based on intellectual interest, rather than the need for practical solutions. In contrast, “applied research is the application of knowledge to solve problems of immediate concern” (Blake; Bailey as cited in [Nunamaker et al. (1990), p. 631]). Since digital forensics is a highly practical field, applied research is arguably the most appropriate research methodology.

Applied research is a fundamentally important aspect of digital forensics, providing reliable knowledge of computer processes to support conclusions concerning what activities occurred on a device or network ([Casey, 2013] p.167).

Baggili et al. (2013) also determined that in 81% of the 500 reviewed publications, it was applied research that was initiated to solve practical problems in the research domain. In digital forensic research the outcome of applied research often results in a practical implementation of the solution; for example a computer forensic tool (computer software specifically tailored for digital investigation). Development and implementation of a computer forensic
tool provides the capability to: 1) Demonstrate and evaluate the solution by providing a platform to perform experimental testing; and 2) Communicate the research in an additional output. This provides an increased confidence in forensic tools through publication, review, and formal testing (Manson et al., 2007, p. 226).

Demonstration and evaluation of a tool/software is an exceptionally important aspect of digital forensic research. The National Institute of Standards and Technology (NIST) manages the Computer Forensic Tool Testing (CFTT) project which has established a methodology for testing CF tools using function orientated testing for general tool specifications, test procedures, test criteria, test sets, and test hardware (Lyle, White, & Ayers, 2008b). Development of standardised computer forensic tool testing methods aid in producing reliable and accurate results. Tool testing also provides additional information to the research and industry communities whose feedback can be further used to improve tool functionality.

Publication of the specific techniques that a computer forensic tool uses is highly advantageous to both researchers and practitioners. Open-source software is an excellent platform as it is readily available to end-users and source code can be reviewed to determine the design and operation of the tool. Carrier (2002) states that open-source tools more clearly and comprehensively meet forensic guidelines and requirements than closed source tools. Additionally, publication of the descriptive procedure for a tool is useful to provide a high level overview of design and operation. However, there is a downside to making research open and available. Open-source tools are also available to suspects who, in-turn, then learn digital forensic methods, techniques and tools to potentially counter the forensic process.

It remains that open-source software does aid computer forensic tool advancement as it continues to build on previous research and development. This is also true of academic research in digital forensics, as the need to continually update and improve on previously conducted research outcomes is recognised as being highly beneficial.

Without a clear strategy for enabling research efforts that build upon one another, forensic research will fall behind the market, tools will become increasingly obsolete, and law enforcement, military and other users of computer forensics products will be unable to rely on the results of forensic analysis (Garfinkel, 2010, p. 64). ‘Reproducibility, rigor, transparency, and independent verification are cornerstones of the scientific method’ (McNutt, 2014, p. 679). Speciafically, reproducible research is the concept of replication of the scientific method by other scientists (Fomel & Claerbout, 2009, p. 5).

Research reproducibility is necessary in digital forensics to adhere to scientific principles. Forensic techniques and tools must achieve a consistent standard throughout the investigative procedure including reliable and measurable quality (Pan & Batten, 2005). In addition to the scientific methods used, the utilisation of standardised and publicly available data sets, dubbed digital corpora, are key to advancing research quality (Garfinkel, Farrell, Roussev, & Dinolt, 2009). Digital corpora in the form of disk images, data files, memory dumps and network traffic captures can be used to test forensic analysis techniques and tools on publicly available data, allowing other researchers to replicate results and build on previous solutions.
This research aims to identify and solve an identified problem that affects the real world digital investigation of application software, specifically the automated forensic analysis of application software (anti-forensic tools). Applied research is specified as a desired research method because it produces findings that are relevant to identified problems and focuses on development of a solution to the problem domain. The desired research method requirements have resulted in the selection of an Information Systems (IS) research methodology known as Design Science (DS).

### 4.2 Design Science Research Methodology

“Information Systems (IS) is an applied research discipline” (Peffers et al., 2007, p. 46). Design Science (DS) can be implemented to perform applied research in the field of IS. “The goal of information systems research is to produce knowledge that enables the application of information technology for managerial and organizational purposes” (Hevner & March, 2003, p.111). This statement aligns with the proposed research aim of solving the real world digital investigation problem area of automated forensic analysis of application software, specifically anti-forensic tools (a type of application software). Furthermore, the expected research outcome has a significant relationship to IS research as the overall goal is to further the knowledge and improve the systems, processes, techniques and tools used to conduct digital investigations.

“Design science seeks to create innovations, or artifacts, that embody the ideas, practices, technical capabilities, and products required to efficiently accomplish the analysis, design, implementation, and use of information systems” (Hevner & March, 2003, p.111). Hevner, March, Park, and Ram (2004) then state that, in the realm of IS research, design science aims to create and communicate a purposeful and innovative Information Technology (IT) artifact which addresses an important problem domain.

There have been a variety of DS research methodology models proposed for conducting IS research. Peffers et al. (2007) reviewed the process elements from seven academic research papers (from IS and other disciplines) and synthesised a Design Science Research Methodology (DSRM) process model specifically tailored for IS research. It was constructed using a consensus-building approach to produce a research process model that is established on accepted elements. Figure 4.1 displays the DSRM process model which will be implemented as the framework to conduct this research.

The DSRM process model follows a nominal sequence based on a variety of possible research entry points dependent on the context of the research being conducted. In this research the entry point is a problem-centred initiation as the research topic is a problem area that has been identified from a thorough review of literature. Each of the six individual DSRM elements will be outlined and discussed in the following subsections.
4.2.1 Identify Research Problem and Motivation

The first element in the DSRM process model is problem identification and motivation. This involves the definition statement of the specific research problem, the importance of the problem and specifying the justification of value that the solution will provide (Peffers et al., 2007). Problem identification aims to highlight the requirement and importance of conducting additional research to aid in development of an artifact which will provide an effective solution.

A collection of problems were outlined and discussed in the background material and the literature review.

The background material (Chapter 2) presented an overview of digital forensics and anti-forensics. Section 2.3 outlined a collection of digital forensic research challenges including: 1) The increase of data volume; 2) The increase in case complexity; 3) Problems with forensic analysis technique and tool scalability; and 4) Temporal diversity challenges caused by continual technology advances. Section 2.5.3 discussed a variety of problems centred around anti-forensic techniques and tools including: 1) The prevalence of anti-forensic techniques and tools encountered in digital investigations; 2) Increased sophistication of anti-forensic tools used to remove digital evidence availability; and 3) The advanced knowledge and skill required by investigators to perform forensic analysis of anti-forensic tool activity.

The literature review (Chapter 3) presented and discussed material relating to forensic analysis of anti-forensic tools. A review of previous research surrounding anti-forensic research was presented accompanied by issues with current research direction and lack of technological advancements. Reverse engineering to aid general application software forensics was discussed and problems outlined including: 1) A lack of standardisation; 2) Experimental testing challenges; and 3) The prevalence of unrelated results from reverse engineering techniques. Finally, automated forensic analysis techniques were investigated and challenges surrounding techniques and tools performance was covered, as well as the challenge of digital artifact diversity.
The reviewed literature has led to the identification of a variety of problems when performing automated forensic analysis of application software using reference sets. The current practice is to implement reference sets of known content; for example, data files represented by metadata. A populated reference set is compared to a target data set and matching is performed usually by comparison of cryptographic hash values to match data files. When a match is found it is reported to the investigator. Reference sets have two main filtering purposes: 1) To identify digital artifacts of forensic interest; and 2) To perform data reduction by filtering irrelevant content. The use of known file filtering is an excellent forensic analysis technique when performing identification of identical and static files (e.g., detecting illicit photographs). There are, however, inherit limitations and challenges of present reference set systems when performing forensic analysis of application software. This research divides the identified problems into three distinct categories:

**Problem 1:** Creating application software reference sets.

**Problem 2:** Storing and sharing application software reference sets.

**Problem 3:** Correlating reference sets against target data sets.

The following three subsections discuss each of the classified research problems and highlight a variety of specific challenges when implementing application software reference sets to perform automated digital forensic analysis.

### 4.2.1.1 Problem 1: Creating Application Software Reference Sets

The reviewed literature aided in identifying a variety of problems surrounding the creation of application software reference sets. These problems are shared by researchers (who attempt to advance the technology) and practitioners (who perform actual digital investigations). The following list highlights the current problems:

- **Reverse engineering techniques lack standardisation:** Reverse engineering is essential to identify application software artifacts. However, Garfinkel (2010) states that researchers lack a standardised and systematic approach to reverse engineering compounded by the absence of a standard set of tools, minimal tool automation and results that are unable to be shared between parties. This culminates in research being a standalone endeavour with minimal advances in the underlying technologies used.

- **Reverse engineering problems:** Previous research has implemented a variety of methods, techniques and tools to perform reverse engineering. However, in terms of creating application software reference sets a number of research challenges remain:

  - **Reverse engineering produces irrelevant results:** Previous research has discovered reverse engineering produces a high number of irrelevant digital artifacts caused by operating system noise (Seifert et al. 2007). Therefore, a created reference set will contain unrelated operating system files and Registry entries that are not uniquely associated with the application software being reverse engineered.
Reference set creation requires a manual decision making process: Due to the number of irrelevant results produced, a reference set then requires a manual decision making process to classify relevant and irrelevant digital artifacts. This is a time-consuming process, requires a high level of knowledge and introduces analyst bias into the reference set creation process.

- **Reference set generation time:** Modern applications are regularly updated which means that maintaining a reference set for every software version is becoming less feasible (Roussev, 2011). Research has attempted to solve this problem using more flexible hashing techniques such as similarity digests (Roussev & Quates, 2012) and small block forensics (Garfinkel et al., 2010). A more practical solution would be to improve the speed and simplicity of data collection to enable rapid reference set creation. This would also increase the potential for application code coverage; for example, collecting data and resultant system changes from conducting various application execution tasks such as creating a document, encrypting a file and securely deleting a file.

As reverse engineering techniques are not specifically designed to aid forensic analysis of application software, further research is needed to improve reference set creation in terms of the time taken, functionality to filter irrelevant results and to provide an automated creation method. Solving the identified research problems would provide a solution specifically designed to reverse engineer application software to aid real digital investigations.

### 4.2.1.2 Problem 2: Storing Application Software Reference Sets

The reviewed literature revealed a variety of problems surrounding storing and sharing application software reference sets. Again, the identified problems are shared by researchers and practitioners. The following list highlights these current problems:

- **No standardised data abstraction:** There currently exist a variety of forensic data abstractions that are commonly used in digital investigations. However, none have been specifically designed and implemented to address the requirements of reference sets for application software. A forensic data abstraction requires the functionality to store, distribute and automate processing of application specific digital artifacts.

- **Challenges incorporating multiple evidence sources:** Reference sets are primarily comprised of metadata that represent data files (Roussev, 2010). However, most Windows applications store system configuration information in the Windows Registry (Morgan, 2008). Registry information has yet to be incorporated in reference sets and forensic analysis of Registry entries from application software is a predominately manual task. This results in limited functionality for application software reference sets as they lack complete application software information. Furthermore, there are currently no methods to store multiple evidence sources in a single reference set document.

- **Challenges classifying known digital artifacts:** Present reference set systems are limited in the functionality provided to perform classification of digital artifacts from
application software. For example, matches returned do not provide information such as which application life cycle phase (e.g., install or execution) the original digital artifact is from. This information could provide evidence of what tasks the user has performed with an application. For example, if a user has just installed the application but never run it; or if a user has installed, run and then later uninstalled the software in an attempt to hide their activity.

Currently available data abstractions are not specifically designed to aid forensic analysis of application software. They lack the functionality to store multiple digital artifact types from different evidence sources. They also lack the functionality to retain application specific information useful to forensic analysis of application software; for example, classifying the digital artifact life cycle. These problems are caused by no standardised or widely accepted forensic data abstractions to store, distribute or automate processing of digital artifacts from application software.

4.2.1.3 Problem 3: Correlating Reference Sets Against Targets

The reviewed literature also showed a variety of problems surrounding correlating application software reference sets against target data sets. Again, the problems are shared by researchers and practitioners. The following list highlights the current problems that have been identified:

- **Challenges detecting fragile data types**: Present reference sets are primarily implemented using only data files and matching is performed using cryptographic hashing (e.g., MD5 or SHA1). Correlating data files using only hash values can limit discovery as only homologous files\(^1\) are identified (Kornblum, 2006). Hash based file identification fails to detect fragile data types. For example, application software usually includes various dynamic files (e.g., configuration or log files) which are frequently modified resulting in a variable hash value that cannot be used for digital artifact detection.

- **Challenges performing version detection**: Modern applications are subjected to regular patching and updates, which means maintaining a reference data set of cryptographic hash values for every software version is not feasible (Roussev, 2011). This problem is related to fragile data types, but has a different cause. An application reference set is constructed using: 1) A specific application version; and 2) A specific operating system version. Using reference sets against variable targets introduces the following challenges:
  - Difficulty detecting known digital artifacts from a **different application version** than that used to create the reference set.
  - Difficulty detecting known digital artifacts on a **different operating system version and architecture** than that used to create the reference set.

In order to address the outlined problems it is proposed that a reference set system is to be designed which is specifically tailored to automate forensic analysis of application soft-

\(^{1}\)Homologous files are completely identical files that share exactly the same sets of bits in the same order.
ware. More specifically, automated decision making of different digital artifacts types (e.g., file system and Registry entries) that are associated with application software is prescribed. Furthermore, automated processing and detection of digital artifacts against a target data set is specified. The three identified problems have been discussed and satisfy the first element in the DSRM process model: to identify the research problem and motivation.

4.2.2 Define the Objectives of a Solution

The second element in the DSRM process model specifies that the objectives of a solution are defined. Objectives are to be stated based on the definition of the problem area and knowledge of what is possible or feasible, including a description of how the proposed artifact is expected to provide a solution to unresolved problems (Peffers et al., 2007). The following subsections outline the objectives for this research project. The high-level research objective is specified including solution requirements and a high-level overview of the proposed system design. Following this, a selection of lower-level research objectives are specified, each are categorised based on the proposed system architecture.

4.2.2.1 High-Level Research Objective

The problems identified in Section 4.2.1 highlighted a number of challenges surrounding the implementation of reference sets and the potential to perform automated forensic analysis. This has led to development of the objective for this research project, that is, to automate the identification of application software presence on an investigation target. Furthermore, due to the digital forensic challenges of data volume and the number of targets requiring analysis, an efficient solution is paramount. This introduced the requirement of designing a solution that adheres to digital forensic triage requirements. Therefore, the high-level objective for this research project is:

**High-level Research Objective:** To enable effective, automated detection of relevant digital artifacts to identify application software presence on a target data set. The solution must adhere to digital forensic triage requirements including high system efficiency and initial forensic examination output

It is important to specify requirements on which to design and later evaluate the high-level research objective. The following list specifies a collection of requirements that the solution must meet to be considered successful. References to the prescribed design elements and evaluation measurements are outlined:

- **Effectiveness:** The system should be functionally capable of detecting a high number of relevant digital artifacts that are unique to an application, while also returning a low number of irrelevant digital artifacts. Ultimately, the system is proven to be effective if it has the ability to provide evidence that an application is present on an investigation target.
- **Automation:** The system should be completely automated with no requirement of human intervention during operation.
- **Efficiency:** The system should be highly efficient while maintaining functional capability to detect forensically interesting digital artifacts.
- **Digital forensic triage:** The system should adhere to triage requirements including performing a partial forensic examination and retaining the output for later in-depth forensic analysis. Additionally, the system should identify investigation targets which have specific application software of interest, therefore, prioritising in-depth analysis of forensically interesting targets.

The high-level research objective specifies for the design, implementation and evaluation of a system to perform automated forensic investigation, specifically, triage of application software. To aid in solving the high-level research objective a system architecture is now proposed. This will aid in dividing the primary objective into a selection of lower-level objectives which will each address a specific component of the system architecture.

A **reference set** represents known content (digital artifacts) that are uniquely associated with an application (e.g., data files, Registry entries). The research objective specifies an automated approach to reference set creation using **reverse engineering**. The reference set can then be used as an input and compared against a **target data set**. The two inputs should be automatically processed and **digital artifact matching** performed. The system should then produce forensic reports populated with known content matches. A match occurs when the same digital artifact resides in both the reference set and the target data set. These results are digital evidence of application usage in the target data set. The high-level research objective specifies for an effective and efficient solution. These terms are very important to evaluate the proposed system design, both are later defined and discussed in the evaluation specifications (see Section 4.3.4 and Section 4.3.5). Figure 4.2 displays a high-level overview of the proposed system design and associated architecture.

![High-level overview of the proposed system design architecture](image-source)
To aid in breaking down the high-level research objective a group of more specific lower-level research objectives should be outlined. Each individual component of the system architecture (as displayed in Figure 4.2) has been singled out and specific lower-level objectives associated with each component are outlined and discussed in the following subsections.

4.2.2.2 Reverse Engineering Objectives

The reverse engineering component of the system design involves identifying the digital artifacts that are uniquely associated with an application. Reverse engineering is required because the source code and/or extensive documentation is not readily available for most application software. However, reverse engineering techniques for application software forensics (and digital forensics in general) lack standardisation. Although there is a variety of methods, techniques and tools available, none are specifically designed for reverse engineering application software to aid automated forensic analysis. Furthermore, reverse engineering techniques have a collection of problems including the prevalence of irrelevant results which leads to a manual decision making process to determine digital artifacts of interest. In order to solve the current approaches used to reverse engineer application software the first lower-level research objective is:

**Lower-level Research Objective One:** To enable effective and efficient automated identification of relevant digital artifacts from application software

A solution to automate reverse engineering of application software would advance the fundamental technology used to create reference sets for application software. A specific focus on digital forensic requirements is essential, something that existing solutions lack. Automation is a key requirement, providing the functionality to remove the human element of the reverse engineering process. It removes the manual decision making requirement and will aid in automating reference set creation. Automation would also decrease reference set creation time while increasing potential code coverage of application software, resulting in a more detailed and useful reference set.

The output from the reverse engineering process is a reference set populated with unique digital artifacts associated with application software. A data abstraction is now required to store, distribute and automate processing of the results from reverse engineering.

4.2.2.3 Reference Set Objectives

The reference set component of the system architecture involves designing and implementing a data abstraction to store, distribute and automate processing of digital artifacts associated with application software. Previous research has conducted numerous investigations centred on forensic analysis of application software by examining digital artifacts (see Section 2.2.1). However, there is no standardised or widely accepted data abstraction available to store or distribute this valuable research output, thus making it accessible to other researchers or practitioners. Instead, forensic practitioners are required to investigate every type of application...
software, determine remnant digital artifacts from multiple evidence sources, and then con-
duct a manual investigation using a variety of software tools. This is a fundamental problem
when advancing the overall technology used in automating forensic analysis of application
software. This process could be solved by designing and implementing a suitable data ab-
straction for application software reference sets. Therefore, the second lower-level research
objective is:

**Lower-level Research Objective Two:** To design an effective data abstraction
for storing, distributing and automating the processing of different digital artifact
types from application software

Once a data abstraction has been designed and implemented the digital artifacts identified
from reverse engineering can be populated into the prescribed format. It is envisaged that
the reference set will be populated with corresponding digital artifact metadata (e.g., name,
size, location, hash value) from file system and Registry entries that are uniquely associated
with an application. Once a reference set is implemented using a suitable data abstraction it
can be populated with identified digital artifacts and correlated against a target data set.

### 4.2.2.4 Target Data Set Objectives

The third component of the proposed system design is the target data set. In a digital
investigation the target data set is a previously identified and collected digital evidence source;
for example, a bit-by-bit copy of a computer Hard Disk Drive (HDD). In order to provide the
ability to automate correlation between the reference set and the target data set both sources
should be stored in the same data representation. Since the reference set contains a metadata
representation of digital artifacts from application software, the target data set should also
be processed and converted to the same metadata representation. The metadata must adhere
to digital forensic requirements and accurately represent the target data set. The generated
metadata representation should also be stored and saved in a data abstraction that can be
reprocessed, which will save time if reprocessing is a future requirement. Therefore, the third
lower-level research objective is:

**Lower-level Research Objective Three:** To establish an effective and efficient
automated technique to generate a metadata representation of the target data set

The result from lower-level research objective three is an accurate and reliable metadata
representation of the target data set. The next component of the system design is to correlate
information between the reference set and target data set.

### 4.2.2.5 Digital Artifact Matching Objectives

The fourth component of the system design is performing digital artifact matching between
the reference set and target data set. Correlation of digital artifacts is a key component of
the system design as it automatically determines if there are matching entries that reside in
both sets. The outcome of this phase is a list of matched digital artifacts and evidence of software presence, or absence, on a target system. However, there are a variety of challenges when performing matching of application software artifacts.

As previously outlined (see Section 4.2.1.3), forensic analysis of application software using a reference set fails to detect fragile data types with variable content. Examples such as application configuration files, log files and some Registry values all have dynamic content which creates difficulties when performing matching; for example, configuration files cannot be matched using hash values as the content is continually modified. Registry values have similar properties and may store an application configuration setting; for example, if the application automatically checks for updated versions. Depending on the option selected by the user, different value data could be stored. Digital artifact types (e.g., data files, folders, Registry keys and values) all have different metadata property values that are able to be correlated to determine matches. Therefore, the forth lower-level research objective is:

**Lower-level Research Objective Four:** To determine effective and efficient automated methods to correctly detect relevant digital artifacts from a target data set using the known content from a reference set

As previously discussed (see Section 4.2.1.3), reference sets for application software are created with a specific application version on a specific host operating system with a specific version and architecture. All of these variables create a reference set with different information. The difference may be vast depending on the variables involved and the effect it has on the resultant reference set. Creating a reference set for every version of an application is not feasible. It is proposed that techniques can be designed and implemented that can aid performing detection of digital artifacts from different application versions and/or operating systems. Therefore, lower-level research objectives five and six are:

**Lower-level Research Objective Five:** To determine effective and efficient techniques to perform detection of digital artifacts created by different application software versions

**Lower-level Research Objective Six:** To determine effective and efficient techniques to perform detection of application software artifacts on different Windows operating system versions

The matching of digital artifacts is an essential objective to be solved in the design of an advanced reference set system tailored for the automated detection of application software artifacts. The proposed lower-level research objectives four, five and six specify the digital artifact matching goals to be achieved. The overall objective of the research is to design and implement a system to perform automated detection of digital artifacts from application software. The objectives of a solution have therefore been proposed and various lower-level objectives specified for each system component.
4.2.3 Design and Development

The third element in the DSRM process model specifies the design and development of the system. The design phase attempts to solve the research problem based on the set down research objectives. Nunamaker and Chen (1990) state that design is one of the most important aspects in the Design Science process and involves the application of relevant scientific and technical knowledge to synthesise the proposed solution. The result of the design and development phase is the creation of an innovative and purposeful artifact.

A design research artifact can be any designed object in which a research contribution is embedded in the design. This activity includes determining the artifact’s desired functionality and its architecture and then creating the actual artifact. Peffers et al. (2007, p. 55).

Hevner et al. (2004) are more specific, stating that design science research must produce an artifact in the form of a construct, a model, a method, or an instantiation. Overall, the goal of the design and development element is to create a purposeful artifact that solves the identified problem. The purposeful artifact of this research is the design and development of a system to perform effective and efficient automated forensic analysis of application software using a reference set populated with known content. A full chapter is dedicated to system design (see Chapter 5). Following design and development of the artifact, is demonstrating that the system can solve the identified problems.

4.2.4 Demonstration

The fourth element of the DSRM process model is demonstration of the designed and developed design science artifact, in this research the artifact is a system. Nunamaker et al. (1990) summarise the demonstration of the artifact by simply building the system. Peffers et al. (2007) elaborate on this premise, stating that demonstration of the artifact involves proving that the idea works by solving one or more instances of the problem and may entail experimentation, simulation, case study, proof, or other appropriate activities. Demonstration of the artifact is an important element in design science research as the proposed solution cannot always be mathematically proven or empirically tested.

Researchers have to develop a system to demonstrate the validity of the solution, based on the suggested new methods, techniques, or design. Once the system has been built, researchers can study its performance and the phenomena related to its use to gain insights into the research problem. Nunamaker & Chen, 1990 p. 635).

Demonstration aims to verify the functionality and the utility of the artifact in a laboratory context. It is proposed that the resultant artifact from this research be demonstrated by the instantiation of the system design via software development. However, “building a system in and of itself does not constitute research” (Nunamaker et al. 1990 p. 103). Therefore, the demonstration of an artifact is inherently linked with the subsequent evaluation phase.
“The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods” (Hevner et al., 2004, p. 83). The implemented software tool will be demonstrated when tested against specifically authored data sets with known content to prove functionality. The tool can then be subsequently evaluated to provide richer details of the resultant solution.

4.2.5 Evaluation

“The essence of IS as a design science lies in the scientific evaluation of artifacts” (Hevner & Chatterjee, 2010, p. 55). Therefore, the designed and demonstrated artifact must be evaluated to determine the effectiveness and efficiency of the proposed solution. The evaluation of the artifact should also validate the functionality and utility in a real world context. Peffers et al. (2007) state that the evaluation phase includes observing and measuring how well the designed artifact supports a solution by comparing the research objectives to the observed results using relevant metrics and analysis techniques.

Evaluation of a designed IT artifact requires the definition of appropriate metrics and possibly the gathering and analysis of appropriate data. IT artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes (Hevner et al., 2004, p. 84).

Evaluation of the designed and demonstrated artifact is not a one-way operation. Peffers et al. (2007) state that researchers can iterate back to the design and development phase in an attempt to improve the overall quality of the artifact and then re-evaluate the new design. Nunamaker et al. (1990) agree, stating that experience gained from design, development and evaluation of a system usually leads to further refinement. This research prescribes that the evaluation phase should appraise the system in terms of effectiveness and efficiency. Therefore, the primary goal of the evaluation phase is to observe how effective and efficient the system design is at providing a solution to the problem area. The high-level research objective specifies the effective and efficient detection of digital artifacts from application software using reference sets. Each lower-level research objective also specifies for effectiveness and/or efficiency to aid in design, development and evaluation of the proposed system. The following subsections outline the requirements of the effectiveness and efficiency measurements.

4.2.5.1 System Effectiveness

The effectiveness of the proposed system can be determined by using measurements to aid in quantifying the overall success of experimental testing. Each measurement informs on different aspects of overall effectiveness as follows:

- **Retrieval** rate when performing digital artifact identification or detection.
- **Relevance** rate when performing digital artifact identification or detection.
- **Accuracy** level when performing digital artifact identification or detection.
Using the prescribed measurements, system effectiveness can be described as:

**Effective:** The artifact is deemed effective if the functionality of the system design can provide a high level of retrieval, relevance and accuracy when performing automated identification and detection of digital artifacts from application software. The system is deemed effective if: 1) It is more successful when compared to previous methods (i.e., implementation of reference sets using hash values for known file identification); and 2) Also meets a certain threshold of success. To achieve this, measurements can be quantified:

- Higher number of correctly detected digital artifacts.
- Lower number of unrelated detected digital artifacts.

4.2.5.2 System Efficiency

The efficiency of the proposed system can be determined by ascertaining the processing speed (computational efficiency) of the implemented system.

**Efficient:** The artifact is deemed efficient if the automated identification and detection of digital artifacts from application software maintains computational efficiency and/or can be judged to be improved when compared to previous similar methods.

The evaluation phase (see Chapter 7) is an important component in this research that will provide the ability to assess the overall success of the design science artifact to solve the research objectives.

4.2.6 Communication

The final element in the DSRM process model is communication of the conducted research to relay the resulting knowledge. Peffers et al. (2007) state that communication of the research should target relevant audiences including researchers and practising professionals and involves: 1) Communication of the problem and its importance; 2) Communication of the designed artifact including its usefulness, novelty, rigour of design and the effectiveness. Potential communication avenues include peer-reviewed scholarly publications in journals and conferences as well as technical reports.

“Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences” (Hevner et al. 2004, p. 83). Technology-orientated audiences require adequate detail to construct (implement) the artifact (system) to allow for future extension, development and evaluation. “This establishes repeatability of the research project and builds the knowledge base for further research extensions” (Hevner et al. 2004, p. 83). In contrast, management-orientated audiences require details regarding the costs needed to construct (or purchase) and use the artifact in their organisation. Management-orientated audiences also expect evidence of the usefulness to determine the potential economic and production benefits.
This research is in the form of a PhD thesis and as such a large component of the material will be communicated in this document. The research will also be communicated via research papers in academic peer-reviewed journals and conferences.

The aim of the research is to construct the proposed artifact in a system design and then implement the design into a proof of concept software solution, or computer forensic tool. The development of software creates additional avenues of research communication by: 1) Releasing software source code to provide complete technical details of the methods and functionality of the system design; and 2) Additional software documentation including computer forensic tool operation and functionality.

4.3 Experimental Testing Method

Experimental testing is a significant aspect in the scientific method. The following subsections outline various components of the experimental testing method. The data sets used for experimental testing are specified including tailored known, publicly available and real world data sets. The remainder of the section outlines the effectiveness and efficiency metrics that will be used to determine the overall system performance to aid system evaluation.

4.3.1 Operating System Selection

The experimental testing method requires the selection of a specific operating system type to be implemented throughout the testing process. A single operating system is prescribed so that a manageable research scope can be established. This research project has selected the Microsoft Windows operating system due to its popularity and prevalence in digital investigations. Although just one operating system is chosen, it is expected that multiple Microsoft Windows versions (e.g., Windows 7, Windows Vista and Windows XP) will be involved in experimental testing. However, the proposal is that Microsoft Windows 7 will be the primary testing operating system as, according to NetApplications (2015), it controls approximately 56% of the total desktop operating system market share.

4.3.2 Application Software Selection

The experimental testing method requires a collection of applications to create reference sets as well as to create known data sets for system demonstration. Reference sets have two primary uses: 1) To identify relevant evidence of application software usage; and 2) To perform data reduction by filtering irrelevant content. The reference sets proposed in this research are to automate the identification of relevant content. Therefore, the selection of application software should incorporate tools that are forensically interesting, that is, the existence of application software on a suspects system is evidence of possible malicious application usage.

Anti-forensic tools have been selected as the application software of choice as they present a relevant and interesting case study. Since anti-forensic tools can be maliciously used, a reference set of an anti-forensic tool would be used to detect the presence, or absence, of that tool on a suspects system. Three anti-forensic tools have been selected for testing: 1)
CCleaner; 2) Eraser; and 3) TrueCrypt. Each of the selected anti-forensic tools are briefly outlined in the following subsections.

4.3.2.1 CCleaner

CCleaner, developed by Piriform, is a system cleaning and privacy tool with the functionality to remove unused files and personal information from a computer system; for example, Internet browsing history [Piriform, 2015]. CCleaner is classified in the anti-forensic technique of evidence destruction. It is a freeware (or freemium) application which has no associated source code available. CCleaner provides support to securely remove a wide variety of Microsoft Windows artifacts including the Recycling Bin contents, recent documents, temporary files, log files and clipboard contents, while also having a built-in Windows Registry cleaner. Furthermore, CCleaner provides support to remove Internet artifacts including temporary files, cookies and web browser history from popular web browsers including Internet Explorer, Mozilla Firefox, Google Chrome and Apple Safari. When performing file deletion to prevent any data recovery, CCleaner has secure deletion capability to enable data overwriting [Velupillai & Mokhonoana, 2008]. However, CCleaner has to be explicitly set by the user to perform secure deletion as it is not turned on by default [Piriform, 2016].

4.3.2.2 Eraser

Eraser, developed by Heidi Computers Ltd, is a secure data removal tool designed for the Microsoft Windows operating system platform [Eraser Team, 2015]. Eraser is classified in the anti-forensic technique of evidence destruction. It supports Microsoft Windows XP (with Service Pack 3), Vista, 7, 8, 10 and Windows Server 2003, Server 2008 and 2012. Eraser is an open-source application released under the GNU General Public License (GPL) version 3.0 and distributed as either compiled binaries[2] or source code[3]. It provides a variety of advanced functionality to perform secure deletion of file system entries and of unused (unallocated) disk space, overwriting any previously deleted files. A built-in scheduler allows users to create scheduled tasks which perform secure deletion of specified targets. In total, Eraser provides 11 different secure deletion algorithms.

4.3.2.3 TrueCrypt

“TrueCrypt is an open-source program that allows the user to create encrypted virtual disks and to encrypt not only entire system partitions, but also the entire system drive containing the operating system” [Forte, 2009]. TrueCrypt performs data encryption which is classified in the anti-forensic technique of evidence hiding. The TrueCrypt project has had a rich and colourful history. TrueCrypt version 1.0 was released by a group of anonymous developers in 2004 and was based on another encryption application called Encryption for the Masses, otherwise known as E4M [van Bergen, 2015]. TrueCrypt has been widely used

since its release. However, according to Leyden (2014), the TrueCrypt project closed down on the 28 May 2014. A warning notice was displayed on the project website stating that the software is not secure and may contain unfixed security issues. At the same time, TrueCrypt version 7.2 was released which only provides the functionality to decrypt previously created TrueCrypt volumes. Although the TrueCrypt project is now defunct, the tool was selected for testing prior to the closure of project. Since the shutdown a number of dedicated repositories have been established to provide users with access to the software. van Bergen (2015) maintains the DrWhax TrueCrypt Archive, a GitHub repository which retains almost every TrueCrypt version. Steve Gibson (2016), a well-known security researcher, released the TrueCrypt Final Release Repository which hosts TrueCrypt version 7.1a. The Open Crypto Audit Project (2014) also host a repository for TrueCrypt version 7.1a.

TrueCrypt still has widespread popularity and has been downloaded almost 700,000 times from Gibson’s repository. Of this number, TrueCrypt version 7.1a for Microsoft Windows 32/64-bit was downloaded over 270,000 times. Since the TrueCrypt project website is no longer available, the software for this research was sourced from the TrueCrypt Final Release Repository (Gibson, 2016). In order to ensure a reliable version of TrueCrypt was sourced for testing, the MD5 hash value of the downloaded executable (TrueCrypt Setup 7.1a.exe) was compared to other executables hosted on the previously mentioned TrueCrypt repositories.

4.3.3 Data Set Selection

The experimental testing method requires a selection of data sets for the digital artifact matching component of the system architecture. The data sets discussed in this section will be used as the target data set input into the system architecture (see Figure 4.2). Three data sets have been chosen which include known, publicly available and real world data. Each proposed data set has a specific purpose to aid in answering the research objectives. Therefore, each data set will be executed in different phases of the DSRM process model adopted for this research project. The following subsections outline the proposed data sets to be used.

4.3.3.1 Known Content Data Sets

The first data set proposed for experimental testing is to aid in system demonstration (see Section 4.2.4). Demonstration involves proving the designed system has the prescribed functionality and can solve the research problem/objective. It is proposed that a data set with known content is to be authored in a controlled laboratory environment. Authoring a data set involves creating a realistic computer system and then populating it with known content. Since this research focusses on application software, the known content is generated by performing tasks with applications; for example, installing an application or executing a specific

---

4 See: http://www.truecrypt.org/
5 See: sourceforge.net/projects/truecrypt/files/TrueCrypt/TrueCrypt-7.2.exe
6 The MD5 hash value of TrueCrypt version used in testing is 7a23ac83a0856c352025a6f7c9cc1526
task with an application.

Experimental testing must be performed in a documented, reliable and secure testing environment. Therefore, it is proposed that Virtual Machines (VMs) will be implemented. VMs provide an excellent platform to perform experimental testing as they can easily be put into use compared to a physical system. This is important as it saves time, especially when it is necessary to conducted multiple experimental runs. VMs also provide a containment free testing environment.

The simulated computer system needs to be populated with application software data so therefore a life cycle is recreated to generate known content (see Figure 2.3). The selected application life cycle phases are to be replicated from the National Institute of Standard and Technology (NIST) Diskprint project \cite{laamanen2014} as it is similar to this research. Selection of the same life cycle phases allows for a comparison to be made between the two projects. This will provide a baseline (the Diskprint project) on which to compare and evaluate the outcomes of this research. The following life cycle phases are to be recreated for each tested application:

1) **Install** the application (using default options)
2) **Open** the application
3) **Close** the application
4) **Uninstall** the application
5) **Reboot** the system

Each application life cycle phase will be recreated on the specified VM test system and the virtual disk will be subjected to a bit-by-bit forensic copy. Therefore, each application will have a total of five (5) separate forensic images (evidence files) to represent each application life cycle phase. The known data set provides a selection of testing scenarios on which to perform system demonstration and verify the implemented system in terms of functionality and ability to solve one or more parts of the research objectives.

### 4.3.3.2 Publicly Available Data Sets

The second data set proposed for experimental testing is a publicly available data set designed specifically for digital forensic research. \cite{garfinkel2009} states that standardised and publicly available digital forensic corpora (data sets) are essential to enable research reproducibility. Therefore, the public data set should be openly distributed and freely available for public download from the Internet.

The public data set for this project is to be implemented when performance evaluation of the system design is carried out (see Section 4.2.5). It will offer the capability for other researchers, forensic practitioners, software and tool developers to test, evaluate, verify and/or build on the findings presented in this research.

The proposed public data set is the **M57-Patents scenario**, authored and distributed by the Digital Corpora project\footnote{See: \url{http://digitalcorpora.org/}}. “The M57-Patents scenario is a multi-modal corpus consisting of...
hard drive images, RAM images, network captures, and images from other devices typically found in forensics investigations such as USB drives and cellphones” (Woods et al., 2011, p. 123). Since this research focuses on forensic analysis of the Microsoft Windows desktop operating system, the data of interest are the hard drive images (a physical forensic image of the original hard drive). There are a total of 79 hard drive images in the M57-Patents corpora which run different Microsoft operating system versions including Windows XP and Windows Vista (Woods et al., 2011). Although the M57-Patents scenario contains information derived from copyrighted materials it is available for research, education, training and the production of educational materials.

There are two additional reasons why the M57-Patents scenario was specifically selected for experimental testing. Firstly, the M57-Patents scenario is known to have at least one of the applications selected for experimental testing. Roussev and Quates (2012) discovered that, using the sdhash tool, a user (Jo) had used the TrueCrypt software. Therefore, at least one of the anti-forensic tools selected for testing is present on the data set. Secondly, the Windows operating system versions present in the M57-Patents scenario are different from the version used to create the application software reference sets (Windows 7 versus Windows XP and Vista). Therefore, the data set provides the capability to test on a different operating system version. This aids in evaluating the system design, specifically lower-level research objective numbers five and six.

The M57-Patents scenario has been used extensively in previous published digital forensic research in a variety of research topics. Roussev and Quates (2012) used the M57-Patents scenario as a case study to demonstrate the utility of similarity digests, specifically the sdhash tool, to perform rapid content-based forensic triage to identify relevant evidence from disk images, RAM snapshots and network traces. Nelson (2012) used the M57-Patents scenario to perform a longitudinal time analysis of Windows Registry entries by investigating modified (last write) timestamp values in Registry keys. N. Beebe and Liu (2014) used the M57-Patents scenario to evaluate a relevancy ranking algorithm for string search output. Garfinkel and McCarrin (2015) used the M57-Patents scenario to evaluate a hash-based file carving approach using sector hashing and the hashdb tool. These examples of previous research using the M57-Patents scenario illustrate the popularity and usefulness in adopting publicly available data sets for digital forensic research.

4.3.3.3 Real World Data Sets

The third data set selected for experimental testing is a real world data set comprised of used (second-hand) hard drives. Performing experimental testing on real world data sets for digital forensic analysis development and evaluation is advantageous due to data diversity and unpredictability, thereby providing more robust research findings (Garfinkel et al., 2009).

---

8 The M57-Patents scenario contains copyrighted binaries from Microsoft Windows XP and Windows Vista operating systems. In order to remove potential copyright issues the bitstream of any executable and library files were altered before distribution. See Woods et al. (2011), Section 9.4 for additional information.

9 See: https://github.com/sdhash/sdhash and http://roussev.net/sdhash/sdhash.html

10 See: https://github.com/NPS-DEEP/hashdb/wiki
The specific goal of a real world data set is to subject the solution developed in this research to unpredictable and diverse data including different Microsoft Windows operating system versions, a wide range of installed application software and a wide variety of file formats. The purpose of this diverse data is to identify system design issues and general programming errors. The end result is validation of the functionality for the implemented software.

The real world data set is to be compiled from two sources. Firstly, a total of 100 second-hand hard drives are to be sourced from previous research conducted by Roberts (2013) at the Department of Information Science, University of Otago (the same university as the author). Secondly, it is proposed that the author will purchase additional second-hand hard drives specifically for this research sourced from online auction websites.

The second-hand hard drives are to be physically imaged using accepted digital forensic techniques. A write blocker will be used to ensure that the original hard drive is not altered when attached to a host computer. Each hard drive will then be forensically collected and a bit-by-bit copy of the source hard drive produced. This process results in a forensic disk image (evidence file) that represents the exact contents of the original hard drive. The forensic disk images will be used as input to the digital artifact matching component of the system design.

4.3.3.4 Data Set Summary

As outlined above, three data sets have been proposed: 1) Known data set; 2) Publicly available data set; and 3) Real world data set. Each data set will be used as the target data set in the proposed system design (see Figure 4.2). The known data set provides the ability to demonstrate the functionality of the system implementation. The public data set provides research reproducibility and different operating system versions to evaluate the system implementation. Finally, the real world data set provides an unknown and complex investigation target on which to further evaluate the system implementation.

4.3.4 Effectiveness Metrics

The purpose of the proposed system is to automate the identification and detection of digital artifacts from application software. As such, the system design has a similar function to Information Retrieval (IR) systems. “Information retrieval is a discipline that deals with the representation, storage, organization, and access to information items. The goal of information retrieval is to obtain information that might be useful or relevant to the user” (Ceri et al., 2013, p. 3). IR has previously been used for a large number of systems and applications. A traditional example of an IR system is a library card catalogue system. However, the example of Internet search engines is arguably the most widespread and important IR system. Search engines operate by taking a search query and returning relevant web page links.

The system design in this research aims to perform IR by automatically detecting digital artifacts from a target data set and returning the results (matched digital artifacts) to the investigator. The effectiveness of the implemented system design can be measured based on the capability to correctly detect and report digital artifacts. IR metrics can therefore be used to determine the effectiveness of the system.
4.3.4.1 Information Retrieval Metrics

“When IR systems return unordered results, they can be evaluated appropriately in terms of precision and recall” (Ceri et al., 2013, p. 7). Since the actual order of returned results is not an issue in determining the effectiveness of the proposed system, the implemented framework can be measured using the standard IR metrics of precision and recall. According to Ceri et al. (2013), precision \( (P) \) is the fraction of retrieved documents that are deemed relevant based on the specified query and provides a measure of soundness of the system. Equation (4.1) displays the formula used to determine precision.

\[
\text{precision}(P) = \frac{\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}}{\{\text{retrieved documents}\}} \tag{4.1}
\]

The intersection (\( \cap \)) of relevant documents and retrieved documents is determined and divided by the total number of retrieved documents. In the experimental testing the documents are digital artifacts (e.g., data files, Registry entries). Relevant documents are digital artifacts that are detected by the system when analysing the target data set; for example, a data file from the application profile that is detected in the target data set, regardless of if they are actually in the application profile. Retrieved documents are all the digital artifacts that were detected in the target data set. Relevant and retrieved documents (detected digital artifacts) need to be classified and correctly grouped.

The results from an experiment are classified based on a coincidence matrix (later discussed in subsection 4.3.4.2). The classifiers True Positive \( (tp) \), True Negative \( (tn) \), False Positive \( (fp) \) and False Negative \( (fn) \) are specified and each digital artifact assigned to a single classification. Equation (4.2) displays the precision metric using the specified document classifiers. The \( tp \) values are the intersection (\( \cap \)) of relevant documents from retrieved documents, while \( tp + fn \) make up the entirety of the retrieved documents.

\[
\text{precision}(P) = \frac{tp}{tp + fp} \tag{4.2}
\]

The precision metric does not incorporate the total number of documents that are deemed relevant. This is an important measurement to determine the number of digital artifacts detected by the implemented system. Therefore, the recall metric was also used to determine system effectiveness. According to Ceri et al. (2013), recall \( (R) \) determines the fraction of explicitly relevant documents that are retrieved. Recall provides a measure of the completeness of the system design. Equation (4.3) displays the recall formula, while Equation (4.4) displays the recall metric using the specified document classifiers.

\[
\text{recall}(R) = \frac{\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}}{\{\text{relevant documents}\}} \tag{4.3}
\]

\[
\text{recall}(R) = \frac{tp}{tp + fn} \tag{4.4}
\]

“As precision and recall have different advantages and disadvantages, a single balanced IR
evaluation measure has been introduced as a way to mediate between the two components” (Ceri et al., 2013). This measurement is known as \( F\)–measure, displayed in Equation 4.5:

\[
F_{\beta} = \frac{(1 + \beta^2) \times P \times R}{(\beta^2 \times P) + R} \tag{4.5}
\]

The \( F\)–measure (or \( F\)–score) metric considers both the precision and recall scores to provide a combined average measurement. Equation 4.5 displays the \( F\)–measure metric with a undefined \( \beta \) value. The \( \beta \) value can be manipulated to derive a \( F\)–measure score which focuses more on either recall or precision depending on the requirements of testing. According to Ceri et al. (2013), \( F_1 \)–measure is the harmonic mean of the precision and recall scores where \( \beta \) equals 1, thus, giving an equal weight to both scores. In this research the \( F_1 \)–measure metric is prescribed and is displayed in Equation 4.6:

\[
F_1 = \frac{(1 + 1^2) \times P \times R}{(1^2 \times P) + R} \tag{4.6}
\]

The final effectiveness metric is \textit{accuracy}, also known as the Rand Index or Rand Accuracy. “The overall accuracy of a classifier is estimated by dividing the total correctly classified positives and negatives by the total number of samples” (Olson & Delen, 2008, p. 138). Accuracy determines the total number of true results (true positive and true negative) against the total number of cases examined. The accuracy metric is displayed in Equation 4.7:

\[
\text{accuracy}(A) = \frac{tp + tn}{tp + tn + fp + fn} \tag{4.7}
\]

### 4.3.4.2 Digital Artifact Classification

In order to calculate the prescribed IR metrics the \textit{documents} (digital artifacts) that are returned by the system require classification. “In classification problems, the primary source of performance measurements is a coincidence matrix” (Olson & Delen, 2008, p. 138). Table 4.1 displays a coincidence matrix for a two-class classification problem.

<table>
<thead>
<tr>
<th>Actual Class</th>
<th>\textit{positive}</th>
<th>\textit{negative}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{predicted} \textit{positive}</td>
<td>true positive (( tp ))</td>
<td>false positive (( fp ))</td>
</tr>
<tr>
<td>\textit{class} \textit{negative}</td>
<td>false negative (( fn ))</td>
<td>true negative (( tn ))</td>
</tr>
</tbody>
</table>
The results from an experiment are classified based on the coincidence matrix. Each digital artifact (e.g., data file, Registry entry) can be classified as one of four possibilities: 1) True positive ($tp$); 2) True negative ($tn$); 3) False Positive ($fp$); and 4) False negative ($fn$). A descriptive method can be used to perform digital artifact classification based on the coincidence matrix. Each of the four available classifiers are described as:

- **True positive** ($tp$) is a digital artifact that is relevant and correctly detected
- **True negative** ($tn$) is a digital artifact that is irrelevant and not detected
- **False positive** ($fp$) is a digital artifact that is irrelevant and incorrectly detected
- **False negative** ($fn$) is a digital artifact that is relevant and not detected

After each digital artifact has been correctly classified, the prescribed effectiveness metrics can be conducted as the first measure in determining system performance.

### 4.3.5 Efficiency Metrics

Two different types of efficiency metrics are proposed to further evaluate system performance. Computational efficiency determines the functional data processing performance of the system. Two computational performance measurements are to be used: 1) Absolute speed, which measures the complete analysis clock time; and 2) Relative speed, which determines the average rate that the tool can process evidence compared to the rate at which data can be read from the source device \cite{Ayers2009}. Equation (4.8) displays the formula to determine the absolute speed. Equation (4.9) displays the formula to determine average processing speed in megabytes per second (MB/s) which is used to determine the relative speed as shown in Equation (4.10).

\[
\begin{align*}
\text{absolute speed} \text{ (time)} & = \text{total elapsed processing time} \\
\text{average processing speed} \text{ (MB/s)} & = \frac{\text{data set size} \text{ (MB)}}{\text{absolute speed} \text{ (seconds)}} \\
\text{relative speed} & = \frac{\text{average processing speed} \text{ (MB/s)}}{\text{drive read speed} \text{ (MB/s)}}
\end{align*}
\]

In order to calculate the relative speed metric two variables are required. The *average processing speed* is first determined by calculating the rate at which the system can process input data (see Equation (4.9)). The *drive read speed* also needs to be ascertained on the testing system. This can be achieved by performing a read speed test on the hard drive using a variety of hard drive benchmark utilities. It is proposed that multiple runs should be performed when determining drive read speed so that the average can be used to determine the input to the relative speed metric.

The purpose of the relative speed metric is to provide results that measure the average rate that a tool can analyse data compared to the rate that data can be read from a target device. Given that the drive read speed is incorporated into the metric, the metric provides
an indication of performance without intricate details of the testing system. Furthermore, the relative speed metric is directly comparable to results from similar research efforts where the testing system is inevitably different.

To ensure comparable computational efficiency results, all testing will be conducted on the same computer hardware: Intel Core i5-3570K CPU with 8 GB RAM and running Microsoft Windows 7. The Hard Disk Drive (HDD) used for testing is a Western Digital Green 2.0 TB SATA 3 desktop 3.5-inch hard drive.

4.4 Ethical Approval

According to [University of Otago (2016)], any research that involves human participants must obtain ethical approval and comply with the University’s ethics policy. The only component of this research that involves human participants is the use of real-world data sets to perform system evaluation (see Section 4.3.3.3). The data set is to be formed using a variety of sources of second-hand (used) hard drives that will likely have personal information from the previous owner. A collection of hard drives will be sourced from [Roberts (2013)] who conducted similar research in the same department as the author. Additionally, a collection of additional second-hand hard drives will be sourced from online Internet auction web sites in order to increase the size of the data set. This research project was granted ethical approval before commencing. Additional documentation regarding ethical approval is available in Appendix A.

4.5 Conclusion

Chapter 4 has specified the selection of Design Science as the most appropriate research methodology for this research project. The DSRM process model is a fitting guide for Information Systems research and meets the requirements of this project to create a purposeful artifact. Research problems were highlighted which include specific challenges centred on creating, sharing and correlating reference sets for application software. A high-level research objective was specified with accompanying lower-level research objectives set out to aid in obtaining a functional solution. From this a system design architecture was formulated. Thereafter, an experimental testing method involved the selection of an operating system platform, application software and data sets to use during testing, demonstration and evaluation. Finally, testing metrics to measure system effectiveness and efficiency were prepared, a vital culmination of the process model which will later aid in evaluation of overall system performance. The following chapter, System Design, solely covers the third element on the DSRM process model; that is, the design and development of the research artifact.
The Design Science Research Methodology (DSRM) process model serves as a functional framework for conducting this research. The third element of the DSRM process model centres on the designing of an artifact. According to Peffers et al. (2007), this involves determining the desired functional requirements of the artifact, specifying the architecture, then creating the actual artifact. The design must contribute to the achievement and ultimately satisfy the objectives of the solution; that is, in this research, the effective and efficient detection of digital artifacts from application software using automated reference set creation followed by automated correlation against a target data set. The artifact created in this research is, thus, in the form of a system design and the subsequent implementation of that design.

Chapter 5 specifies the design of the system to advance the automated detection of digital artifacts from application software. This research uses anti-forensic tools as the application software of interest. The system encompasses two main stages: 1) Design and implementation of a system to create a reference set to aid detection of application software; and 2) Matching the reference set against a target data set. To achieve this an overview of the system architecture is first described. The actual artifact (system design) is specified based on four components of the system architecture which includes: 1) Identifying digital artifacts to populate a reference set; 2) Development of a data abstraction for the reference set; 3) Processing the target data set; and 4) Performing digital artifact matching between the reference set and target data set. Each component has design requirements specified, is designed based on the requirements and finally implemented in the form of computer software.
5.1 System Design Overview

The overall high-level system design was established in the research methodology based on the research objectives (see Section 4.2.3 and Figure 4.2). The system design is centred around the concept of creating a reference set of known digital artifacts uniquely associated with a specific application. Reference sets for the forensic analysis of application software have a variety of naming conventions: Application profile, application footprint, application fingerprint, application signature and hash set. For reasons of clarity, the term used in this research is application profile. Figure 5.1 displays a high-level overview of the system architecture with four main components (displayed with corresponding section numbers).

Stage one of the system design is the creation of an application profile, starting with the input of an application software; for example, Figure 5.1 shows four applications: 1) Google Chrome web browser; 2) Mozilla Firefox web browser; 3) TrueCrypt disk encryption tool; and 4) CCleaner privacy suite tool. An application is subjected to reverse engineering which aids in performing digital artifact identification; for example, determining the file system entries and Windows Registry entries that are uniquely associated with that application. The identified digital artifacts are populated into an application profile data abstraction. The application profile itself is a reference set document that contains detailed metadata which represents identified digital artifacts and provides the functionality to store, distribute and automate processing of digital artifacts.

Stage two of the system design is digital artifact matching, that is, matching the reference set against the target data set. Two inputs are required for matching to take place: 1) the application profile; and 2) the target data set. Every digital investigation has at least one target data set; for example, a forensic image of a hard disk. However, before matching can be performed the target data needs to undergo target data set processing by generating metadata to represent the original evidence source (e.g., file system entries and Windows Registry entries). Digital artifact matching can then be performed by correlating...
digtial artifacts between the application profile and the target data set. The results are then presented as forensic reports to the investigator.

As displayed in Figure 5.1, the system design architecture is divided into four distinct components: 1) Digital artifact identification; 2) Application profile data abstraction; 3) Target data set processing; and 4) Digital artifact matching. The following sections outline the design requirements and design processes for each component and the subsequent implementation of the design as functional computer software.

5.2 Digital Artifact Identification

The first component in the system architecture is to reverse engineer application software to identify digital artifacts uniquely associated with an application. The identified digital artifacts are later populated into a specified application profile data abstraction. This section starts by specifying the system design requirements for performing digital artifact identification. The design process is then specified to achieve the desired functionality including a data collection and comparison method, as well as specific functionality to filter irrelevant digital artifacts and perform efficient file hashing. The design process is described with accompanying software development which culminates in a computer forensic tool to perform automated digital artifact identification.

5.2.1 Digital Artifact Identification Requirements

The identification of digital artifacts determines the additions, modifications, and/or deletions to the computer system that occur throughout the different phases of the application life cycle (see Section 2.4.1). On the Microsoft Windows operating system, file system entries (directories and data files) and Windows Registry entries (keys and values) encompass the majority of the digital artifacts of interest to aid digital investigation (see Section 2.2.2).

As the source code and/or extensive documentation is not readily available for most application software, reverse engineering is required to identify the digital artifacts associated with an application (see Section 2.4.2 and Section 3.2). It is proposed that differential forensic analysis (see Section 3.2.2) be implemented to perform reverse engineering of application software. The system design requires the solution of several problems which exist with previous approaches that have been used for creating application software reference data sets (see Section 3.2.3). To achieve this the following list of design requirements have been specified:

- Support for a portable Microsoft Windows tool to run on a live system
- Support for an efficient automated data collection procedure
- Support to determine file system and Windows Registry changes
- Functionality to blacklist irrelevant digital artifacts
- Functionality to include data file hashing (e.g., MD5 and SHA1)
- Functionality to report results to a suitable data abstraction
The specified requirements for a new tool can be met in part by a mix of previous approaches (see Section 3.2.2.1 and Section 3.2.2.2). The Regshot tool provides rapid data collection as it runs on a live system. However, Regshot has limited functionality in terms of reporting sufficient digital artifact metadata; for example, file hash values are not included in the differential analysis strategy or reported in the results. In contrast, the DFXML differencing tools (idifference.py and rdifference.py) provide exceptionally detailed metadata reports (in DFXML syntax), but lack rapid data collection due to the performance overhead of post-mortem analysis (using acquired forensic disk images to perform differential analysis).

It is proposed that the new solution is designed and implemented based on the strengths of these different approaches, while also removing the existing weaknesses. Similar to Regshot and the DFXML differencing tools, the new solution should support file system and Registry entry differencing and subsequent reporting of identified system-level changes. Additional extension is also required to fulfil the specified functionality outlined by the design requirements including blacklisting irrelevant digital artifacts, performing file hashing and reporting results to a suitable data abstraction. This leads to the design specification of a live portable tool that is effective at detecting system-level changes, runs efficiently on a live Microsoft Windows operating system and provides functionality to produce detailed and accurate reporting (specified later in Section 5.3).

5.2.2 Advanced Tool Design: LiveDiff

A new tool has been authored to perform digital artifact identification and subsequent population into a data abstraction. The resultant tool is named LiveDiff, after the reverse engineering technique that it implements: live differential forensic analysis. LiveDiff is based on the Regshot project (discussed in Section 3.2.2.1) and authored using the C programming language. Regshot is licensed under the terms of the GNU General Public License (version 2) as stated in the project ReadMe.txt file. This allows modification and redistribution of the source code. Since Regshot performs the same high-level functionality as required by the new proposed tool, it makes sense to implement and then develop from source code that has been previously authored and thoroughly tested.

LiveDiff has been implemented using the fileshot.c and regshot.c source code files from the Regshot project. The two source code files provide the functionality to capture and compare system snapshots (containing all file system and Registry entries) on a running Windows system. However, the source code taken from the Regshot project has required extensive modification and additional source code to achieve the desired functionality as specified by the system design requirements. Almost all of the programming functions and data structures taken from the Regshot source code have required some refinement but for the most part have been rewritten using the original source code as an example. The following subsections outline the core design of the system and LiveDiff tool including a data collection method, required tool functionality and an overview of software development.

\[\text{See: } \text{http://sourceforge.net/p/regshot/code/HEAD/tree/trunk/files/ReadMe.txt}\]
5.2.3 Data Collection Method

The proposed data collection method is to perform live differential forensic analysis by collecting and comparing system snapshots. Data collection is achieved by taking a system snapshot that contains all file system and Windows Registry entries. A snapshot is taken before and after a single action; for example, installing an application. Two snapshots are compared using differential forensic analysis and the system changes (in the form of digital artifacts) are then identified. A data collection procedure is required to collect and compare system snapshots throughout multiple application life cycle phases. Each data collection requirement is discussed in the following subsections as to how the method has been implemented in the LiveDiff tool.

5.2.3.1 Data Collection: System Snapshots

Each system snapshot consists of two different evidence sources: 1) Entries present in the local file system; and 2) Entries present in Windows Registry hive files. Both of these evidence sources have been established to be avenues of forensic interest (see Section 2.2.2).

File system data collection is achieved by performing a snapshot of the system drive (usually C:\). System information functions have been implemented to determine the correct system drive (volume) at run-time. To populate a system snapshot, data collection involves enumerating (processing) every file system entry and saving each entry and associated metadata in a data structure (FILECONTENT).

Windows Registry data collection is accomplished by enumerating (processing) every entry from the HKEY_LOCAL_MACHINE (HKLM) and HKEY_USERS (HKU) Registry hives. The selected Windows Registry data sources incorporate the SAM, SECURITY, SOFTWARE, SYSTEM and NTUSER.DAT Registry hive files. Metadata from each entry is saved in a data structure (KEYCONTENT for keys and VALUECONTENT for values). Table 5.1 displays the implemented data structures used to store digital artifact information. All the processed file system entries and Registry entries are stored in a single snapshot data structure, named SNAPSHOT. Each system snapshot stores a variety of child data structures to retain metadata values for every processed entry. All data structures were originally sourced from the Regshot project with modifications to include additional metadata information, consisting of: 1) Hash values for data files, 2) Modified (or last write time) for Registry keys; and 3) Additional timestamp information for all file system entries. Along with the properties listed in Table 5.1, each structure retains a pointer to the parent, sibling and/or sub-structures to provide access to related structures.

---

2Various differential analysis methods were investigated and trialled including virtual machine snapshot comparison and various Windows APIs functions to monitor file system and Registry changes. However, collection and comparison of system snapshots was selected due to potential future scalability of the method to different operating system types without reliance on specific platform dependent APIs.

3When LiveDiff is executed, the system drive is dynamically determined by calling the GetVolumePathName() on the result from the GetSystemWindowsDirectory() function.

4The HKLM and HKU Registry hives were selected for inclusion as they commonly contain application software configuration information (see Section 2.2.2.2). Inclusion of other Registry hive files have a low possibility of yielding results of interest while also increasing tool processing time.
<table>
<thead>
<tr>
<th>Digital artifact</th>
<th>Structure name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data file</td>
<td>FILECONTENT</td>
<td>File name, size, write time, access time, hash value and attribute</td>
</tr>
<tr>
<td>Directory</td>
<td>FILECONTENT</td>
<td>Directory name, size, write time, access time and attribute</td>
</tr>
<tr>
<td>Registry key</td>
<td>KEYCONTENT</td>
<td>Key name, modified time</td>
</tr>
<tr>
<td>Registry value</td>
<td>VALUECONTENT</td>
<td>Value name, type, data and data size</td>
</tr>
</tbody>
</table>

After collecting two snapshots (one before and after an action), a comparison is made to determine the system changes that have occurred using a differential analysis strategy.

### 5.2.3.2 Data Comparison: Differential Analysis Strategy

The proposed differential analysis strategy has been implemented based on the general differential forensic analysis strategy specified by [Garfinkel et al. (2012)](http://example.com), represented by the following notation: $\text{Snapshot}_1 \xrightarrow{R} \text{Snapshot}_2$. The formula is based on the previously discussed differential forensic analysis formula (see Section 3.2.2). $\text{Snapshot}_1$ is the system state before an application life cycle phase is conducted (e.g., application installation). $\text{Snapshot}_2$ is the system state after an application life cycle phase has been conducted. The two snapshots are compared and result in a list of created, modified and/or removed digital artifacts, represented by $R$. Figure 5.2 displays a visual representation of the snapshot method. $\text{Snapshot}_1$ is a fresh installation of Microsoft Windows, while $\text{Snapshot}_2$ is the same system after installation of the Mozilla Firefox application. $\text{Snapshot}_2$ clearly illustrates a collection of files, directories and Registry entries that have been created on the system after application installation.

![Figure 5.2: Visual representation of the snapshot differencing method (Image sources: Mozilla Firefox icon taken from The Mozilla Foundation (2013). Other images are public domain.)](http://example.com)
In order to perform comparison between the content of two SNAPSHOT structures, a differential analysis algorithm is required for both file system and Windows Registry entries. Algorithm 1 specifies the differential analysis strategy for file system entry correlation (FC refers to FILECONTENT structures). The algorithm has been developed based on the original Regshot comparison algorithm outlined in the CompareFiles() function from the fileshot.c source code file. The primary modification made to the original implementation was to include correlation of hash values (MD5 and/or SHA1) when comparing file system entries.

Algorithm 1 Differential analysis strategy for file system entries

1: procedure CompareFiles
2:   for each FILECONTENT in Snapshot1 (FC1) do
3:     for each FILECONTENT in Snapshot2 (FC2) do
4:       if (FC2 has previously been matched) then skip FC2
5:     end if
6:     if (FC1 type and name does not equal FC2 type and name) then skip FC2
7:     end if
8:     if FC1 is a file then
9:       if (all FC1 properties equal all FC2 properties) then FC2 is matched
10:     end if
11:     if (FC1 write time does not equal FC2 write time) then FC2 is changed
12:     end if
13:     if (FC1 size or hash does not equal FC2 size or hash) then FC2 is modified
14:     end if
15:     end if
16:     if FC1 is a directory then
17:       if (all FC1 properties equal all FC2 properties) then FC2 is matched
18:     end if
19:     end if
20:   end for
21:   if FC1 is not matched then FC1 must be deleted
22: end if
23: end for
24: for each FILECONTENT (FC2) in Snapshot2 do
25:   if FC2 has not been matched then FC2 must be new
26: end if
27: end for
28: end procedure

The differencing algorithm for Registry entries follows a similar differential analysis strategy. However, differencing of Registry values is performed in an embedded loop after two matching Registry keys are discovered. The algorithm is taken from the original Regshot comparison algorithm outlined in the CompareRegKeys() function. No major changes to the original Registry algorithm have been made. The only additions have been to perform an additional check to determine modified Registry keys based on the modified (last write time) timestamp value. The Window Registry differencing algorithm is available in Appendix B.1.
The result from snapshot comparison (namely, differential forensic analysis) are a list of entries that are deemed new, changed, modified or deleted. These results are added to another data structure (RESULTS) which retains all system changes and are later parsed and reported to the application profile data abstraction (discussed later in Section 5.3).

5.2.3.3 Data Collection Procedure

In order to provide an automated data collection procedure LiveDiff has been intentionally implemented as a console application to reduce required user interaction. The implementation of an automated method to perform data collection simplifies and accelerates application profile generation and is displayed in Figure 5.3.

![Data collection procedure implemented in LiveDiff](image)

Data collection follows a simple automated procedure that requires minimal user interaction. After executing LiveDiff, the user is prompted to enter the application name, version number and application life cycle phase; for example, if the user was performing the installation of TrueCrypt version 7.1a, they would enter: 1) TrueCrypt; 2) 7.1a; and 3) install. The user would press Enter to collect Snapshot1, be prompted to perform the actual application life cycle phase (e.g., install the application), and press Enter to collect Snapshot2. The two snapshots are then compared using the prescribed differential forensic analysis algorithms and the reports appended to the output file. The data collection procedure then starts a looped process. Snapshot1 is discarded and Snapshot2 copied to the Snapshot1 variable. For each subsequent life cycle phase the user must: 1) Enter the life cycle state; 2) Perform the application life cycle phase (e.g., open the application); 3) Press Enter to collect Snapshot2. Again, the two snapshots are compared and results appended to the output file. If subsequent application life cycle phases are required, the process is looped again until the user exits the program.
The prescribed automated data collection procedure greatly simplifies performing multiple snapshots and comparisons. This provides rapid recreation and data collection of the entire application life cycle. Theoretically, the ability to copy a previous end snapshot (Snapshot2) to the first snapshot position (Snapshot1) reduces the number of system snapshots by approximately 50%, as only one snapshot needs to be collected for each application life cycle phase. Thus far, the design incorporates data collection and subsequent comparison to identify digital artifact changes on a system. However, additional tool functionality is required to meet the system design requirements.

5.2.4 Tool Functionality

LiveDiff has been developed specifically to address the shortcomings of other system-level reverse engineering tools, primarily because other tools are not designed for reverse engineering application software to aid digital forensic investigations. The requirements of digital forensics are paramount for this research in terms of system design and software development. The additional functionality can be classified into three main categories: 1) Dynamic blacklisting of irrelevant operating system artifacts; 2) Inclusion of MD5 and SHA1 cryptographic hashing for data files; and 3) Reporting of comparison results to a specialised data abstraction. The following subsections describe each tool function while reporting of comparison results is later discussed in Section 5.3.

5.2.4.1 Dynamic Blacklisting

The system design requirements specified the use of blacklisting in an attempt to filter irrelevant operating system artifacts from differential analysis results. This is because system-level reverse engineering is known to produce a high number of unrelated results (see Section 3.2.3). In order to remove time consuming manual analysis, automated techniques are required to filter irrelevant entries while retaining digital artifacts of interest. To achieve this, it is proposed that blacklisting is implemented.

Blacklisting is a common practice in digital investigations and computing in general; for example, blocking spam email using a list of known bad email addresses. This research proposes an approach to perform data reduction using blacklisting to filter irrelevant results. The proposed technique has been named dynamic blacklisting, as the blacklist is dynamically generated at application run-time. This is in contrast to common blacklisting approaches which are manually populated before usage and populated with known bad or irrelevant entries. Manual blacklisting methods are not flexible because they are: 1) Too time consuming to create; and 2) Require blacklists to be created for every operating system. Dynamic blacklisting can be a powerful data reduction method if we assume the following hypothesis:

Hypothesis: That any filesystem or Registry entry that exists before an application is introduced to the testing environment is not unique to the application.

The dynamic blacklisting strategy follows simple logic; if a specific file system or Registry path (absolute logical path) existed before an application was introduced to the testing environment
it is not unique to the application and therefore not included in data collection. Dynamic blacklisting is achieved by collecting a system snapshot before performing any data collection or differential analysis. The entries in the snapshot are considered **known content** which exist before performing any actions with a specific application. The snapshot of known content is used to create the dynamic blacklist, where known content is established based on the full path of each file system or Registry entry.

To perform blacklisting during **LiveDiff** run-time a highly efficient string search and comparison implementation is required. This is because additional processing may significantly reduce data collection speed and adversely affect differential analysis results. In order to perform run-time blacklisting a **Prefix Tree**, or **Trie**, has been implemented using a `trieNode` data structure designed for string insertion and searching. Each file system and Registry entry found from the initial system snapshot is populated into the Prefix Tree using the full path acting as the string index. Figure 5.4 displays an example of a Prefix Tree that is populated with six file system entries. The inserted file system entries are listed on the right side of the figure under the **Paths** legend.

![Prefix Tree Diagram](image)

**Figure 5.4: Example of a prefix tree (trie) populated with file system entries**

After populating the dynamic blacklist, any subsequent entry is subjected to a search function against the populated prefix trees (blacklists). If a matching entry is found, the entry is not processed any further and is simply excluded from data collection and subsequent differential analysis; for example, **LiveDiff** is invoked, dynamic blacklist specified and an initial snapshot is collected and used to populate the prefix trees. Another snapshot is invoked,
the file system is scanned and each processed entry has its full path resolved and used to search against the prefix tree. If the path is found, the entry is discarded and the next entry processed. Otherwise, if the entry is not found it is processed and added to the snapshot.

LiveDiff implements two prefix trees: 1) blacklistFILES for file system entries; 2) blacklistREGISTRY for Registry entries. Both prefix trees are stored in volatile memory and only retained while running LiveDiff. Storing blacklists in memory adds a significant advantage when performing system-level differencing, since no file system entries are created to store the blacklist there is no chance of being erroneously included in the system snapshot.

A key advantage of the proposed dynamic blacklisting method is that since the blacklist is dynamically generated during run-time it should be scalable to any Windows-based testing system. This is exceptionally important because traditional operating system blacklists are tailored to a specific version (e.g., Windows XP or Windows Vista). A Windows XP blacklist on a Windows Vista system is unable to be (effectively) used as there are differences in the core operating system files. Basically, all systems are unique and contain different directory structures and data files that are a part of the operating system content. However, the dynamic blacklisting technique should be scalable to any system, removing the requirement to manually create a blacklist for every operating system for which LiveDiff will run on.

5.2.4.2 Data File Hashing

File hashing is a fundamental forensic analysis technique used to identify data files that are exactly the same (see Section 3.3.1). The use of reference sets to perform automated file detection on a target rely on computing and comparing file hash values. Therefore, the inclusion of file hashing is essential in the identification of data files between the application profile and target data set components of the system design. The reviewed literature identified a variety of potential file hashing solutions including: 1) Traditional hash functions such as Message Digest 5 (MD5) or Secure Hash Algorithm version 1 (SHA1); 2) Block-based hashing; and 3) Approximate matching using tools such as ssdeep and sdhash. The high-level research objective specifies an efficient solution suitable for digital forensic triage, therefore, a computationally efficient hashing algorithm is required. This requires a hash function with both fast hash value generation and comparison. Reviewed literature dictated that traditional file hashing is by far the most computationally efficient for hash generation and comparison (see Section 3.3.3 Table 3.6 and Table 3.7). Therefore, traditional hash functions have been selected. However, traditional hash functions are unable to detect similar files (see Section 3.3.2). This limitation has been counteracted by designing and implementing a collection of matching methods to perform matching of similar files (see Section 5.5.3).

Although traditional MD5 and SHA1 file hashing functions perform fast hash generation, it is hypothesized that calculating a hash value for every data file on a target system would be computationally inefficient. This is especially the case for performing data collection on a live system and would also be true for any suitable hashing algorithm commonly used in digital forensics. Initial testing of LiveDiff has confirmed this premise, taking approximately 15 minutes to calculate a SHA1 hash value for every data file on a default Microsoft Windows
installation. This is problematic as the time taken to perform data collection (snapshots) with file hashing would increase application profile creation time. Furthermore, an increase in time to capture snapshots could also adversely effect differential analysis results as more irrelevant system noise would be collected during the longer snapshot collection process. This could potentially increase the number of irrelevant digital artifacts that would need to be filtered in order to obtain a profile with only unique application software artifacts.

In order to provide a more efficient approach to file hashing the proposed dynamic blacklisting technique (see Section 5.2.4.1) has been leveraged in that only data files which are not blacklisted are hashed. It means that any data file that is present before performing differential analysis will not be subjected to hashing. When performing a snapshot of file system entries the file hashing function is only called on when: 1) The file system entry has not been blacklisted; and 2) The non-blacklisted file system entry is a data file (directories are not hashed). Only new data files created by the application software are therefore subjected to file hashing. This method results in a dramatically smaller number of calls to the hashing function while maintaining effectiveness by only hashing non-operating system files. Preliminary testing of this implementation discovered that hashing non-blacklisted files proved computationally efficient and only increased the time taken to collect a system snapshot by an average of two (2) seconds. Furthermore, preliminary testing also discovered that all unique application related data files were correctly hashed and included in the differential analysis results.

Two cryptographic hashing algorithms were selected for inclusion in the LiveDiff tool: 1) Message Digest 5 (MD5); and 2) Secure Hash Algorithm version 1 (SHA1). Both algorithms are widely accepted in digital investigations. Each hashing function has been implemented by using the Microsoft Cryptography Application Programming Interface (API), provided by the Wincrypt.h header file. To perform file hashing, an MD5 and SHA1 hashing function, CalculateMD5() and CalculateSHA1(), have been authored and included in the fileshot.c source code file. For each non-blacklisted data file that is encountered each hashing function can be called and the hash value appended to the FILECONTENT data structure. Inclusion of file hashing during LiveDiff execution is determined by user-specified command line options which controls if file hashing is to be performed and which hashing algorithm to implement.

5.2.5 Software Development: LiveDiff

LiveDiff has been designed based on the system requirements and a data collection procedure and specified tool functionality for digital artifact identification and documentation outlined. The LiveDiff implementation has resulted in approximately 6,000 lines of source code in seven different files. A total of four of these were authored specifically for LiveDiff: 1) livediff.c; 2) dfxml.c; 3) blacklist.c; and 4) output.c. These four source code files contain around 2,000 lines of new code while the remaining 4,000 or so lines of code has been taken from the Regshot project and subjected to various levels of modification. These files include: 1) regshot.c; 2) fileshot.c; and 3) global.h. Appendix B provides an
in-depth comparison of source code for both projects and the changes that have been made. In summary, approximately 2,500 lines (out of 4,000 lines) of original source code from the Regshot project has been either rewritten or heavily modified. In addition, a total of five (5) new programming functions were added to the existing Regshot code to achieve the desired functionality that LiveDiff required. Table 5.2 outlines a summary of the LiveDiff source files with an accompanying description of code purpose and functionality.

<table>
<thead>
<tr>
<th>Source File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>livediff.c</td>
<td>Program entry point. Parses command line arguments. Executes selected mode of operation including collecting and loading snapshots</td>
</tr>
<tr>
<td>regshot.c</td>
<td>Windows Registry processing. Executes Registry snapshot collection, comparison, saving and loading. Helper functions to fetch the full path, data type and value data for Registry entries</td>
</tr>
<tr>
<td>fileshot.c</td>
<td>File system processing. Executes file system snapshot collection, comparison, saving and loading. Helper functions to fetch the full path and calculate SHA1 hash values for file system entries</td>
</tr>
<tr>
<td>output.c</td>
<td>Processing of comparison results (system changes) to be reported. Manages output files and helps display snapshot and result information</td>
</tr>
<tr>
<td>dfxml.c</td>
<td>Writes comparison results in XML syntax including writing APXML structure, populate FileObjects and CellObjects from comparison results. Helper functions to write XML element formats</td>
</tr>
<tr>
<td>blacklist.c</td>
<td>Perform blacklisting of file system and Registry entries. Generates blacklists from a system snapshot and populates a Prefix Tree (or Trie) to enable full path indexing and searching</td>
</tr>
<tr>
<td>global.h</td>
<td>Header file. Specifies global variables, functions and structures</td>
</tr>
</tbody>
</table>

The output from LiveDiff is an application profile that is populated with file system and Windows Registry entries that are created, modified and/or deleted during the life cycle of an application. The populated application profile can be used as input to compare against a target data set. However, a viable data abstraction is first required in order to store, distribute and automate processing of the file system entries and Registry entries that have been identified.

5.3 Application Profile Data Abstraction

The application profile is the second component of the system architecture (see Figure 5.1). An application profile has previously been defined as a “collection of files that make up an application, the Windows Registry or Macintosh plist information associated with an application, volatile memory (RAM) information, document signatures and network traffic signatures” (Garfinkel 2010, p. S69). Therefore, an application profile is comprised of various digital artifact types that are uniquely associated with an application. The digital artifacts are commonly represented by metadata, do not contain actual content and are stored in a file (usually a document or database) for future processing. Each digital artifact is created, modified or
deleted through the application life cycle. This information is the resultant output from the
identification of digital artifacts and implemented in the LiveDiff tool (see Section 5.2.3).

This section describes the design and implementation of a suitable data abstraction to
retain application software information which will later be used to perform detection of known
digital artifacts by matching metadata properties. Design requirements are first specified
for an application profile data abstraction followed by the design to incorporate the relevant
digital artifact types and metadata property values selected. Further metadata property values
are identified to provide application software-specific information. Finally, an Application
Programming Interface (API) is designed and implemented based on the data processing
requirements.

5.3.1 Data Abstraction Requirements

An application profile requires a data abstraction to store, distribute, and automate pro-
cessing of digital artifacts. On the Microsoft Windows operating system, file system entries
-directories and data files) and Windows Registry entries (keys and values) encompass the
majority of the digital artifacts of interest to aid digital investigation (see Section 2.2.2). The
LiveDiff tool was designed specifically to identify these digital artifacts to aid in creating
an application profile. However, there is no data abstraction expressly designed to act as an
application software reference set (that is, an application profile). Furthermore, there is no
data abstraction that has the functionality to store multiple evidence sources in one document.
To achieve the desired system functionality the following design requirements are specified:

- Provide support for file system and Windows Registry entries
- Provide support for automated processing
- Adhere to digital forensic requirements
- Is open and extensible

5.3.2 Data Abstraction Design

The system requirements for an application profile data abstraction can be achieved through
the extension and development of previous partial solutions. It is advantageous, as well
as common practice, to utilize prior solutions that have been specifically designed for, and
adhere to, digital forensic requirements. The following subsections describe the design of a
data abstraction that will be used to store digital artifact metadata for file system entries and
then Registry entries, followed by a proposed data abstraction structure.

5.3.2.1 Digital Forensics XML

Digital Forensic XML (DFXML), in combination with Registry XML (RegXML), has been se-
lected as the data abstraction on which to develop a new application profile format. DFXML
conforms to all the identified requirements including functionality to document file system en-
tries, Windows Registry entries, as well as case provenance [Garfinkel 2012a]. Furthermore,
DFXML adheres to digital forensic requirements and has an Application Programming Interface (API) to provide automated processing. However, DFXML was not designed specifically for application software reference sets and requires additional design, development and implementation for this research project. According to [Garfinkel][2009], DFXML stores file system entries within XML tags named fileobjects. Extensive metadata is stored for each entry including file name, file size, allocation status, MAC timestamps, hash values and location of data using byte_run elements. Listing 5.1 displays an example of a DFXML fileobject for a digital image.

```xml
<fileobject>
  <filename>DCIM/100CANON/IMG_0044.JPG</filename>
  <filesize>105195</filesize>
  <partition>1</partition>
  <alloc>1</alloc>
  <mtime prec='2'>2008−12−25T04:21:44</mtime>
  <ctime prec='2'>2008−12−25T04:21:44</ctime>
  <atime prec='86400'>2008−12−24T00:00:00</atime>
  <libmagic>JPEG image data, EXIF standard 2.2</libmagic>
  <byte_runs>
    <byte_run offset='0' fs_offset='88576' len='32768'/>
    <byte_run offset='32768' fs_offset='1497600' len='32768'/>
    <byte_run offset='65536' fs_offset='6330880' len='39659'/>
  </byte_runs>
  <hashdigest type='md5'>cef79634dd3a86455a2cd00a691adff3</hashdigest>
  <hashdigest type='sha1'>916a88a00c58b7a566711acd25e61d549df5d303</hashdigest>
</fileobject>
```

Listing 5.1: DFXML fileobject example (Source: Listing taken from Garfinkel (2012a, p. 165))

According to [Nelson][2012], RegXML stores Registry entries using XML tags named keys and values. However, while DFXML stores flattened fileobjects, RegXML has a nested structure based on the hierarchical nature of the Registry. Listing 5.2 displays a RegXML example taken from the HKEY_CURRENT_USER Registry hive. The nested structure of RegXML makes it difficult to store a collection of single Registry entries. Therefore, a new RegXML structure is required to meet the prescribed design requirements. In addition, the design of DFXML and RegXML means that file system entries and Windows Registry entries are independent and, therefore, usually stored in different documents. Hence, a new data abstraction structure is required to store all digital artifacts in a single reference set document. Although amalgamation of all digital artifacts into a single abstraction is not essential, it makes sense to group data to ease distribution of the output. The following subsections introduce a new RegXML structure to store Registry information in a flattened structure, followed by a new application profile document structure that amalgamates DFXML and the revised RegXML structure components into a single entity.
5.3.2.2 Revised Registry XML Elements

As previously outlined, RegXML represents Registry entries in a nested XML structure where a Registry sub-key or value is stored under the associated parent key. Unfortunately, using a nested structure means that a Registry single entry is unable to be extracted and represented by itself (as each child entry requires the parent entry to establish the full path). To effectively store and process Registry entries a modified XML structure is required. This research revised the design of RegXML elements to mimic the flattened structure used by DFXML. Although not implemented in any Registry parsing tool, the DFXML Objects.py API provided a new set of Python objects to store flattened RegXML syntax in a new object named CellObjects. The new object bindings have the functionality to store a variety of Registry entry properties depending on the type of entry; either keys or values. Listing 5.3 displays the revised RegXML CellObject structure that has been designed for this research. The listing displays three Registry entries that are the same as the first three entries as displayed in Listing 5.2. As illustrated, the revised RegXML CellObject structure provides a single XML tag (element) for every Registry entry. The key difference is that each CellObject stores the full cellpath for every Registry entry. However, no Registry parsing tool can produce the revised RegXML syntax since the new structure has been specified in this research. Therefore, a new or modified tool will be required to perform RegXML generation of the Registry hive files that will be extracted from the target data set processing (discussed later in Section 5.4.3).

5.3.2.3 Application Profile Structure

When utilizing DFXML and RegXML, file system and Registry entry information is usually stored individually in a separate XML document. However, the design requirements specify that all digital artifact metadata should be stored in a single document. Therefore, in order to implement the proposed application profile, a hybrid data abstraction is required using a combination of the functionality provided by DFXML and RegXML. This new data abstrac-
tion has been named Application Profile XML, or APXML. Listing 5.4 illustrates the proposed structure of the APXML data abstraction. The document structure amalgamates storage of different digital artifact types into a single XML file using DFXML FileObject entries, to store file system information and the revised RegXML CellObject entries to store Registry information.

The root XML element, apxml, defines that the XML file is an APXML document. Various XML namespaces are included in the root element which specify unique XML elements (tags) and associated attributes that are stored in the document. A number of XML namespaces are needed to form the APXML structure as it includes DFXML FileObject entries, RegXML CellObject entries and DFXML delta annotations to describe FileObject and CellObject differencing states; for example, new, deleted and changed.

An APXML document contains a metadata element which documents information about the created application profile. It uses XML Dublin Core to annotate the XML document including the document type and publisher. Information is also stored about the application.
profile including application name and version number.

The creator element provides information about the tool used to create the APXML document and is sourced from the DFXML standard (version 1.0). The creator element outlines the program name and version number which created the document. Additional elements provide detailed information about program compilation and execution environments. The build_environment element contains information about the system that was used to compile the program. The execution_environment element specifies details about the system on which the program was run. Finally, the rusage element provides tool run-time CPU information. All of these XML elements are populated by LiveDiff when invoked.

Further information regarding the APXML structure is available in Appendix B.3. Examples are included for the creator and metadata elements, as well as an example of a partially populated APXML document (see Listing B.3). Thus far, the high-level design of a suitable data abstraction has been proposed using DFXML and revised RegXML elements. The following subsection discusses the inclusion of pertinent metadata properties to establish the information that should be retained for file system and Registry entries.

### 5.3.3 Digital Artifact Metadata Properties

The DFXML and RegXML standards store exceptionally detailed metadata for file system entries using FileObjects and Windows Registry entries using CellObjects. Both are stored as XML elements with standardised metadata properties to retain important information about files and Registry entries; for example, file name, and file hash value. However, the application profile for this research only requires particular metadata properties to perform digital artifact matching; for example, the file name, file size, hash value and allocation status are all beneficial metadata properties for file system matching. Similarly, the cell path, allocation status and data type are beneficial metadata properties for Windows Registry matching. However, there are a number of metadata properties that do not aid digital artifact matching. DFXML FileObjects store timestamp information which is not beneficial for file system or Windows Registry matching as this value changes depending on when the suspect installed or used the application. Another example of an unnecessary metadata property is the user identifier (UID) value which retains which user owns a file. A UID varies between users and systems and is therefore not useful to aid digital artifact matching either.

Table 5.3 displays the selected metadata properties that are populated in an application profile to aid digital artifact matching. A total of four different digital artifacts are recorded: 1) File system directories (folders); 2) File system data files; 3) Windows Registry key entries; and 4) Windows Registry value entries. Each digital artifact type is displayed with the selected metadata properties that are included in the application profile. The digital artifact metadata properties listed in Table 5.3 use the naming conventions from the DFXML project, specifically the Objects.py Application Programming Interface (API).

---

5 Although the timestamp metadata property is not required in the application profile for matching, it is essential that the framework output report includes timestamp information for any matched digital artifact found on the target data set. It provides evidence of when the suspect installed or last used an application.

6 See: [https://github.com/simsong/dfxml/blob/master/python/Objects.py](https://github.com/simsong/dfxml/blob/master/python/Objects.py)
Table 5.3: Overview of the metadata properties recorded in the application profile for different digital artifact types (DFXML Objects.py API naming conventions are used)

<table>
<thead>
<tr>
<th></th>
<th>File System</th>
<th>Windows Registry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Directory</strong></td>
<td>filename</td>
<td>cellpath</td>
</tr>
<tr>
<td>meta_type</td>
<td>meta_type</td>
<td>name_type</td>
</tr>
<tr>
<td>alloc_name</td>
<td>filesize</td>
<td>alloc</td>
</tr>
<tr>
<td>alloc_inode</td>
<td>sha1</td>
<td>data_type</td>
</tr>
<tr>
<td></td>
<td>alloc_inode</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>alloc_name</td>
<td>alloc</td>
</tr>
</tbody>
</table>

5.3.3.1 Additional Digital Artifact Metadata Properties

Although FileObject and CellObject store detailed metadata about file system and Registry entries, it was envisioned that additional metadata properties would be useful in other components of the system design, specifically to aid digital artifact matching. Therefore, a variety of other metadata properties have been designed and added. Table 5.4 outlines these properties including a description of the contents of the added metadata property.

Table 5.4: Properties added to the application profile for FileObject and CellObject

<table>
<thead>
<tr>
<th>Property</th>
<th>FileObject</th>
<th>CellObject</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_name</td>
<td>✔</td>
<td>✔</td>
<td>Application name (e.g., TrueCrypt)</td>
</tr>
<tr>
<td>app_state</td>
<td>✔</td>
<td>✔</td>
<td>Application state (e.g., install)</td>
</tr>
<tr>
<td>basename</td>
<td>✔</td>
<td></td>
<td>File name without logical full path</td>
</tr>
<tr>
<td>basename_norm</td>
<td></td>
<td>✔</td>
<td>Normalised file name without full path</td>
</tr>
<tr>
<td>filename_norm</td>
<td>✔</td>
<td></td>
<td>Normalised file name with full path</td>
</tr>
<tr>
<td>cellpath_norm</td>
<td>✔</td>
<td></td>
<td>Normalised cellpath of Registry entry</td>
</tr>
<tr>
<td>orphan_name</td>
<td>✔</td>
<td></td>
<td>Normalised deleted path</td>
</tr>
<tr>
<td>data_raw</td>
<td></td>
<td>✔</td>
<td>Value data in hexadecimal format</td>
</tr>
<tr>
<td>rootkey</td>
<td></td>
<td>✔</td>
<td>Registry hive root key (e.g., SYSTEM)</td>
</tr>
</tbody>
</table>

It is proposed that the DFXML Objects.py API is leveraged to parse and represent file system and Registry entry metadata in FileObject and CellObject. Any additional metadata properties need to be implemented in the existing Objects.py source code and is discussed in the following subsections.

5.3.3.2 Digital Artifact Classification

Classifying which particular application and which particular phase of the application life cycle a digital artifact is associated with is crucial to the application profile. It provides the forensic investigator with further information regarding application software usage; for example, installing an application is a different scenario from installing and then executing the software for a specific task. Both scenarios provide useful evidence for the investigator
to determine what tasks a suspect conducted with an application. The APXML structure therefore requires a flexible method to store particular application-specific information. Two additional metadata properties are therefore specified: 1) app_name stores the application name (e.g., TrueCrypt); and 2) app_state stores the application life cycle state (e.g., install, open, close, uninstall). These modifications made to the Objects.py API thus provide functionality for every FileObject and CellObject to have an app_name and app_state property available for storage and processing.

5.3.3.3 Metadata Property Normalisation

It is proposed that various metadata properties can be subjected to normalisation techniques to aid digital artifact matching. In this regard, five additional metadata properties are specified: 1) basename_norm stores a normalised base name (e.g., file name without the logical full path) for file system and Registry entries; 2) filename_norm stores a normalised path of a file system entry with logical full path; 3) cellpath_norm stores a normalised full path of a Registry entry; 4) orphan_name stores a normalised path for deleted entries to align with how fiwalk represents deleted file system entries; and 5) data_raw stores Registry value data in hexadecimal format without any data transformation (e.g., transforming the binary value of a Registry string value). The importance of metadata property normalisation is covered in detail later in Section 5.5.3.1 when outlining the digital artifact matching component of the system design.

5.3.3.4 Compilation of Digital Artifact Metadata Properties

This subsection has presented in-depth information and the associated design of the application profile in regards to storing pertinent metadata properties for file system and Registry entries. Available metadata properties from the DFXML project were first selected (see Table 5.3) and a collection of new metadata properties have been designed to aid in digital artifact matching (see Table 5.4). A summary of all available metadata properties are presented in Table 5.5 where each metadata property is provided using the prescribed naming conventions with an associated description and various examples of data that would stored by each property. These metadata properties will be used in the later design for digital artifact matching (see Section 5.5). The top half of the table specifies all properties for file system entries represented by FileObjects, while the second half of the table specified all properties for Registry entries represented by CellObjects. The bottom of the table displays application specific information (app_name and app_state) that is stored by both FileObjects and CellObjects.

Thus far, this section has designed a suitable data abstraction with all required information to store and represent digital artifacts in an application profile. The following subsection designs a suitable software solution to parse the designed application profile structure and contents.
Table 5.5: Summary of application profile metadata properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>File or folder full path</td>
<td>Program Files/TrueCrypt/TrueCrypt.exe</td>
</tr>
<tr>
<td>filename_norm</td>
<td>Normalised full path</td>
<td>%PROGRAMFILES%/TrueCrypt/TrueCrypt.exe</td>
</tr>
<tr>
<td>basename</td>
<td>File or folder name</td>
<td>TrueCrypt.exe</td>
</tr>
<tr>
<td>orphan_name</td>
<td>Deleted file name</td>
<td>$OrphanFiles/TrueCrypt.exe</td>
</tr>
<tr>
<td>meta_type</td>
<td>File system entry type</td>
<td>1 = file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = directory</td>
</tr>
<tr>
<td>filesize</td>
<td>File size in bytes</td>
<td>36563</td>
</tr>
<tr>
<td>sha1</td>
<td>SHA1 hash value</td>
<td>7689d038c76bd1d695d295c026961e50e4a62ea</td>
</tr>
<tr>
<td>alloc_inode</td>
<td>Metadata allocation status</td>
<td>1 = allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = unallocated</td>
</tr>
<tr>
<td>alloc_name</td>
<td>File allocation status</td>
<td>1 = allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = unallocated</td>
</tr>
<tr>
<td>cellpath</td>
<td>Registry cell full path</td>
<td>$$$PROTO.HIV\Classes\AppID\TrueCrypt.exe</td>
</tr>
<tr>
<td>cellpath_norm</td>
<td>Normalised full path</td>
<td>SOFTWARE\Classes\AppID\TrueCrypt.exe</td>
</tr>
<tr>
<td>basename</td>
<td>Name of Registry value</td>
<td>C:\Program Files\TrueCrypt\TrueCrypt.exe</td>
</tr>
<tr>
<td>basename_norm</td>
<td>Normalised value name</td>
<td>%PROGRAMFILES%/TrueCrypt/TrueCrypt.exe</td>
</tr>
<tr>
<td>rootkey</td>
<td>Normalised hive root key</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>name_type</td>
<td>Registry artifact type</td>
<td>k = Registry key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v = Registry value</td>
</tr>
<tr>
<td>alloc</td>
<td>Cell allocation status</td>
<td>1 = allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = unallocated</td>
</tr>
<tr>
<td>data_type</td>
<td>Registry value data type</td>
<td>REG_SZ = Null terminated string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REG_DWORD = 32-bit number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REG_BINARY = Binary data in any form</td>
</tr>
<tr>
<td>data</td>
<td>Registry value data</td>
<td>@C:\Program Files\TrueCrypt\TrueCrypt.exe</td>
</tr>
<tr>
<td>data_raw</td>
<td>Untransformed data</td>
<td>54 00 72 00 75 00 65 00 43 00 72 00 79 00 70 (in hexadecimal encoding)</td>
</tr>
<tr>
<td>app_name</td>
<td>Application name</td>
<td>TrueCrypt, Firefox, Chrome</td>
</tr>
<tr>
<td>app_state</td>
<td>Life cycle state</td>
<td>install, open, close, uninstall, reboot</td>
</tr>
</tbody>
</table>

119
5.3.4 Software Development: apxml.py

The APXML data abstraction requires an Application Programming Interface (API) to read and automate processing of APXML documents. Since the APXML data abstraction is similar to the standardised DFXML format, a similar approach to developing and implementing an API can be used.

[Garfinkel (2012a)] implemented an object-orientated API for the DFXML data abstraction in the form of the dfxml.py module. The dfxml.py module reads DFXML files using Python’s Simple API for XML (SAX) parser or Python’s xml.dom.minidom class. The DFXML document is parsed and Python objects are created to represent volume, fileobject and byte_run structures. [Nelson, Steggall, and Long (2014)] created an additional DFXML API, Objects.py which provides a mutative object-orientated model with built-in type-safety which conforms to the DFXML schema when reading and writing. Type safety is advantageous for forensic data processing as it provides the functionality to transparently check errors and that the correct data type is used for each metadata property; for example, a file name must be a string and the allocation status (alloc_inode, alloc_name or alloc) must be a boolean value (in the form of 0, 1, true or false).

5.3.4.1 APXML API Design Requirements

The APXML API has the following design requirements:

- Functionality to read, write and automate processing of APXML documents
- Capability to perform data type checking
- Support for file system entries using FileObjects from the Objects.py API
- Support for Registry entries using CellObjects from the Objects.py API

5.3.4.2 APXML API Functionality

In order to meet the requirements above the APXML API has been implemented using the Python programming language and a new module, apxml.py, has been authored. In order to store an APXML document, the apxml.py module implements Python objects for the following XML elements that are permitted in APXML documents:

- APXMLObject to store the entire APXML document and child objects
- MetadataObject to store document information
- CreatorObject to store document provenance
- RusageObject to store tool processing information
- FileObject to store file system entries
- CellObject to store Registry entries

All XML processing is handled by the Python ElementTree module which is included in the Python standard library, making it available on any system with Python installed.

7See: [https://github.com/simsong/dfxml/blob/master/python/dfxml.py](https://github.com/simsong/dfxml/blob/master/python/dfxml.py)
8See: [https://github.com/simsong/dfxml/blob/master/python/Objects.py](https://github.com/simsong/dfxml/blob/master/python/Objects.py)
9See: [https://docs.python.org/3/library/xml.etree.elementtree.html](https://docs.python.org/3/library/xml.etree.elementtree.html)
“The Python xml.etree.ElementTree module implements a simple and efficient API for parsing and creating XML data” [Python Software Foundation, 2015]. Each Python object class in the apxml.py module has methods to:

- `populate_from_Element()`: populate the object from an ElementTree element
- `to_Element()`: convert an object to an ElementTree element object
- `to_xml()`: convert an ElementTree element object to an XML string

Both FileObject and CellObject Python classes have been sourced from the DFXML project Objects.py API. Therefore, apxml.py requires that the Objects.py module be available and imported at run-time. In order to provide the required functionality, both classes were modified to store additional object properties (as stated in Table 5.4). The additional properties require that the Objects.py source code be updated to reflect the changes. Each added metadata property has been included to the object’s `_all_properties` set, new setters and getters to perform type checking have been added and each property has been included in the `to_Element()` functions. These modifications provide the functionality to populate and interact with the new metadata properties in the existing object classes.

Various helper methods, type casting and type checking functions have been sourced from the Objects.py API including:

1) `_ET_tostring` to convert an Element to a string;
2) `_qsplit` to split an Element object during parsing to a namespace and tag name pair in a Python tuple;
3) `_typecheck` to check object type against another specified object;
4) `_strcast` to cast a value to a sting;
5) `_intcast` to cast a value to an integer; and
6) `_boolcast` to cast a value to a boolean. One additional helper casting function, `_datecast`, was added to convert a timestamp to a Python datetime object to provide an easy technique to manipulate and compare date and time properties.

Finally, a statistics-based class has been added in an attempt to ease statistical processing during experimental testing. A Python class, named StatisticsObject, has been implemented to store statistical information regarding a specific APXML document. The `generate_stats()` function calculates statistics for a specified APXMLObject.

The apxml.py module is approximately 600 lines of original code written specifically for this research. In addition, the original Objects.py API required modification to implement the additional functionality needed for the system design; for example, adding extra metadata properties. These changes resulted in approximately 200 lines of added source code and approximately 100 lines modified source code. Appendix B.4 provides a collection of simple Python scripts to illustrate APXML API usage including functional examples to process APXML documents and examples of how the statistics class can be used to provide summary information of APXML document contents.

### 5.3.5 Adding APXML Output Support to LiveDiff

Thus far, this section has designed and implemented a specialised forensic data abstraction to store, distribute and automate processing of application software artifacts, namely file system and Registry entries. The LiveDiff tool design and implementation has been previously
specified (see Section 5.2) which identifies file system and Registry entries that are unique to an application. However, reporting the output of identified digital artifacts first needed a suitable data abstraction which has now been designed and implemented. Support can therefore be added to LiveDiff to output all differential analysis results to the Application Profile XML (APXML) document format.

This section specifies the design and implementation of adding APXML reporting functionality to the LiveDiff tool including reporting results to APXML documents, handling of special XML characters, handling of Unicode control characters and representing Windows Registry data to facilitate metadata correlation.

5.3.5.1 Reporting Results to the APXML Data Abstraction

A skeleton APXML file structure was designed and implemented to store digital artifact metadata specifically from application software (see Listing 5.4). The APXML structure leverages the DFXML standard to store file system entries as FileObjects and the associated revised RegXML standard to store Windows Registry entries as CellObjects. LiveDiff development needs to be advanced to provide the functionality to report the results of the specified metadata properties and associated values to aid digital artifact correlation.

Although Regshot only included support for text or HTML reporting functionality, the original output.c source code was used as a reference to understand and re-implement the reporting function from the original data structures. LiveDiff uses a new output.c file to parse all digital artifact changes (file system or Registry entries) that have been identified. Digital artifact changes are sourced from the RESULTS data structure that is populated after performing differential analysis. Two new functions have been authored to parse the comparison results and extract each file system and Registry entry to be reported. Each comparison result is individually passed to either: 1) PopulateFileObject() to produce a populated FileObject element in DFXML format; or 2) PopulateCellObject() to produce a populated CellObject in RegXML format. The populated XML element is then appended to the APXML document. Various methods to generate and print XML elements have been sourced and adapted from the PhotoRec tool which includes DFXML reporting for data files. A variety of challenges were discovered when implementing XML output including: 1) Handling of special characters and unicode control characters; and 2) Correctly representing Registry value data.

5.3.5.2 Handling Special XML Characters

The XML specification version 1.0 defines five special characters that must be correctly represented when printing character data (strings) ([Yergeau et al., 2004]). All XML processors will crash and present an error if a special character is encountered when parsing an XML document. Therefore, LiveDiff required functionality to escape these characters. Table 5.6

10See: http://sourceforge.net/p/regshot/code/HEAD/tree/trunk/src/output.c
11See: http://git.cgsecurity.org/cgit/testdisk/tree/src/dfxml.c
displays the five special characters with the name, original character, escaped character and
programming functions added to LiveDiff to search strings for each special character.

<table>
<thead>
<tr>
<th>Name</th>
<th>Original</th>
<th>Escaped</th>
<th>LiveDiff Check Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand</td>
<td>&amp;</td>
<td>&amp;</td>
<td>xml_ampersand_check()</td>
</tr>
<tr>
<td>double quote mark</td>
<td>“</td>
<td>&quot;</td>
<td>xml_quote_check()</td>
</tr>
<tr>
<td>single quote mark</td>
<td>’</td>
<td>'</td>
<td>xml_apos_check()</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>xml_gt_check()</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>xml_lt_check()</td>
</tr>
</tbody>
</table>

Each function searches an input string for any special character (&, “, ’, > and <) and
replaces it with the corresponding escape character (&amp;, &quot;, &apos;, &gt; and &lt;).
The string check functions are only called on when printing certain XML strings. During
preliminary testing special characters have been encountered in a variety of metadata property
values, including 1) Registry key and value full path (cellpath); 2) Registry value data (data); and 3) File system entry full path (filename). The content from each listed
property is, therefore, checked and fixed before appending to an APXML document.

Similar problems were identified when printing and parsing Unicode control characters. Similar to special XML characters, Unicode control characters cause an XML parser to
-crash during document parsing. Testing identified control characters were present in only
Registry value data. In order to fix this, a new function named xml_check_control() has been included. Each Registry value data entry is checked for control characters by utilising
the built-in C programming function iswcntrl()\(^\text{12}\). If at least one control character is found,
the Registry value data is subject to Base64 encoding using the CryptBinaryToString() function from Wincrypt.h library. If a control character is found and value data encoding
performed, the data_encoding metadata property attribute is added to the CellObject
with the attribute base64. This provides the capability to determine data encoding during
later processing.

5.3.5.3 Representing Registry Value Data

Another identified functional limitation was how Registry value data is represented in the
original Regshot code and in other Registry parsing tools. When informally comparing
LiveDiff output to other Registry parsing tools it was discovered that many tools inter-
preted, encoded and output Registry value data in a variety of different formats. Simply,
LiveDiff performed different value data transformation (encoding) compared other tools.
Although not tested, this would inevitably cause problems when performing correlation of
Registry value data between the application profile and target data set. Table 5.7 provides

\(^\text{12}\) The iscntrl() C programming function the commonly known function to check for control characters. The implemented iswcntrl(). function is a wide character (Unicode) function available in Visual C++. Since LiveDiff is authored for full Unicode support the wide character function was implemented.
examples of Registry values represented in output from LiveDiff compared to the hivexml tool as an example. Since the Windows Registry is stored in a binary data format, the corresponding hexadecimal representation is also included.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>LiveDiff</th>
<th>hivexml</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_DWORD</td>
<td>0x00000001</td>
<td>1</td>
<td>00 00 00 01</td>
</tr>
<tr>
<td>REG_QWORD</td>
<td>0x0000000000000000</td>
<td>0</td>
<td>00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>REG_BINARY</td>
<td>04 C0 00 00</td>
<td>BMAAAA==</td>
<td>04 C0 00 00</td>
</tr>
</tbody>
</table>

In order to simplify Registry value data correlation a new metadata property has been added to RegXML CellObjects to store the data extracted from Registry values. The new element was named data_raw and included in APXML documents for all Registry value entries (see Table 5.5 for an example). This new element stores the native data in hexadecimal format without performing any data encoding or transformation. In the above example (Table 5.7), the hexadecimal value is stored by the data_raw element. The original data element has not been removed as it maintains human readability of Registry value data entries; for example, Registry values with a string data type.

5.3.6 Application Profile: Data Abstraction Summary

The Application Profile XML (APXML) document is a hybrid data abstraction which leverages the standardised DFXML and RegXML forensic data abstractions to store metadata associated with file system and Windows Registry entries. Various additional metadata properties have also been implemented for the existing FileObject and CellObject Python classes that are required for application specific information. The incorporation of a Python API, apxml.py, has been designed and authored as a key feature to aid automated processing of application profiles stored in APXML documents.

The final result of integration of APXML reporting to LiveDiff is the functionality to successfully report all file system and Registry changes to the prescribed data abstraction format in order to automate processing of digital artifacts from application software. The next component in the system architecture is the processing and transformation of the target data set into a metadata representation to facilitate digital artifact matching.

5.4 Target Data Set Processing

The third component in the system architecture is processing the target data set. This section discusses, in detail, the various operations performed to the target data set which includes parsing, extracting and transformation of file system and Windows Registry entries to facilitate digital artifact matching.

In a digital investigation the target data set is typically a forensic disk image (evidence file) which is a bit-by-bit copy of a digital storage device. Common examples include Hard
Disk Drives (HDD) and Universal Serial Bus (USB) flash memory drives. Processing the target data set involves parsing the file system and Registry data structures to extract and transform entries in a metadata representation of the original evidence source. The ultimate outcome are metadata reports stored in a suitable format for digital artifact matching. Figure 5.5 displays the method used to process the target data set and generate reports populated with metadata representing the original evidence sources. The following subsections outline the target data set processing requirements, followed by the method used to generate file system and Windows Registry metadata reports.

Figure 5.5: High-level overview of target data set processing and the method used to generate DFXML and RegXML reports to represent the original evidence sources

5.4.1 Target Data Set Processing Requirements

The application profile has been specified to be stored in the designed Application Profile XML (APXML) data abstraction (see Section 5.3). The APXML structure stores digital artifact information in DFXML FileObjects for file system entries and RegXML CellObjects for Windows Registry entries. Therefore, it is logical that the target data set should also be represented in the same format in order to support subsequent analysis using the same standardised forensic data abstractions. Two different evidence sources from the target data set therefore require processing: 1) File system entries; and 2) Windows Registry entries.
Both need to be transformed into the DFXML and RegXML structure to represent the original evidence sources and leads to the following design requirements:

- Functionality to generate DFXML representation of target file system
- Functionality to extract Windows Registry hive files from evidence file
- Functionality to generate RegXML representation of Window Registry hive files
- Functionality to recover deleted (unallocated) file system and Registry entries

The following subsections outline the design to achieve target data set processing and produce a high-level metadata representation of the original evidence source. Firstly, an overview of file system metadata generation is outlined and achieved using existing solutions. Secondly, the requirements for Windows Registry metadata generation are specified followed by design and implementation of a new parsing tool to generate the revised RegXML structure.

5.4.2 File System Metadata Generation

The first step in target data set processing is to parse the forensic disk image (evidence file) and generate file system metadata. This is shown on the left-hand side of Figure 5.5 and displays the process used to parse the target data set file system to produce a DFXML report. The \texttt{fiwalk}\textsuperscript{13} tool has been used to generate the report as the tool is inherently linked to the DFXML project and data abstraction. \texttt{fiwalk} was the first program authored to generate DFXML reports and describes the partitions and files on a hard drive or disk image \cite{Garfinkel2009}. The DFXML report contains every file system directory and data file that resides on the target data set. Each file system entry is stored in a DFXML \texttt{FileObject} with extensive metadata properties. The \texttt{fiwalk} tool provides built-in support to recover deleted file system entries.

The \texttt{Objects.py} API is to be leveraged to generate the DFXML report. The API provides support to pass the target evidence file to the \texttt{Objects.iterparse()} function, which in turn invokes \texttt{fiwalk} against the target. Each file system entry returned by \texttt{fiwalk} is populated into a \texttt{FileObject} and built-in transparent error and type checking performed based on the DFXML schema (\texttt{dfxml.xsd}\textsuperscript{14}). This provides a robust solution that is able to handle file system errors if present.

5.4.3 Windows Registry Metadata Generation

The second step in target data set processing is to find, extract and process Windows Registry hive files to produce RegXML reports that represent the original evidence source. Unfortunately, generating Registry metadata is not as simple as the file system counterpart. This is primarily due to the inability of commonly implemented Registry parsing tools to recover deleted Registry entries. Therefore, it is important to first establish the system design requirements in order to successfully achieve the required system functionality.

\textsuperscript{13}See: http://digitalcorpora.org/downloads/fiwalk/
\textsuperscript{14}See: https://github.com/dfxml-working-group/dfxml_schema/blob/master/dfxml.xsd
5.4.3.1 System Design Requirements

The system design requirements for Windows Registry metadata generation are:

- Functionality to process an evidence file and extract Registry hive files
- Support to parse offline Registry hive files and generate the revised RegXML format
- Functionality to recover deleted Registry entries
- Support for different operating systems (desirable)

The first requirement is the extraction of Registry hive files from the target data set. The regxml_extractor\(^{15}\) tool would be the obvious choice as it was specifically designed to extract Windows Registry hive files and generate RegXML reports of each extracted hive file (Nelson 2012). However, regxml_extractor is limited due to reliance on the hivexml\(^{16}\) tool to generate XML. The hivexml tool has the following limitations:

- Inability to process deleted (unallocated) Registry entries
- Exports Registry entries to a nested XML structure\(^{17}\)
- Only available on Linux-based operating systems

The inability to recover deleted Registry entries is a major limitation when attempting to correlate digital artifacts from application software. Without recovery of deleted entries it impossible to identify deleted Registry entries from application software that has been uninstalled. Therefore, this research requires a solution to extract the Registry hive files and subsequently parse each file and produce a metadata representation of all Registry entries, including deleted keys and values. Furthermore, the solution must output Registry entries in the revised RegXML format (see Listing 5.3).

A total of approximately 20 Registry parsing tools and different programming libraries were found that support processing Windows Registry hive files. However, as discovered, there are very few that support the recovery of deleted Registry entries, while none have the functionality to export to the revised RegXML syntax. In terms of Registry parsing tools only three were found that support the recovery of deleted Registry entries.

1) Morgan (2008) developed reglookup-recover\(^{18}\) which has the functionality to recover unallocated Registry entries and is available for Microsoft Windows and Linux based systems

2) Thomassen (2008) developed the regslack\(^{19}\) Perl script (and in Windows executable format) to recover unallocated Registry entries

3) Zimmerman (2015) developed the Registry parser project\(^{20}\), a fully featured offline Registry hive parser written in the C# programming language

15See: https://github.com/ajnelson/regxml_extractor
16See: http://libguestfs.org/hivex.3.html
17See: https://github.com/ajnelson/regxml_extractor/issues/4 for additional details.
18See: http://projects.sentinelchicken.org/reglookup/
19See: https://github.com/keydet89/Tools/blob/master/source/regslack.pl
20See: https://github.com/EricZimmerman/Registry
Unfortunately, none of the above tools, that is reglookup-recover, regslack and the Registry parser have the functionality to export to RegXML syntax. Nor is there an available performance comparison for each tool (in terms of effectiveness or efficiency) which makes selecting a possible solution difficult. However, a manual review of source code revealed the Registry parser project to be much more exhaustive in terms of active software development and extensive documentation and as a result it was selected as the basis for the creation of a new tool in this work. The design and implementation of a modified tool based on the Registry parser project is discussed in the following subsections.

5.4.3.2 Windows Registry Metadata Generation Solution

The right-hand side of Figure 5.5 displays the solution for the process used to generate RegXML reports for Registry hive files from the target data set. There are two inputs to the Windows Registry metadata generation process: 1) The forensic disk image (evidence file); and 2) The previously generated DFXML report previously generated from file system metadata generation (see left-hand side of Figure 5.5). The Registry hive files are extracted using a newly authored Python program, named HiveExtractor.py. Each extracted hive file is then processed using the new Registry processing tool, named CellXML-Registry, which generates a RegXML report to represent the original Registry hive file. The following subsections discuss each of the implemented tools used for Windows Registry metadata generation.

5.4.3.3 Software Development: HiveExtractor.py

The HiveExtractor.py script has been specifically designed and implemented to process the target data set, discovered Registry hive files using file system metadata (the DFXML report) and extract the hive files to be processed. HiveExtractor.py is based on the rx_extract_hives.py script from the regxml_extractor tool. HiveExtractor.py requires three inputs: 1) Forensic disk image (evidence file); 2) DFXML report previously generated by fiwalk; and 3) A user specified output directory for exporting Registry hive files and the tool log file. HiveExtractor.py processes the DFXML report and searches the full path of each FileObject for known Registry hive file locations (based on the logical file system path). If a hive file is discovered it is extracted and saved to the specified output directory. After executing HiveExtractor.py against the target data set the output directory is populated with all discovered Registry hive files. Each hive file requires additional processing to transform the contents into a RegXML report.

The full HiveExtractor.py script is provided in Appendix B.5. Additionally, the script is available online from the author’s GitHub account. The script totals approximately 200 lines of original Python source code.

---


5.4.3.4 Software Development: CellXML-Registry

Based on the system design requirements (see Section 5.4.3.1), a better solution is needed to provide the functionality to process Registry hive files and generate a metadata representation of the Registry entries. The tool requirements specified support for the recovery of deleted Registry entries and functionality to export to the revised RegXML CellObject structure. A new tool, CellXML-Registry has been developed using the Registry library from the Registry parser project. The ExampleApp\textsuperscript{23} from the Registry parser project was used as a reference when creating CellXML-Registry.

To provide support for reporting in RegXML syntax two new functions were included in the RegistryHive.cs source code file. After processing a Registry hive file the new ExportDataToXMLFormat() function is called which starts exporting all Registry entries to the RegXML format. After processing the Registry hive root key, the new recursive DumpKeyXMLFormat() function is called which generates CellObjects for every Registry entry. After a Registry key is processed, the next key entry is called by the recursive function.

Initial testing revealed similar problems to those encountered by LiveDiff when generating XML to represent Registry entries, specifically XML special characters and unprintable control characters (see Section 5.3.5.2). Therefore, additional functions have been authored and implemented to check certain strings before printing results in XML format: 1) ControlXMLCharacterCheck() to search and encode Unicode control characters in Base64 encoding; and 2) SpecialXMLCharacterCheck() to search and replace special characters with their escaped counterpart (see Table 5.6). Both functions are called for the same metadata properties as implemented in LiveDiff.

The CellXML-Registry source code is provided in Appendix B.6. The additional functions that have been included in the Registry parser project to provide XML report functionality have also been included for reference. CellXML-Registry is available online from the author’s GitHub account\textsuperscript{24} and contains approximately 250 lines of original C# source code, while the modifications to the Registry parser project contains approximately 300 lines of additional C# source code to produce RegXML output.

5.4.4 Target Data Set Processing Summary

Target data set processing generates reports that represent the original evidence sources of the file system and Registry. Each report is used as input to the digital artifact matching stage to correlate digital artifacts between the application profile and target data set. The next, and final, component in the system architecture involves digital artifact matching.

\textsuperscript{23}See: \url{github.com/EricZimmerman/Registry/tree/master/ExampleApp}
\textsuperscript{24}See: \url{https://github.com/thomaslaurenson/CellXML-Registry} for full source code and \url{https://github.com/thomaslaurenson/CellXML-Registry/releases} for precompiled executable binaries (.exe files) for Microsoft Windows.
5.5 Digital Artifact Matching

Digital artifact matching is the fourth and final component of the system architecture which correlates an application profile(s) (APXML documents) against the metadata representation of the target data set (DFXML and RegXML documents). Each document is used as input to the matching process and links between digital artifacts are established by matching selected metadata properties. The output of the process is a list of detected digital artifacts that are reported to the investigator. Any detected digital artifact can be considered known content that is uniquely associated with a profiled application.

A variety of methods to detect application software were discussed in Section 3.3, primarily performing file identification using cryptographic hashing algorithms – the de facto standard in digital forensic investigation and path matching/searching for Windows Registry entries. Various other avenues were investigated for potential solutions including newer forensic techniques such as block-based hashing and similarity digests. Other computing fields were also investigated for potential, including common anti-virus and malware detection techniques such as signature-based and heuristics-based methods. However, given the system design requirements, the proposed metadata matching approach coupled with traditional file hashing was selected as it is a highly reliable and efficient solution that fits the research objective of a forensic triage solution. This solution incorporates the requirement of an initial forensic examination (triage) allowing the system output to be directly used as input to other commonly used forensic tools.

This section discusses the design and implementation of a solution to perform matching of digital artifacts. Design requirements and specifications are based on the developed research objectives. A novel approach to implement path normalisation is designed and a selection of advanced matching methods will be established. The actual process to achieve correlation of the file system entries and Windows Registry entries are then individually designed and implemented in a proof-of-concept tool. Forensic reporting then conveys the tool results to an investigator. Finally, the implementation of the system design is in the form of a newly authored forensic analysis tool.

5.5.1 Digital Artifact Matching Requirements

The primary goal of digital artifact matching is to detect digital artifacts from the target data set that are also in the application profile(s). The matching process is divided into two phases: 1) Matching of digital artifacts from the target file system; and 2) Matching of digital artifacts from the target Windows Registry hive files. Currently, a tool does not exist to perform correlation of multiple evidence sources (e.g., file system and Windows Registry entries) or correlation of multiple digital artifact metadata properties (e.g., file hash value, file name, file path and allocation status). Available reference solutions operate solely on data files using cryptographic hash value to detect known content (see Section 3.3). Therefore, a new software tool is required.

As specified in the research objectives (see Section 4.2.2.5), digital artifact matching needs
to be accomplished in a variety of complex scenarios; for example, matching digital artifacts from different operating system versions and different application software versions. Any detected results from digital artifact matching need to be reported to an investigator to ultimately ascertain the presence, or absence, of application software. Therefore, the system design requirements for digital artifact matching are:

- Support for comparison of file system entries (FileObjects)
- Support for comparison of Windows Registry entries (CellObjects)
- Support for completely automated comparison using multiple metadata properties
- Functionality to perform digital artifact matching in complex scenarios
- Functionality to report digital artifact matches in a human and machine readable format
- Functionality to ascertain application software presence

The specified system requirements could not be satisfied by current tool availability and so a new software tool is essential to provide the desired support and functionality. The tool needs to correlate different digital artifact types which each have different metadata properties. Lower-level research objectives four and five specify that the design should address limitations of other solutions and provide functionality to detect digital artifacts in complex matching scenarios such as different operating system versions and from different application software versions. A similar design requirement is the detection of fragile data type; for example, a data file that has a variable hash value (e.g., an application log file or configuration file). Finally, the tool needs to report any detected digital artifacts to the investigator in the form of a human readable data abstraction, as well as a machine readable data abstraction that provides the functionality for further automated processing of tool output.

5.5.2 Advanced Tool Design: Vestigium

A new tool has been authored to perform digital artifact matching between the application profile(s) and target data set. The resultant proof-of-concept tool has been named Vestigium, from Latin meaning footprint or trace. This name is highly fitting for the system design which attempts to determine the footprint that application software creates and trace evidence that software leaves on a computer system. Vestigium is completely original work produced for this research project. The Python programming language was selected as all the data abstraction inputs to the tool (APXML, DFXML and RegXML documents) have an API written in Python. The tool has been released using the open-source GNU General Public License (version 2) license and is freely distributed and available to other researchers and practitioners. Vestigium is available online from the author’s GitHub account\[25\].

The following subsections outline the core design of the Vestigium tool including a digital artifact matching strategy, file system matching process, Windows Registry matching process, forensic reporting of digital artifact matches and an overview of software development.

\[25\]See: https://github.com/thomaslaurenson/Vestigium/
5.5.3 Digital Artifact Matching Strategies

A major constraint when implementing reference sets to perform automated detection of digital artifacts is the failure to detect potential evidence in complex matching scenarios (see Section 4.2.1.3). The following list highlights the different scenarios that need to be solved in this research:

- Functionality to detect fragile digital artifact types; for example, a log file with variable content and variable file hash value
- Functionality to detect digital artifacts on a different operating system version
- Functionality to detect digital artifacts from a different application software version

Based on the specified system design requirements and the prescribed digital artifact matching scenarios, the following digital artifact matching strategies are required: 1) File system matching methods to correlate digital artifact metadata properties for file system directories and data files; and 2) Windows Registry matching methods to correlate digital artifact metadata properties for Registry keys and values. Additionally, to aid matching logical path properties (e.g., file system paths and Registry paths) between operating system versions the technique of path normalisation has been designed and implemented to transform file and Registry path values.

5.5.3.1 Path Normalisation

Each version of Microsoft Windows introduces slight variations in how and where data is logically stored by the operating system. For example, the logical file system location of a user’s home directory (in the example the username forensic) in Windows XP and Windows 7 are respectively:

Windows XP: C:\Documents and Settings\forensic
Windows 7: C:\Users\forensic

Furthermore, a username can be any character combination as selected by the user. This results in variable file system paths between systems and, ultimately, the inability to correlate the full path of a file system and/or Registry entry in many scenarios. Therefore, a method is needed to transform full path values to a generalized value to allow path correlation. Path normalisation is a transformation method designed specifically in this research to address full path correlation and has been defined by the author as:

Path normalisation is the process of transforming a full path value of a file system or Registry entry to a generalized value, thus, enabling correlation in complex matching scenarios where the logical location differs (e.g., different operating system versions or paths with variable user content)

To further explain path normalisation, the example of a user’s home directory in Microsoft Windows XP and Windows 7 can be applied. Path normalisation would transform both
file system paths to the same generalized value as each path logically points to the same location, that is, a user’s home directory. Path normalisation would also remove the username component as matching cannot be performed between two file system paths with different user generated content. Furthermore, in the Microsoft Windows operating system family, each hard drive has an assigned drive letter (e.g., C:\). Path normalisation would remove the drive letter to enable correlation of file path information where the drive letter may differ between systems.

Path normalisation has been previously used in digital forensics in an informal context without specification of the technique; for example, the fiwalk tool automatically removes the assigned root drive letter/value. This is a type of path normalisation and is not just beneficial to full path values for file system artifacts. Carvey (2011) states that Windows Registry key and value structure often changes (sometimes dramatically) between Windows versions. Therefore, path normalisation is beneficial to both file system and Registry artifacts.

In this research, path normalisation needs to be implemented to achieve two particular goals: 1) The ability to correlate the full path value on different Windows operating system versions where the path has the same function but a different naming convention; and 2) Replace random values, such as the variable username value, to correlate full path values. It is proposed that path normalisation can be achieved using environment variables to normalise the path value and remove/transform operating system-specific or user-specific values from the full path value. Microsoft has specified known folder naming conventions that use a common variable name for different file system paths (Microsoft, 2014). Using this naming convention technique, the previous example of a user’s home directory can be normalised to %USERPROFILE%. Table 5.8 displays four different examples of a user’s home profile for two different operating system versions (again, Windows XP and Windows 7), with the original path and the transformed normalised path.

Table 5.8: Path normalisation examples using environment variable naming conventions to translate the full path of a user’s home directory in Windows XP and 7 (in the examples the Windows XP username is forensic and the Windows 7 username is investigator)

<table>
<thead>
<tr>
<th>OS</th>
<th>Original Path</th>
<th>Normalised Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings\forensic</td>
<td>%USERPROFILE%</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Users\investigator</td>
<td>%USERPROFILE%</td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings\forensic\Desktop</td>
<td>%USERPROFILE%/Desktop</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Users\investigator\Desktop</td>
<td>%USERPROFILE%/Desktop</td>
</tr>
</tbody>
</table>

Table 5.8 illustrates that path normalisation can translate and return a full path value that can provide the functionality to correlate a full path property value for a digital artifact on different versions of Windows operating systems. Even though a different file system location and different username is apparent in the examples provided, both variable values can be normalised to allow full path correlation. As stated, path normalisation is also important for Windows Registry entries. According to research by Metz (2016a), the primary difference between a Windows Registry full path is the root key which varies based on: 1)
Windows operating system version; and 2) The hive file type. Windows Registry path normalisation should replace the root key with standardized Windows naming conventions such as: SOFTWARE and SYSTEM for the hive file that is being processed. Additionally, Windows Registry key entries are not case sensitive, meaning that key names do not rely on matching the case of the path to be correlated. For example, the RegOpenKeyEx() function from the Windows API does not require the correct case to open Registry keys (Microsoft, 2016).

Based on the preceding discussion, this research has designed and implemented path normalisation to aid matching of digital artifact path properties. Specifically, two Python modules have been designed: 1) File system path normalisation implemented in the Python module named FilePathNormaliser.py; and 2) Registry path normalisation implemented in the Python module named CellPathNormaliser.py. Both Python modules are designed to take a file system or Registry full (absolute) path value as input and produce a normalised path as output. The source code for both modules are available in Appendix B.7 and also electronically distributed with the Vestigium tool.

The techniques for path normalisation have been designed and implemented based on the authors’ knowledge coupled with preliminary testing of authored application profiles. The specified path normalisation techniques are outlined in the tables on the next page and display an actual path and the subsequent normalised path example for different operating system versions/platforms. Table 5.9 outlines all normalisation techniques for file system path values that have been implemented in the system design. Commonly observed file system paths for application software that require normalisation have been identified including: 1) The Program Files directory used to store application specific files; 2) The All User directory used to store global application configuration information; 3) The Users home directory to store user-specific application configuration information; 4) The AppData directory to, again, store user-specific application configuration information; 5) The Start Menu directory to store Windows ShortCut link files used for application execution; and 6) The Windows directory to store system-relevant application files or configuration information.

Table 5.10 displays normalisation techniques for Registry root key and path values. Three techniques have been implemented: 1) Root key normalisation to transform variable root key values to common Registry hive names (e.g., software, system); 2) Transformation to lower case for all Registry key paths; and 3) Normalisation of Control Set names from the SYSTEM Registry hive. Table 5.11 displays normalisation techniques for path information stored by Registry value data that have been implemented. Two normalisation techniques have been implemented: 1) Transformation of Registry value data where the property stores a file system path; and 2) Decryption and transformation of UserAssist Registry values which contain a ROT-13 encrypted file system path. According to Stevens (n.d.), UserAssist Registry value data is encrypted using the ROT-13 algorithm, therefore, any UserAssist value data is decrypted using the Python standard codec library to decrypt the path value. Any file system path stored in Registry value data is used as input to the FilePathNormaliser.py module, subjected to normalisation and then appended to the original CellObject.

---

26See: https://github.com/thomaslaurenson/Vestigium/tree/master/src
Table 5.9: Overview of file system path normalisation values. The actual file system path is displayed with Windows XP and Windows 7 examples followed by normalised path examples.

<table>
<thead>
<tr>
<th>OS</th>
<th>Actual path</th>
<th>Normalised path</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86/x64</td>
<td>C:\Program Files</td>
<td>%PROGRAMFILES%/TrueCrypt/TrueCrypt.exe</td>
</tr>
<tr>
<td>x64</td>
<td>C:\Program Files(x86)</td>
<td></td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings\All Users</td>
<td>%ALLUSERSPROFILE%/Desktop/TrueCrypt.lnk</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Program Data</td>
<td></td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings&lt;username&gt;</td>
<td>%USERPROFILE%/Desktop/TrueCrypt.lnk</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Users&lt;username&gt;</td>
<td></td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings&lt;username&gt;\Application Data</td>
<td>%APPDATA%/TrueCrypt/Configuration.xml</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Users&lt;username&gt;\AppData\Roaming</td>
<td></td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings\All Users\Start Menu</td>
<td>%STARTMENU%/Programs/TrueCrypt</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\Users&lt;username&gt;\Start Menu</td>
<td></td>
</tr>
<tr>
<td>WinXP</td>
<td>C:\W\IN\D\OW\NS\system32</td>
<td>%SYSTEMROOT%/drivers\truecrypt.sys</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\W\IN\D\OW\NS\system32</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10: Overview of Windows Registry path normalisation. The actual Registry path is displayed with Windows XP and Windows 7 examples followed by the normalised path examples. Two Registry root key examples are displayed. The system hive examples display the use of the environment variable naming convention for the system control set key.

<table>
<thead>
<tr>
<th>OS</th>
<th>Actual Path</th>
<th>Normalised Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinXP</td>
<td>$$$PROTO.HIV\Classes\tc</td>
<td>software\classes\tc</td>
</tr>
<tr>
<td>Win7</td>
<td>CMI-CreateHive{D43B12B8-09B5-40DB-B4F6-F6DFEB78DAEC}\Classes\tc</td>
<td>software\classes\tc</td>
</tr>
<tr>
<td>WinXP</td>
<td>$$$PROTO.HIV\ControlSet001\services\truecrypt</td>
<td>system%controlset%\services\truecrypt</td>
</tr>
<tr>
<td>Win7</td>
<td>CMI-CreateHive{2A7FB991-7BBE-4F9D-B91E-7CB51D4737F5}\ControlSet001\services\truecrypt</td>
<td>system%controlset%\services\truecrypt</td>
</tr>
</tbody>
</table>

Table 5.11: Overview of Windows Registry base name normalisation. Examples from two Registry keys are taken: 1) NewShortcuts; and 2) UserAssist. Both examples display the operating system version, actual path and associated normalised path values.

<table>
<thead>
<tr>
<th>OS</th>
<th>Actual Path</th>
<th>Normalised Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinXP</td>
<td>C:\Documents and Settings\All Users\Start Menu\Programs\TrueCrypt\TrueCrypt.lnk</td>
<td>%STARTMENU%/Programs/TrueCrypt/TrueCrypt.lnk</td>
</tr>
<tr>
<td>Win7</td>
<td>C:\ProgramData\Microsoft\Windows\Start Menu\Programs\TrueCrypt\TrueCrypt.lnk</td>
<td>%STARTMENU%/Programs/TrueCrypt/TrueCrypt.lnk</td>
</tr>
<tr>
<td>WinXP</td>
<td>HRZR_EHACNGU\P:\Qbphrzaf\nq\Frggvytf\VHR\Qrfrgbc\GehrPeleg Frghc 7.1n.rkr</td>
<td>%USERPROFILE%/Desktop/TrueCrypt Setup 7.1a.exe</td>
</tr>
<tr>
<td>Win7</td>
<td>P:\H\ref\sberafvp\Qrfrgbc\GehrPeleg Frghc 7.1n.rkr</td>
<td>%USERPROFILE%/Desktop/TrueCrypt Setup 7.1a.exe</td>
</tr>
</tbody>
</table>
Path normalisation may prove to be advantageous for digital artifact detection, as an application profile authored using one Windows version could also be effective at detecting digital artifacts on other Windows versions. Path normalisation could, therefore, overcome the need to create an application profile for every operating system version. Profile authoring is a time consuming process and producing one profile that can detect digital artifacts on multiple operating systems would be exceptionally beneficial.

5.5.3.2 Available Digital Artifact Metadata Properties

It is proposed that a selection of matching methods needs to be designed and implemented based on the type of digital artifact that needs to be correlated. The system design incorporates four different digital artifact types: 1) File system directories; 2) File system data files; 3) Registry keys; and 4) Registry values. Each digital artifact type is retained by FileObjects in the meta_type element and by CellObjects in the name_type element. This provides the functionality to determine what type of digital artifact is stored. Table 5.12 displays a complete list of metadata properties available for each of the four digital artifact types.

Table 5.12: Complete list of available APXML metadata properties for directories, data files, Registry keys and Registry values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Directory</th>
<th>Data File</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_type</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>name_type</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>filename</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filename_norm</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>basename</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>basename_norm</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cellpath</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cellpath_norm</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>orphan_name</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>alloc_name</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>alloc_inode</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>alloc</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>filesize</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>md5</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>sha1</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>data_type</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data_raw</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>app_name</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>app_state</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Each digital artifact type has different metadata properties available, so each type requires

\footnote{Although environment variables are widely used in tasks such as system administration including writing scripts to automate administration, tasks between different systems and system versions, this research is the first to implement path normalisation to perform correlation of digital artifact full path values. Thus, it is not yet known if this technique, although theoretically valid, will be effective at aiding digital artifact detection and also be computationally efficient.}
tailored matching methods to enable correlation; for example, each directory has the file name, normalised file name and allocation status available for matching, whereas data files have file name, normalised file name, allocation status, file size and file hash value (MD5 and SHA1). These metadata properties differ because directories have no actual contents and data files do. Therefore, different matching methods (functions) are needed to correlate different properties. The same is true for Registry keys and values. The following subsections outline the matching methods for file system entries, followed by matching methods for Windows Registry entries.

5.5.3.3 File System Matching Methods

Preliminary system design and informal testing of Vestigium revealed that file system entries require specific matching methods to perform detection of known digital artifacts. File system entries can be broadly classified by their type, either directories or data files. For each FileObject this information is retained by the meta_type element and can either be: 1) A directory specified by the integer value 2 (two); or 2) A data file specified by the integer value 1 (one). Both types require different matching methods due to different metadata properties. The following list specifies a total of five file system matching methods that have been designed and implemented in Vestigium. One matching method has been authored for directories while four methods have been authored for data files. Each matching method indicates the digital artifact metadata properties that are used for correlation. All metadata properties are specified using the DFXML naming conventions (see Table 5.5 for examples).

1) **Directory**: filename_norm, meta_type, alloc_inode, alloc_name
2) **File Soft**: filename_norm, meta_type, alloc_inode, alloc_name
3) **File Hard**: filename_norm, meta_type, filesize, md5/sha1, alloc_inode, alloc_name
4) **File Deleted**: orphan_name, meta_type, filesize, md5/sha1, alloc_inode, alloc_name
5) **File Hash**: meta_type, filesize, sha1, alloc_inode, alloc_name

Each specified file system matching method is achieved by comparison of all metadata properties between two entries of the same type; that is, one entry from the target data set and one entry from the application profile. All correlation is achieved by determining equality of each specified metadata property; for example, when comparing two directories, each must have the same normalised file name, be the same type (directory) and have the same allocation status (deleted or not). If all properties are the same, a match is confirmed.

Multiple data file matching methods are required as there are complex matching scenarios when correlation of every metadata property will fail; for example, an application executable (e.g., TrueCrypt.exe) with a default installation path has a fixed full path and hash value. However, an application configuration or log file has a fixed full path but a variable hash value. This is because a configuration file is typically modified based on the options selected by the user resulting in a variable hash value. A special matching method (file soft) has been included in the prescribed matching methods to match a data file which has all the same
properties as a complete match (file hard) but without the file hash value. This provides the ability to detect fragile digital artifact types with a variable hash value. Additionally, a file deleted method is prescribed. Deleted files require a dedicated matching method due to the technique which fiwalk represents unallocated file system entries where a deleted file has a unique prefix ($OrphanFiles) followed by the data file basename; for example: $OrphanFiles\TrueCrypt.exe. The final matching method is file hash correlation, which only matches the file hash value of the data file without comparing any file system path information.

5.5.3.4 Windows Registry Matching Methods

Windows Registry matching methods are similar to the prescribed file system matching methods. This is because Registry keys are very similar to directories and Registry values are very similar to data files and although Registry entries have different metadata property naming conventions, the properties and behaviour are similar; for example, directories have a file name and allocation status, while Registry keys have a cell path and allocation status. Therefore, a similar matching method should be specified. Each of the designed matching methods correlate the metadata properties specified in Table 5.12. The following list specifies three matching methods for Registry entries, one for keys and two for values.

1) **Key**: cellpath_norm, name_type, alloc
2) **Value Soft**: cellpath_norm, name_type, alloc, data_type
3) **Value Hard**: cellpath_norm, name_type, alloc, data_type, data_raw

Metadata correlation is achieved by comparison of all specified metadata properties for each matching method between two Registry entries of the same type; again, one entry from the target data set and one from the application profile. As specified for file system matching methods all correlation is performed by determining equality of each specified metadata property; for example, when comparing two Registry keys, each must have the same normalised cell path, be the same type (key) and have the same allocation status (deleted or not). If all of these properties are equal, a match is confirmed.

Similar to data file matching methods, multiple Registry value matching methods have been specified. This aids in matching Registry values in a variety of complex scenarios where properties may differ. A Registry value can be used for configuration settings and can store different value data depending on the option selected by the user. A Registry value may store a value to check for application updates. Depending on the option selected by the user, different value data can be stored; for example, if the application should automatically check for updates (value = 1) or if the application should not automatically check for updates (value = 0). This is a similar scenario to data file matching. Therefore, two matching methods are specified. The **value hard** matching method correlates all available metadata properties, including the raw_data contents, while the **value soft** compares the same properties except for the raw_data contents. This is similar to the file hard and file soft matching methods which either include or exclude matching of the file hash property.
The specified matching methods have outlined a total of eight different matching methods for the four digital artifact types. Each matching method specifies different metadata properties that are to be used for correlation. All correlation is achieved by equality of each specified metadata property. The prescribed matching methods provide the functionality to match digital artifacts in a variety of complex scenarios. The actual technique used to compare digital artifacts using the designed matching methods is discussed in the following subsections for file system and Registry matching respectively.

5.5.4 File System Matching

The purpose of file system matching is to detect file system artifacts in the target data set that are present in the application profile. Figure 5.6 displays a high level overview of the process designed to perform file system artifact matching. The two inputs are: 1) The application profile that represents a reference set of known digital artifacts from a profiled application; and 2) The DFXML report that represents the file system entries from the target data set. Both inputs have been previously transformed and stored in metadata reports and are represented as DFXML FileObjects to provide ease of comparison.

The industry standard approach for correlation of file system artifacts is a brute force (or an all-against-all) comparison between the reference data set and target data set. According to Breitinger, Rathgeb, and Baier (2014), the lookup complexity of a single entry from a reference data set can be expressed as $O(n)$, where $n$ denotes the number of entries in the reference data set. This simply means that every digital artifact in a reference set needs to be compared against every digital artifact (of the same type) in the target data set. Due to the increasing size of digital investigation targets, a brute force comparison can be a computationally intensive operation; for example, “the lookup complexity of similarity digests hamper the usage in the field” (Breitinger, Baier, & White, 2014, p. S2). This is due to the problem of comparing two similarity digest values as it is not performed using equality, rather the similarity digests are compared based on the likeness to each other. This is why traditional cryptographic hashing was chosen for the system design. The proposed system design does not implement a brute force comparison. Instead, the application profile is processed and divided into a selection of Python dictionaries to perform lookups based on various matching methods. In addition, matching methods are implemented using previously generated metadata property values (e.g., file name) and correlation is based on equality (e.g., if the two objects are equal). The overall design is envisioned to be computationally feasible.

At the outset of file system artifact matching, the application profile (APXML document) is first processed and stored in a variety of data structures to enable fast metadata property comparison. Next, the metadata report (DFXML document) containing a representation of the target file system is processed. File system entry matching can then be conducted using the prescribed matching methods (see Section 5.5.3.3).

---

28 The computational efficiency of performing equality comparison between multiple digital artifact metadata properties is envisioned to be efficient. However, this requires confirmation by performing efficiency testing during the experimental testing phase of this research.
5.5.4.1 Application Profile Processing

An application profile (APXML document) is parsed using the APXML API, namely the apxml.py, and each entry processed and appended to a selection of Python dictionaries for later processing. Listing 5.5 provides an example of the code implemented in Vestigium that is used to read in multiple APXML documents (profiles) and populating a dictionary with all file system entries (FileObjects). The simplicity and briefness of code is due to the APXML API which provides simple reading and writing of APXML documents.

Each FileObject from a profile (PFO) is extracted and stored in a variety of Python dictionary data structures which maps a metadata property to the actual FileObject. A Python dictionary structure is used because, according to the Python Wiki (2012), the average lookup complexity (Get Item) for a dictionary is $O(1)$. This is more computationally efficient
than other data structures; for example, a Python list data structure has an average lookup complexity of \(O(n)\). For each dictionary a string value is specified for the dictionary key because it is: 1) A unique value; and 2) A Python string object has faster lookup performance as it mitigates error checks when compared to other object types (Python Wiki, 2012).

```python
import sys, collections, apxml, Objects, dfxml

# Specify profile dictionary
pfos_dict = collections.defaultdict(list)

for profile in sys.argv[1:]:
    # Read the APXML document using interparse function
    apxml_obj = apxml.iterparse(profile)

    # For each entry, if it is a FileObject, append to dictionary
    for pfo in apxml_obj:
        if isinstance(pfo, Objects.FileObject):
            pfos_dict[pfo.filename_norm].append(pfo)
```

Listing 5.5: Example of reading multiple APXML documents

A total of three Python dictionaries map different metadata properties to actual file objects. The following dictionaries have been specified which include the dictionary name, key and linked object:

1) **Normalised path**: filename_norm \(\rightarrow\) [FileObject]
2) **Deleted path**: orphan_name \(\rightarrow\) [FileObject]
3) **File hashes**: md5/sha1 \(\rightarrow\) [FileObject]

The first dictionary (normalised path) maps the normalised file path to the FileObject and is utilised for the directory, file soft and file hard matching methods. Since not all file system entries are matched based on an exact normalised full path, other dictionaries are also implemented. The deleted path dictionary maintains a deleted name (orphan_name) lookup for possible deleted file entries, while the file hashes dictionary maintains a hash table lookup. All three dictionaries are used when processing the file system entries from the target data set using the process specified in the following subsection.

### 5.5.4.2 File System Artifact Matching Process

As displayed in Figure 5.6, the first action in the matching process is to read in the DFXML report which provides a metadata representation of the target file system. All file system matching is specified in the Vestigium source file named FileSystemProcessing.py.

The DFXML report is processed by iterating (reading) each file system entry using the `Objects.iterparse()` function. If a previously generated DFXML report is not provided, the same function is still called with the disk image (evidence file) as the target. This parses
the file system and generates FileObjects on-the-fly using the fiwalk tool. In this scenario, a DFXML report is saved to the user-specified output directory as it may be used later for additional processing. Providing support to directly parse the forensic disk image allows Vestigium to operate directly on the evidence source without any pre-processing requirements. Each Target FileObject (or TFO) from the target is processed individually. A simple generic exclusion rule is applied for all file system entries by excluding the following file system navigation entities: 1) Dot dirs that represent the current directory (/ .); or 2) Dot dot dirs that represent the parent directory (/ ..). The TFO file name (filename) is then normalised on-the-fly to produce a normalised full path (filename_norm). Three separate matching tests are then performed for each TFO.

Firstly, the normalised full path is used as a dictionary lookup to identify a matching file system path. If the normalised path is discovered the TFO is further processed. Three of the five file system matching methods are tested (directory, file hard and file soft). If the TFO is a directory, the directory matching method is called, match_dir(), and the normalised file name and allocation status checked for equality. If all match, the TFO is deemed a matching directory and appended to a Python list of matched FileObjects from the target data set. If the TFO is a data file, the file hard and then file soft matching methods are tested. The match_file() function is called and the normalised file name, file size, hash value (MD5 and/or SHA1) and allocation status checked for equality. If all match, the TFO is appended to the list of matches. Secondly, if no match was found in the dictionary of normalised file names and the TFO is not allocated the deleted file dictionary is used for a path lookup. If a matching deleted file name (orphan_name) is found, the TFO is deemed a match. Thirdly, if no match has yet been found, the PFO hash value is used as a lookup to the hash dictionary. If a matching file hash is found the TFO is deemed a match and appended to the results. Finally, if no match has been found, the TFO is deemed to not be a match to any entry in the application profile and the next TFO is subsequently processed.

When a file system artifact is detected and deemed a match to an application profile artifact the FileObject is added to a Python list of matched artifacts that are processed at the end of the file system matching phase. This is later discussed in further detail in the forensic reporting section.

An additional task when performing file system matching is to process each TFO and determine if it is a Registry hive file, and if so, extract the file to allow later processing by the CellXML-Registry tool. Matching a Registry hive file is accomplished by performing a string search on the full path. This is achieved using the HiveExtractor.py module.

5.5.5 Windows Registry Matching

The purpose of Windows Registry matching is to detect Registry artifacts (keys and values) in the hive files from the target data set. Figure 5.7 displays a high-level overview of the process designed to perform Registry artifact matching. Matching Windows Registry entries follows
a similar overall process to file system matching. However, there are a number of differences during initial processing as each target system will have multiple Registry hives. Most Registry entries (keys or values) only reside in a specific hive file; for example, UserAssist values are known to only exist in the NTUSER.DAT hive file. Therefore, it is pointless to search for UserAssist values in the SYSTEM or SOFTWARE hive files. A Registry entry hive file can be determined by inspecting the rootkey property of each CellObject being processed and only the required hive is searched.

Similar to file system artifact matching, two inputs are required to perform Windows Registry matching: 1) The application profile (APXML document) that represents a reference set of known digital artifacts from a profiled application; and 2) A collection of Registry hive files that processed and transformed to RegXML reports that represent each individual hive file from the target data set. Both inputs are stored in the revised RegXML CellObject format (see Listing 5.3). This allows ease of comparison as the profile and target are stored in the same data abstraction.

5.5.5.1 Application Profile Processing

As with file system artifact matching, an application profile must first be processed. The APXML document is parsed using APXML API and each Registry entry (CellObject) is extracted and stored in a Python dictionary. However, only one Python dictionary is implemented that uses the normalised path (cellpath_norm) as the key which maps to the actual CellObject. This is the same technique as implemented when populating dictionary objects for file system matching; a dictionary is utilised to provide an efficient lookup technique and the normalised full path is used as the key because it is a unique string object which provides a more efficient lookup performance.

An additional step performed when processing the application profile(s) is to determine the Registry hives that are required to perform matching. When parsing the application profile a Python set data structure is populated and returned which contains a unique collection of the different Registry hive file rootkeys that are present. This is achieved by extracting the rootkey property from each profile entry and populating into a set named target_hives. Take the example of an application profile which only contains Registry entries from the SOFTWARE and NTUSER.DAT hive files. The resultant set will have a Python set populated with: software and ntuser.dat. This set will only be used during matching to iterate (read) and search Registry hive files where a potential match to the application profile may reside. This process is not necessary to perform Registry matching and is solely designed to reduce the number and type of Registry hives to process and, ultimately, improve system efficiency.

5.5.5.2 Registry Artifact Matching Process

As displayed in Figure 5.7, the first action in Registry entry processing is to read the RegXML documents that provide a metadata representation of the Windows Registry. All Registry matching is specified in the Vestigium source file named RegistryProcessing.py.
Figure 5.7: High-level overview of the Registry matching process designed to automate correlation of Registry entries between the application profile and the target data set.

The information regarding the necessary hive files have already been determined when processing the application profile and provides a set of required hives and any hive file that is not needed is not processed.

Each **Target CellObject** (or **TCO**) from the required RegXML documents are processed. Two variables are specified that track which Registry hive is currently being processed: 1) The **active_rootkey** represents the hive file root key currently being processed (e.g., system, ntuser.dat); and 2) The **active_hive** stores the actual hive file name that has been extracted from the target system (e.g., Users/Terry/NTUSER.DAT.hive). When one RegXML document is finished processing, the two variables are updated. These variables are later used for reporting the correct Registry file that a matched entry was discovered.

For each TCO processed from each RegXML document, the first step is to transform the Registry path (**cellpath**) into the normalised path (**cellpath_norm**). This is divided into two phases. Firstly, the root key is normalised (as depicted in Table 5.10). Secondly, the
actual path is normalised (as depicted in Table 5.11). Additionally, if the Registry entry is a value and contains a file system path (e.g., starts with C:\) it is subjected to file system path normalisation (as depicted in Table 5.9). All of these normalised path components are joined and populated into the normalised path (cellpath_norm) and used as a lookup value to the single Python dictionary. If a match is found the entry is subjected to further processing.

Additional matching is performed based on Registry entry type. Registry keys are subjected to the key match method and the normalised cell path, type and allocation status checked. For a match to be confirmed all three metadata properties much be equal and if so, the entry is appended to a list of matched objects. Registry values are subjected to two matching methods. A value hard matching method correlated the normalised cell path, type, data type, actual data contents and allocation status. Again, if all metadata properties are deemed equal a match is confirmed. The only difference in the value soft matching method is that an equality check for data contents does not need to be confirmed. This provides the functionality to match a Registry value with a variable data content; for example, a Registry value that stores a variable user configuration setting.

When a Windows Registry entry is detected and deemed a match to an artifact in the application profile, the CellObject is added to a Python list of matched artifacts that are processed at the end of the matching phase. This is discussed in further detail in the following section.

5.5.6 Forensic Reporting

Forensic reporting entails the communication of results to the investigator. Vestigium provides two output formats: 1) A human-readable text-based log file; and 2) Machine-readable reports containing file system matches in a DFXML report and Registry matches in a RegXML report.

5.5.6.1 Log File

The goal of the human-readable log file is to convey the tool results and the configuration used during operation. As it is part of the standard library, Vestigium implements the Python logging module to perform event logging. Recorded events include:

- Case information derived from Vestigium command line arguments including: evidence file, output directory, application profile(s), DFXML input file, RegXML input directory and other command line parameters
- Application profile information including name, state and full path for each entry
- Digital artifact matching information including a list of detected and not-detected digital artifacts (with application name, state, full path)
- Time information: Start and end time, processing time for each matching phase, DFXML and RegXML generating time

[29] https://docs.python.org/3.4/library/logging.html
5.5.6.2 DFXML and RegXML Reports

Machine-readable reports are prescribed in the system design as they provide the functionality to: 1) Convey and report detected digital artifacts in a standardised manner; and 2) Provide the functionality for automated post processing. All detected file system and Windows Registry artifacts are exported to a DFXML and RegXML report respectively. The reports are populated with the FileObjects or CellObjects from the target data set that have been correctly matched. Each object has the matched entry from the application profile appended to the original FileObject or CellObject. This provides the functionality to save the detected digital artifact and the associated the profile artifact for potential post-processing.

DFXML and RegXML reports provide efficient post-processing capability by any tool or script that supports the DFXML standard. The availability of the DFXML API provides the functionality to extract the file or Registry entry information; for example, the DFXML report output from Vestigium can be ingested into simple scripts to provide further automated processing such as generating a printed list of all file paths and associated hash values. This information could be used to create a text file hash set. A more complex example is that the DFXML report could be processed and a timeline of the file access dates conveyed to the investigator.

5.5.7 Software Development: Vestigium

The system design requirements specified the need for a forensic tool that could ingest an application profile(s) and a target data set and correlate the entries in each. Digital artifact matching is performed by matching every digital artifact from the target data set against every digital artifact from the application profile. A proof-of-concept software tool, Vestigium, was authored solely for digital artifact matching to provide later demonstration and subsequent evaluation of the system design.

The Vestigium tool uses the Python programming language (version 3.4.0) and is compatible with Microsoft Windows operating systems. Vestigium can be executed on Linux operating systems but lack the functionality to perform Registry matching (as the underlying CellXML-Registry tool is only available on Windows platforms). Vestigium was solely written by the author and contains approximately 3,000 lines of original source code. It is implemented through a variety of Python modules that provide programming code to perform specific tasks. Table 5.13 outlines the implemented Python modules with an accompanying description of functionality.

Vestigium has a collection of dependencies that are required to run the tool. Table 5.14 lists these dependencies including the software name, version, if the package is required to run Vestigium and a description of the usage. Three dependencies are explicitly required: 1) Python as the primary programming language; 2) fiwalk to parse a forensic image, extract file system entries and produce a DFXML report populated with FileObjects; and 3) CellXML-Registry to parse Windows Registry hive files, extract Registry entries and produce RegXML reports populated with CellObjects. Table 5.14 also includes two
Table 5.13: List of Vestigium Python modules

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vestigium.py</td>
<td>Program entry point. Parses command line arguments. Handles file system and Windows Registry matching configuration</td>
</tr>
<tr>
<td>FileSystemMatching.py</td>
<td>Matching engine for file system entries. Performs correlation between the file system entries (FileObjects) from the application profile and the target data set</td>
</tr>
<tr>
<td>RegistryMatching.py</td>
<td>Matching engine for Windows Registry entries. Performs correlation between the Registry entries (CellObjects) from the application profile and the target data set</td>
</tr>
<tr>
<td>FilePathNormaliser.py</td>
<td>Performs path normalisation of a file system entry</td>
</tr>
<tr>
<td>CellPathNormaliser.py</td>
<td>Performs path normalisation of a Windows Registry entry</td>
</tr>
</tbody>
</table>

non-required packages for reading forensic disk image formats: 1) The AFFLIB library for Advanced Forensic Format (AFF) support; and 2) The libewf library for Expert Witness Compression Format (EWF) support. Both of these libraries are not essential to run the Vestigium tool, but provide the support to process a target disk image that is stored in .aff and .E01 disk image formats. Therefore, the inclusion of these two libraries is advantageous as both disk image formats are regularly used in digital investigations [Flaglien et al., 2011].

Table 5.14: List of Vestigium software package dependencies. Required dependencies are marked with a tick (✓) while non-required dependencies are marked with a cross (✗)

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Required</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3.4.0</td>
<td>✓</td>
<td>Primary programming language</td>
</tr>
<tr>
<td>fiwalk</td>
<td>0.6.9</td>
<td>✓</td>
<td>Parse the file system of the target data set and produce a DFXML report</td>
</tr>
<tr>
<td>CellXML-Registry</td>
<td>1.2.0</td>
<td>✓</td>
<td>Parse and Registry hive files and produce RegXML reports</td>
</tr>
<tr>
<td>afflib</td>
<td>3.7.4</td>
<td>✗</td>
<td>Support for the Advanced Forensic Format (AFF) disk image format</td>
</tr>
<tr>
<td>libewf</td>
<td>20140608</td>
<td>✗</td>
<td>Support for the Expert Witness Compression Format (EWF) disk image format</td>
</tr>
</tbody>
</table>

5.5.8 Digital Artifact Matching Summary

The design requirements to perform digital artifact matching between the application profile(s) and the target data set bought about the introduction of a proof-of-concept forensic analysis tool named Vestigium. A collection of matching strategies were designed to perform correlation of digital artifacts in complex detection scenarios and implemented in the Vestigium tool. A novel technique to perform path normalisation was proposed and implemented in.
the tool to provide full path correlation in complex matching scenarios. Communication of any results detected during digital artifact matching was achieved by implementing a human-readable log file, while a machine-readable data abstraction was leveraged to export results for potential post-processing.

5.6 Conclusion

The goal of the third element of the DSRM process model was to design an artifact by specifying the functional requirements to solve the identified research objectives from Chapter 4. A system architecture formed and divided into two main stages to ease the design process which each had two components. The total four system architecture components were each subjected to requirement specification, design and subsequent practical implementation. The ultimate outcome has resulted in a system design theoretically capable of automating the detection of known interesting digital artifacts on a target data set using application profiles. However, to prove that the system design is functional, it needs to be subjected to demonstration, the next element in the DSRM process model.

Chapter 6 therefore proceeds from the progress made thus far. Each component of the architecture is initially demonstrated to display an overview of software usage with examples. The implemented system will then be demonstrated to prove that it can solve the specified research objectives in terms of functionality in a lab controlled environment. Each component of the system design is to undergo demonstration in an established experimental testing environment. Firstly, application profile creation is demonstrated using the LiveDiff tool and various data reduction techniques tested in an attempt to remove irrelevant operating system noise from the authored profiles. Secondly, the created profiles are to be tested against a selection of data sets with known content to demonstrate functionality of the Vestigium tool to automate matching of known content.
The Design Science Research Methodology (DSRM) process model serves as the framework for this research. Peffers et al. (2007) state that the demonstration element of the DSRM process model involves validating that the designed and implemented artifact can solve one or more of the research problems and associated objectives (see Section 4.2.1 and 4.2.2). The research objective specified for the effective and efficient detection of digital artifacts from application software using automated reference set creation and subsequent automated correlation against a target data set. A system was designed based on specified requirements and implemented in the form of various designed forensic analysis tools (see Chapter 5). Chapter 6 now demonstrates the implemented system to determine the effectiveness and efficiency in a controlled laboratory testing environment, while the following chapter (Chapter 7) considers the utility of the system in a selection of more typical non-laboratory investigation scenarios.

This chapter begins with an overview of the implemented software based on the system design including tool operation. The LiveDiff tool is first demonstrated by creating application profiles for three selected anti-forensic tools followed by investigation of a variety of data reduction techniques to remove irrelevant results from the created application profiles. The Vestigium tool is then used to perform digital artifact matching between the created profiles and a target data set. The target data set is comprised of known content, authored to verify the system design and to demonstrate functionality. New initiatives arising from the results of testing are then acted on and system refinements conducted as per the cyclical nature of the design science approach.
6.1 Tool Operation

The practical implementation of the system design is in the form of computer software, specifically forensic analysis tools. Two separate tools were implemented which map to the two main stages in the system architecture:

1) **LiveDiff**: A portable live Microsoft Windows tool that automates application profile creation by executing differential forensic analysis (a form of reverse engineering) to determine operating system-level changes throughout the life cycle of an application (see Section 5.2)

2) **Vestigium**: A forensic analysis tool that performs automated processing of a target data set and subsequent correlation of digital artifacts between an application profile and target data set (see Section 5.5)

In addition to these two pivotal tools, other components and associated software were authored to achieve the required functionality specified by the system design requirements. This resulted in the design and implementation of two additional computer software components:

3) **apxml.py**: An Application Programming Interface (API) written in Python to parse and automate processing of the prescribed Application Profile XML (APXML) data abstraction (see Section 5.3.4)

4) **CellXML-Registry**: A portable Microsoft Windows tool that parses offline Registry hive files and extracts entries, recovers deleted entries and produces a RegXML report to represent the original evidence source (see Section 5.4.3)

Figure 6.1 displays a high-level overview of the implemented system design. The first main stage of the system architecture begins with the identification of digital artifacts. As illustrated, application software is used as input to the LiveDiff tool with four examples shown, including: 1) The Chromium web browser; 2) Mozilla Firefox web browser; 3) The Tor anonymous web browser; and 4) Mozilla Thunderbird email client. The selected application is reverse engineered using differential forensic analysis provided by the LiveDiff tool and the results (identified digital artifacts) are populated into an Application Profile XML (APXML) document.

The second main stage of the system architecture begins with target data set processing, followed by matching of digital artifacts between the created application profile and target data set. To achieve matching, the application profile and target data set are used as input to the Vestigium tool. Correlation is performed between the two inputs and matched digital artifacts are reported to the investigator in the form of human-readable and machine-readable data abstractions.

The following subsections outline the operation of LiveDiff, the apxml.py API, the CellXML-Registry tool and Vestigium tools.
6.1.1 Creating Application Profiles: **LiveDiff**

The first stage of the system design centres on the creation of application profiles. This involves identification and documentation of digital artifacts including file system directories and data files and Windows Registry keys and values. The specified digital artifacts are uniquely associated with a particular application and used to populate a reference set for application software, named application profiles. The resultant output from this process is an application profile stored in an Application Profile XML (APXML) data abstraction which is populated with known application software artifacts and used to automate application detection on an investigation target.

A new tool was authored to create application profiles according to the functionality required by the system design (see Section 5.2). To reiterate, the system design requirements specified the need for a tool that could run on a live Microsoft Windows operating system, functionality to automate collection and comparison of system snapshots (file system and Registry entries) and the capability to output results to the prescribed APXML document structure. The resultant tool was named **LiveDiff**, after the reverse engineering technique that it implements: **live differential forensic analysis**. LiveDiff is authored using the C programming language and released under the GNU Lesser Public License Version 2.1. The LiveDiff tool is approximately 6,000 lines of source code, with approximately 1,500 lines sourced directly from the Regshot project (on which LiveDiff is based on). LiveDiff source code is available from:

```
https://github.com/thomaslaurenson/LiveDiff
```

Compiled binaries for Microsoft Windows (e.g., LiveDiff.exe) are available from:

```
https://github.com/thomaslaurenson/LiveDiff/releases
```

LiveDiff is a console application and, as such, a variety of command line arguments are
available to control what operations the tool performs. All command line arguments are specified in the livediff.c source code file, the main program entry point. LiveDiff can be executed using the Windows Command Prompt (or cmd.exe). It is recommended, although not essential, that the Command Prompt is started using administrator permissions. Administrator privilege is advantageous because the system scanning process requires elevated permissions to access and read the entire file system and Windows Registry. According to Microsoft (2010), a Command Prompt can be invoked as administrator using the actions displayed in Listing 6.1.

Start Menu > All Programs > Accessories
Right-click Command Prompt and click Run as Administrator
NOTE: If the User Account Control dialog box appears click Continue.

Listing 6.1: Steps to invoke Command Prompt with administrator rights

Once a command prompt with administrator privileges has been started a user must navigate to the directory with the LiveDiff executable (e.g., LiveDiff.exe). The following command executes LiveDiff and prints the help menu to the user:

```
$ LiveDiff.exe -h
```

Figure 6.2 shows the LiveDiff menu that is displayed to the user when executing the above command. The help menu describes a variety of configuration options available to the user that can be selected when running LiveDiff.

Figure 6.2: LiveDiff program help menu

---

1When LiveDiff is executed in a Command Prompt without administrator privileges a number of file system and Registry entries cannot be accessed and, therefore, are not included in the system snapshots or differential analysis processing; for example, there are some portions of the Windows root folder structure that cannot be accessed without the appropriate system rights.
Table 6.1 displays a complete list of all command line arguments that the LiveDiff tool accepts and an associated description of each argument.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code></td>
<td>Display the help menu</td>
</tr>
<tr>
<td><code>-help</code></td>
<td>Display the help menu</td>
</tr>
<tr>
<td><code>/?</code></td>
<td>Display the help menu</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>Boolean argument to toggle saving of snapshot files [Default = FALSE]</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Boolean argument to toggle inclusion of the dynamic blacklist when performing snapshots [Default = FALSE]</td>
</tr>
<tr>
<td><code>-profile</code></td>
<td>Default mode of operation which implements a looped scanning, comparison and output process to perform multiple differential analysis runs</td>
</tr>
<tr>
<td><code>-profile-reboot</code></td>
<td>Mode of operation to finish creating an application profile after a system reboot</td>
</tr>
<tr>
<td><code>-load</code></td>
<td>Mode of operation to request loading of previously saved snapshot files which requires one or two snapshot file locations</td>
</tr>
</tbody>
</table>

The LiveDiff tool has three modes of operation each designed to perform a specific tasks: 1) Profile mode (default); 2) Profile reboot mode; and 3) Loading mode. However, the profile mode of operation (`-profile`) is key to the generation of application profiles by implementing a looped snapshot and comparison cycle to collect data from all application life cycle phases (e.g., install, open, close, uninstall and reboot). This strategy simplifies application profile creation by using an automated data collection and reporting method (see Figure 5.3). The resultant output from the profile mode is a populated Application Profile XML (APXML) document containing file system and Registry entries associated with the application software being profiled. By default, the profile mode includes dynamic blacklisting and file hashing using the MD5 and SHA1 algorithms and therefore does not require additional command line arguments.

The profile reboot mode of operation (`-profile-reboot`) is provided to continue the data collection procedure after a system reboot. This mode re-loads the previously created dynamic blacklist into memory and continues populating the previously created APXML document. The following command invokes LiveDiff after a system reboot and automatically loads the previously saved snapshot:

```
$ LiveDiff.exe --profile-reboot
```

The load mode of operation (`-load`) provides the functionality to load one or two previously saved snapshot files. If one snapshot file is provided the file is loaded and the second snapshot is then captured on the live system (this is useful for collecting snapshots before and after
a system reboot). If two snapshot files are provided both are loaded and then compared (this is useful for tool debugging and development). In both scenarios the differences between snapshots are reported to the user in an APXML document. The following command line argument executes LiveDiff and loads two previously saved snapshot files (1.shot and 2.shot) and then compares both files:

```bash
$ LiveDiff.exe --load 1.shot 2.shot
```

The different modes of operation provide a variety of program functionality when performing differential analysis. Each mode provides the necessary components to perform experimental testing and demonstrate tool functionality. However, the profile mode of operation is the recommended method to use when performing differential forensic analysis of application software. This mode was specifically designed to implement a data collection strategy to determine system-level changes throughout the entire life cycle of an application with minimal user intervention. The output from the profile mode is an APXML document with all digital artifacts deemed new, modified, changed and/or deleted that are classified based on the application life cycle phase. The APXML document can then be directly used as input to digital artifact matching against a target data set.

### 6.1.2 Processing Application Profiles: `apxml.py`

In order to simplify application profile processing an API was written, named `apxml.py`, with the ability to automate processing of Application Profile XML (APXML) documents. Simple scripts can therefore be created to read, write and manipulate the contents of APXML documents. The API was written in Python primarily to integrate with the existing DFXML Python APIs, namely the original `dfxml.py` and newer `Objects.py` modules, as well as to provide compatibility with the components built and software developed in this research. The `apxml.py` module is approximately 700 lines of original code written specifically for this research. The `apxml.py` module is released under the GNU Lesser Public License Version 2.1 and is available from:

[https://github.com/thomaslaurenson/apxml](https://github.com/thomaslaurenson/apxml)

An APXML document can be read using the simple Python script displayed in Listing 6.2. The first step is to import the `apxml.py` module. The APXML document is then read using `apxml.iterparse()` function with the specified target file name. In the example script, the application profile file name is `TrueCrypt.apxml` and resides in the current working directory. The function returns a populated APXMLObject stored in the `apxml_obj` variable and contains Python objects which represents all of the information in the application profile. The final portion of the script loops over all profile entries, determines if each entry is a data file (if the Python object is an instance of the `FileObject` type and `meta_type` of 1) and, if so, prints the file name and SHA1 hash.

---

2 See: [https://github.com/simsong/dfxml/tree/master/python](https://github.com/simsong/dfxml/tree/master/python)
### Import APXML module

```python
import apxml
```

### Read the APXML document using `iterparse`

```python
apxml_obj = apxml.iterparse("TrueCrypt.apxml")
```

### Loop through each object and print file name and SHA-1 hash value

```python
for fi in apxml_obj:
    if isinstance(fi, Objects.FileObject) and fi.meta_type == 1:
        print(fi.filename, fi.sha1)
```

Listing 6.2: Functional example of the APXML API

The APXML API also has a statistics function to facilitate generation of statistics for experimental testing. Listing 6.3 displays a short Python script that prints the total number of files, the total number of Registry keys and the number of files created during the application installation phase using the `StatisticsObject` class implementation.

```python
# Import APXML module
import apxml

# Read the APXML document using `iterparse`
apxml_obj = apxml.iterparse('TrueCrypt.apxml')

# Generate statistics about the APXML document
apxml.generate_stats(apxml_obj)

# Print a collection of statistics
print('Total Files : ' % apxml_obj.stats.files)
print('Total Reg Keys : ' % apxml_obj.stats.keys)
print('Installed Files : ' % apxml_obj.stats.fs_state['install'])
```

Listing 6.3: Example of the APXML API statistics function

### 6.1.3 Target Data Set Processing: CellXML-Registry

A software tool, named CellXML-Registry, was authored (based on, and using the functionality provided, by the Registry parser library) to generate a metadata representation of Windows Registry hive files to fulfill the following requirements: 1) Represent Registry entries using the revised RegXML CellObjects format; and 2) Provide functionality to recover deleted Registry entries. CellXML-Registry uses the Registry parser project as the underlying Registry processing library. The tool is written in the C# programming language, contains approximately 600 lines of new code. The tool is released under the MIT license (the same license as the Registry parser project) and available from:

https://github.com/thomaslaurenson/CellXML-Registry
Compiled binaries for Windows (e.g., CellXML-Registry.exe) are available from:

https://github.com/thomaslaurenson/CellXML-Registry/releases

CellXML-Registry provides the functionality to parse offline Registry hives, process each Registry entry and potentially recover deleted entries. A RegXML report is produced which represents the original evidence source. CellXML-Registry is a console application and provides a variety of arguments to control tool operation as displayed in Table 6.2.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Display the help menu</td>
</tr>
<tr>
<td>-f</td>
<td>Specify an offline Registry hive file to process</td>
</tr>
<tr>
<td>-d</td>
<td>Specify a directory of offline Registry hive files to process</td>
</tr>
<tr>
<td>-r</td>
<td>Recover deleted Registry entries [Default = FALSE]</td>
</tr>
<tr>
<td>-v</td>
<td>Specify log verbosity: 0 = Info, 1 = Debug, 2 = Trace [Default = 0]</td>
</tr>
</tbody>
</table>

Table 6.2: List of command line arguments for the CellXML-Registry tool

CellXML-Registry is used by the Vestigium tool to process the target data set before digital artifact matching is performed. As Vestigium automates all target data set processing, the investigator is not required to manually execute the CellXML-Registry tool. However, the tool can be used to generate RegXML metadata reports for ingestion to other forensic tools that are able to parse the RegXML format, or alternatively, simply search the document using utilities such as grep. Furthermore, support was added to the existing DFXML Objects.py API to process the revised RegXML structure produced by CellXML-Registry. Similar to the apxml.py module, simple Python scripts can be written to perform processing and analysis.

6.1.4 Digital Artifact Matching: Vestigium

The second main stage of the system design is the matching of digital artifacts between the application profile(s) and target data set. The resultant output from this process are forensic reports containing digital artifact matches; that is, the digital artifacts that are present in both the application profile and the target data set. The reports therefore document known and relevant application software artifacts that are of forensic interest.

In order to perform matching between the application profile and target data set a new tool was designed and implemented based on the system requirements. The tool was named Vestigium, from Latin meaning footprint or trace. This name is highly appropriate for the system design which attempts to determine the footprint that application software leaves on a computer system. Vestigium is authored using the Python programming language and is comprised of approximately 2500 lines of original source code. The tool is released under the GNU Lesser Public License Version 2.1 and available from:

https://github.com/thomaslaurenson/Vestigium
Provided that the git version control system is available on an investigator’s system, the Vestigium tool and all dependencies for the Microsoft Windows system can be downloaded to the current working directory using the `clone` utility as displayed below:

```
$ git clone https://github.com/thomaslaurenson/Vestigium.git
```

Vestigium was designed as a console application. As such, a variety of command line arguments are available to control what operations the tool performs. The command line arguments are specified and handled by the `Vestigium.py` source code file which acts as the main program entry point. Similar to LiveDiff, the Vestigium tool can be executed using the Windows Command Prompt (or `cmd.exe`). Being dependant on the CellXML-Registry tool, which is only available on the Windows platform, Vestigium was designed to operate on the Microsoft Windows operating system. However, Vestigium can be used on Linux, but then provides no support for Registry processing. Administrator privileges are not required to run the tool. The following command invokes Vestigium using Python (version 3.4) and prints the program help menu:

```
$ C:\Python34\python.exe Vestigium.py -h
```

Figure 6.3 shows the Vestigium help menu that is displayed to the user when executing the above command. The help menu displays a variety of required arguments and configuration options available to the user.

![Figure 6.3: Vestigium program help menu](image-url)
Vestigium requires three positional command line parameters that must be provided by the user, including: 1) A forensic disk image file (evidence file); 2) A user specified output directory; and 3) One or more previously authored application profiles. Table 6.3 displays a complete list of all command line arguments for Vestigium with an accompanying description of usage. The top of the table displays arguments to print the program help menu, the middle displays mandatory positional arguments and the bottom displays optional arguments to further control tool operation behaviour.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>Display the help menu</td>
</tr>
<tr>
<td>-help</td>
<td>Display the help menu</td>
</tr>
<tr>
<td>imagefile</td>
<td>Forensic disk image (evidence file)</td>
</tr>
<tr>
<td>outputdir</td>
<td>Target directory for all tool output and reports</td>
</tr>
<tr>
<td>profiles</td>
<td>One or more application profiles</td>
</tr>
<tr>
<td>-dfxml</td>
<td>Specify a DFXML report generated using the fiwalk tool</td>
</tr>
<tr>
<td>-regxml</td>
<td>Specify previously generated RegXML reports</td>
</tr>
<tr>
<td>-d</td>
<td>Do not remove files ending in '/.' and '/..' [Default = FALSE]</td>
</tr>
<tr>
<td>-t</td>
<td>Report timestamps in UNIX time format [Default = FALSE]</td>
</tr>
<tr>
<td>-z</td>
<td>Zap (delete) a output directory if it already exists [Default = FALSE]</td>
</tr>
</tbody>
</table>

An example of a command to invoke Vestigium against a target forensic disk image named CaseOne.E01 is shown in Listing 6.4. Three mandatory positional arguments are specified, including: 1) The target forensic disk image (CaseOne.E01); 2) A user-specified output directory (C:\CaseOneOutput); and 3) A total of three application profiles for the CCleaner, Eraser and TrueCrypt anti-forensic tools. The listing specifies new line characters (\^) and comments (REM) using Windows command conventions.

```
$ C:\Python34\python.exe Vestigium.py ^
D:\CaseOne.E01 ^ REM Forensic disk image
D:\CaseOneOutput ^ REM Output folder
CCleaner-5.09-6.1.7601.apxml ^ REM CCleaner profile
Eraser-6.2.0.2970-6.1.7601.apxml ^ REM Eraser profile
TrueCrypt-7.1a-6.1.7601.apxml REM TrueCrypt profile
```

Listing 6.4: Command line example to invoke Vestigium

Listing 6.5 provides an example of supplementary command line arguments and specifies a previously generated DFXML report (CaseOne.xml) and a directory of previously generated RegXML reports (D:\CaseOneHives). Again, the same Windows command conventions are used for newline characters and comments.
6.1.5 System Implementation Summary

The operation of the two main tools (LiveDiff and Vestigium), plus one supplementary tool (CellXML-Registry) and one API (apxml.py) have been implemented using the system design and each operation has been described. LiveDiff provides an automated technique to reverse engineer the initial application software input using differential forensic analysis. The application life cycle is recreated and LiveDiff identifies digital artifacts that are the result of the system changes during different life cycle phases. The associated metadata from each individual digital artifact is extracted and populated into the specifically designed Application Profile XML (APXML) forensic data abstraction.

The Vestigium tool takes an authored application profile and a target data set as input. The target data set is processed and correlated against the APXML profile to determine digital artifacts that reside in both the application profile and the target data set. When a digital artifact is deemed a match Vestigium reports it to the investigator using a hybrid DFXML report, as well as documenting tool operation in a text based log file.

The designed and implemented software tools provide a practical solution to feasibly perform automated forensic analysis. Experimental testing in a controlled laboratory environment is now required to demonstrate functionality and determine if the system design meets the specified objectives of a solution (see Section 4.2.2).

6.2 Experimental Testing

Experimental testing demands a robust method to perform data generation and collection. Such a testing environment is provided using Virtual Machines (VM) which is both rapid and contaminant-free. A method to perform data generation is specified which involves recreating the application profile life cycle. This provides the ability to subsequently collect data to allow the creation of application profiles. It also allows the creation of known data sets to be used as the target data set for system demonstration. A method for determining the ground truth (baseline information) of digital artifact presence on the specified testing data sets is also outlined. Finally, an overview of system effectiveness and efficiency metrics are covered including information on score interpretation.
6.2.1 Virtual Machine Testing Environment

The rigour of experimental testing calls for a documented, controlled and secure testing environment. Virtual Machines (VM) were therefore implemented to satisfy these requirements. VMs provide an excellent platform to carry out experimental testing as implementation is simple and time-effective compared to a physical system. This is important, especially when multiple experimental runs are necessary. Another advantage is that VMs guarantee a testing environment that is free from contaminants. The VirtualBox software by Oracle was selected to create the VM testing environment being open source with a GPL license providing accessibility and, therefore, research reproducibility. VirtualBox is a general purpose virtualisation software giving the necessary functionality to create and modify VMs.

6.2.2 Data Generation: Recreating the Application Life Cycle

The application life cycle is a chronological progression in the life span of an application and the changes the software makes to the host operating system (see Section 2.4.1 and Figure 2.3). It is important that the recreated life cycle phases are strictly specified outlining how the resultant data was generated and collected as the same procedure must be followed for creating application profiles and known data sets. Furthermore, later system evaluation will compare this research to other solutions and the same life cycle phases must be performed to enable a fair comparison.

The selected application life cycle phases were replicated from those used in the National Institute of Standard and Technology (NIST) Diskprint project (Laamanen & Nelson, 2014) which is similar to this research. However, it only focusses on file system entries (primarily data files), whereas this project includes file system and Registry entries. Another significant difference, is that the data collection technique used to perform identification of application software artifacts in this research is performed on a live system, while the Diskprint project performs post-mortem data collection by cloning the VM disk. In spite of this, the Diskprint project will serve as a useful experimental testing baseline against which to compare results. Table 6.4 displays a total of five application life cycle phases specified for testing with an associated description of how each phase was conducted using the selected application.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install</td>
<td>Copy application installer from shared folder to the Windows Desktop, launch the application installer and install using default options</td>
</tr>
<tr>
<td>Open</td>
<td>Run the application software. Order of precedence for running the software is: 1) Windows Desktop link file; 2) Windows Start Menu link file; and 3) Run directly from the Windows Command Prompt</td>
</tr>
<tr>
<td>Close</td>
<td>Close the application. Where applicable ensure any application related services or processes are also exited. This is predominantly achieved by quitting the application using the Windows notification area on the taskbar</td>
</tr>
<tr>
<td>Uninstall</td>
<td>Removal of application software using the Programs and Features interface available in the Microsoft Windows Control Panel</td>
</tr>
<tr>
<td>Reboot</td>
<td>Reboot the system using the restart option in the Windows Start Menu</td>
</tr>
</tbody>
</table>
Figure 6.4 illustrates the application life cycle with visual examples of various file system and Windows Registry changes. The diagram uses the Firefox browser to describe the application life cycle phases. The ground truth phase is representative of a clean Windows 7 installation. The installation of an application (Firefox) creates directories, executable files (.exe), library files (.dll), system files (.sys) and a variety of Registry entries (keys and values). Opening the application creates additional artifacts (e.g., additional folders for configuration settings). Closing the application creates more artifacts (e.g., creation of configuration files). Uninstallation of the application results in most artifacts being removed. However, artifacts may still reside in the Recycle Bin, Registry and some files may not be completely removed. Finally, rebooting the system may remove some residual digital artifacts (e.g., files not removed during uninstallation as they were being used or the application uninstaller was not configured to remove them).

To achieve data generation to recreate the application life cycle phases the VM testing environment was leveraged. A VM was created and installed with Microsoft Windows 7 32-bit which was selected due to its popularity as, according to NetApplications (2015), it controls approximately 56% of the total desktop operating system market share. VirtualBox (version 5.10) was used. The created VM is a ground truth data set from which to generate additional VMs. The ground truth VM was produced and a new disk created using the Virtual

---

3See: https://www.virtualbox.org/
4A Ground Truth image is also known as a baseline image from which to compare future system changes.
Machine Disk (VMDK) format. Microsoft Windows 7 32-bit was installed on the VM and a new user named forensic was added. All other options were left at default values. After installation a small number of modifications were made; for example, Windows Updates were disabled in an attempt to reduce system noise (see Section 3.2.3.3). The VirtualBox Guest Additions were installed on the guest VM to provide the functionality to mount a shared folder between the host system and guest VM. The shared folder was used to copy data between the guest and host operating system.

The implemented testing environment provides an easy method of data generation from which to author an application profile and to create test data sets with known content.

6.2.3 Data Collection: Creating Application Profiles

The data generation technique recreates the application life cycle phases and then, using the VM testing environment, the LiveDiff tool can be executed and data collection performed. The resultant output is a populated application profile stored in the APXML data abstraction. The APXML structure supports any application life cycle phase and is stored in an additional metadata property (XML element) named app_state (see Table 5.4). This information is populated in the XML tag for each FileObject or CellObject by the LiveDiff tool and is specified by the user when performing data collection.

A key benefit to implementing VMs for testing is the speed and convenience of performing experiments on a guest operating system. An efficient method is essential to performing data collection, as creating a fresh VM for each application is time-consuming. Additionally, the testing environment should provide the same VM state for all testing scenarios to ensure consistent results. To accomplish this the ground truth VM (named Win7-32-GT) was cloned into a new VM specifically for application profile creation. The disk of the new VM was configured as an immutable disk (Oracle Corporation [2015]), which is a read-only virtual disk where all modifications made when running are stored in volatile memory and discarded after powering off the VM. Immutable disks are transparent to the user and the guest operating system functions the same as a normal VM.

Listing 6.6 displays an example of the command used to change a normal virtual disk to immutable by using the VBoxManage tool with the modifyhd parameter. In the example, the VM disk name is Win7-32-LD (LD represents LiveDiff) and is purpose-designed to create application profiles. The listing specifies new line characters (^) and comments (REM) using Windows command conventions.

Listing 6.6: Example of method to create an immutable virtual disk

Immutable disks are useful when creating application profiles with LiveDiff as each appli-
cation can be profiled on a clean system without the need to reinstall an operating system for each testing scenario. This minimises the time required to perform experimental testing. It also allows each test to be performed on the same system, thus, removing the variability of the operating system state from experimental testing.

Using the immutable VM, the prescribed data generation method (see Section 6.2.2) is to be performed for each selected anti-forensic tool. In particular, the LiveDiff tool is used to collect a snapshot before and after one application life cycle phase (see Table 6.4). The output is then populated to an APXML document with the digital artifact changes that have occurred during each application life cycle phase (e.g., files created from installing the application).

### 6.2.4 Data Collection: Creating Known Data Sets

The VM testing environment for creating known data sets differs from that used when creating application profiles. Immutable disk images cannot be used because known data set generation is accomplished by collecting a forensic image of the virtual disk. Since immutable disks make no changes to the virtual disk, they cannot be implemented. Therefore the default normal virtual disk type is required.

Known data sets are created by cloning the original ground truth VM into additional VMs as needed using the VirtualBox VBoxManage tool. A new cloned VM is required for each anti-forensic tool tested. Listing 6.7 demonstrates how the ground truth VM (named Win7-32-GT) was cloned using the VBoxManage tool with the clonevm argument.

```
$ cd C:\Program Files\Oracle\VirtualBox
$ VBoxManage.exe clonevm                  REM VirtualBox tool
   D:\Win7-32\Win7-32-GT.vmdk               REM Location of VM disk
   D:\Win7-32-clone\Win7-32-TC              REM Specifies new VM for TrueCrypt
   --register                               REM Add VM to VirtualBox
```

Listing 6.7: Example of method used to clone virtual disks

Each cloned VM is used to recreate the application life cycle for a given application. At the end of each life cycle phase, data collection was conducted by pausing the VM and collecting a forensic image of the virtual hard disk using the dc3dd5 tool. The dc3dd tool was implemented as it provides an automated forensic acquisition process (using a bash script), maintains evidence integrity using cryptographic hashing and has been tested and approved by the National Institute of Justice (NIJ) under the Computer Forensic Tool Testing (CFTT) project (NIJ, 2011).

The previously specified application life cycle phases (see Table 6.4) were recreated on each cloned VM. This process generates the required data for each forensic disk image in the known data set. The first stage of known data set generation is to clone the ground truth VM. This new VM is then started and the application installer is copied from the VirtualBox

---

shared folder to the guest operating system. The application is installed using default options. Installation results in the creation of various files, directories and Registry entries (see Figure 6.5). The VM is paused and a forensic disk image (in the .raw format) of the VM disk file is acquired using the dc3dd tool. The VM state is resumed, the application opened, the VM is paused again and a further forensic image created. Additional digital artifacts are created in the form of files, directories and Registry entries. The VM is resumed, the application software closed, the VM paused and another forensic disk image acquired. The VM is resumed and the application software is uninstalled using the Programs and Features interface. Various files, directories and Registry entries are removed from the system. However, many of these digital artifacts remain on the system in an unallocated state. The VM is paused and another forensic image acquired. Each profiled application resulted in a total of five forensic images: 1) Install; 2) Open; 3) Close; 4) Uninstall; and 5) Reboot. Figure 6.5 visually illustrates the ground truth data set generation method based on the prescribed application life cycle phases specified in this research.

Figure 6.5: High-level overview of the known data set generation method (Image sources: Mozilla Firefox icon taken from [The Mozilla Foundation](https://www.mozilla.org/en-US/) (2013). All other images are public domain or the author’s original work.)

### 6.2.5 Establishing Data Set Ground Truth

A data set with unknown content is problematic when performing system demonstration and evaluation as there is no ground truth, or baseline, on which to accurately and reliably measure the effectiveness of the implemented system design; for example, it is unknown if an application is present, or absent, on a specific target disk image. Furthermore, if an application is present, the number of digital artifacts associated with the application is also unknown. Therefore, in order to determine the effectiveness of the system, the ground truth needs to be established to allow comparison of the results from digital artifact matching.

The required ground truth information includes: 1) Establishing the presence (or absence) of the anti-forensic tools selected for testing; and 2) Establishing the digital artifacts uniquely associated with an anti-forensic tool on each target data set. Basically, determining ground truth should provide a list of all digital artifacts (directories, data files and Registry key and values) that are uniquely associated with the application that also reside on the target data set. The ground truth can be determined by using the contents of the application profile, but
only if the following two parameters are constant: 1) The same Microsoft Windows version; and 2) The same anti-forensic tool version. Given these two constants, the target data set should have the same contents as the application profile; for example, consider the installation phase of the TrueCrypt tool, the same digital artifacts should appear in both the application profile and target data set.

In order to establish the ground truth of data sets with unknown content, certain requirements first need to be specified to initiate a suitable an accurate solution which will now be described.

6.2.5.1 Ground Truth Requirements

For each of the forensic disk images to be used for experimental testing, the following ground truth requirements must be determined:

1) Identify Microsoft Windows operating system version
2) Determine anti-forensic tool presence
3) Determine anti-forensic tool artifacts (if tool present):
   a) Determine anti-forensic tool version
   b) Reverse engineer anti-forensic tool (using LiveDiff) to establish associated file system and Registry entries
   c) Document results in a machine-readable data abstraction to ease storage, processing and distribution to aid this research as well as future research and development

Three distinct phases of chronological investigation are therefore prescribed to determine anti-forensic tool ground truth on each data set. These phases will now be separately outlined.

6.2.5.2 Ground Truth: Determining Windows Version

The first phase of ground truth determination is to establish the Microsoft Windows version on the target forensic disk image. Two methods can be utilised: 1) Reviewing the documentation provided with the data set (if available); and 2) Forensic analysis of the forensic disk image to indicate operating system information. This is a straightforward task if documentation is available. However, without this information a manual investigation is essential being that forensic tools are not available to ascertain Windows operating system versions. [Forensic Wiki](2012) outline the following chronological steps to detect the operating system for a potential Microsoft Windows system:

1) Check root file system for a windows or winnt directory
2) Extract the SOFTWARE Registry hive file
3) Locate the following Registry key: Microsoft\Windows NT\CurrentVersion
4) Examine the following Registry values: ProductName, CSDVersion, ProductId

Since this research promotes automated forensic analysis, a manual investigation to determine the Microsoft Windows version seems cumbersome. Therefore, a Python script,
named DetectWindows.py, was authored to perform the steps outlined above with an automated ability to report if the target system has Microsoft Windows installed and the version present. The script utilises the data abstractions (DFXML and RegXML) and tools developed (CellXML-Registry) in this research. However, detailed discussion of the script operation is beyond the scope of this section, so the method and the full source code is available in Appendix C.1. If a valid Microsoft Windows operating system installation is discovered on the target forensic disk image, the desired anti-forensic tool can then be searched for.

6.2.5.3 Ground Truth: Determining Anti-forensic Presence

The second phase of ground truth determination is to confirm the presence, or absence, of the anti-forensic tool of interest. An existing method is not available for determining application software presence on a target forensic disk image, therefore, this research proposes the following:

1) Determine anti-forensic tool presence using string searching with unique keywords
2) Determine anti-forensic tool version using manual analysis

Establishing the presence of an anti-forensic tool can be accomplished by performing a text-based string search of all forensic disk image contents. This can be achieved by invoking the strings utility with the data set disk image as input. It is proposed that the anti-forensic tool name (e.g., ccleaner, eraser and truecrypt) can be used as the keywords in this research. The method may sometimes yield false positive search hits if the keyword is not unique. However, a preliminary test using a default Microsoft Windows 7 installation (without any anti-forensic tools installed) produced no search hits. This indicates that using the anti-forensic tool name (for the selected tools) is a unique keyword. Once an anti-forensic tool is discovered on a data set further manual investigation is required to determine the software version. Again, no widely accepted method or forensic tool exists to achieve this. However, based on the author’s knowledge there are various sources to determine tool version, including: 1) Locating the tool installer with an embedded version number in the executable; and 2) Locating a Registry value created by the tool with the version number stored in the Registry value data property. Given the variables involved, this is a manual investigation method.

6.2.5.4 Ground Truth: Determining Digital Artifacts

The previous investigation phases have determined the Windows operating system version and the anti-forensic tool version. Given these two variables it is possible to create a new application profile which will document the digital artifacts that are uniquely associated with the anti-forensic tool. This is achieved by creating an application profile with the same anti-forensic tool version on the same operating system version. Figure 6.6 displays a high-level overview of the implemented method to establish ground truth.

The known application software is used to create a new application profile for the same anti-forensic tool version and operating system version; for example, if Eraser version 4.0
was discovered on the target data set running Microsoft Windows Vista, a new application profile would be created for *Eraser* version 4.0 using Windows Vista as the host operating system. In contrast, the application profile created during this research is for *Eraser* version 6.2.0 on Windows 7.

The `fiwalk` and `CellXML-Registry` tools are used to generate DFXML and RegXML reports for each identified disk image which is known to have an anti-forensic tool present. The newly created profile is then manually compared to the metadata reports to identify the actual digital artifacts that reside in both the disk image and the newly created application profile. Manual analysis is achieved by comparing metadata properties for each entry in the application profile, primarily using file system and Registry full path values. Any entries deemed a match are then copied to a new application profile (APXML document) which retains the digital artifact metadata that is taken directly from the target data set. This document contains the actual file system and Registry entries that appear in the disk image and provide an accurate metadata representation of ground truth for the selected anti-forensic tool.

### 6.2.6 System Evaluation Metrics: Score Interpretation

Two different categories of metrics for system demonstration and evaluation were previously specified in the proposed research methodology (see Section 4.3). To reiterate, one category of metrics provides an applicable statistical measurement to determine the *effectiveness* of the implemented system to detect relevant digital artifacts from a target data set. The other metrics category was specified to provide the ability to measure the computational *efficiency* of the implemented system. Both metrics are essential to determine the capability of the implemented system and, ultimately, provide robust evidence to prove that the system is both effective and efficient and can solve the high-level research objective. The purpose of this section is to restate the evaluation metrics to be used with examples of score interpretation.
6.2.6.1 Effectiveness Metrics

A total of three effectiveness metrics were sourced from the field of Information Retrieval (IR) to aid in determining system effectiveness: 1) Precision ($P$); 2) Recall ($R$); and 3) $F_1$–measure ($F_1$). The required input to calculate each metric can be determined using a coincidence matrix to provide four levels of digital artifact classification: 1) True positive ($tp$); 2) True negative ($tn$); 3) False positive ($fp$); and 4) False negative ($fn$). Classification of detected digital artifacts is determined by comparison to the established ground truth data (see Section 6.2.5). Further information regarding each effectiveness metric is covered in Section 4.3.4, while digital artifact classifier information is covered in Section 4.3.4.2. To provide a baseline on which to evaluate system effectiveness the following score interpretations have been provided:

1) **Precision** ($P$): Measures the correctness of the system at detecting relevant digital artifacts. Low scores (e.g., 0.40) indicate that a high number of false positive (incorrect) digital artifacts were detected, while high scores (e.g., 0.95) indicate a low number of false positive digital artifacts were detected; for example, a precision score of 0.90 dictates that 10% of detected digital artifacts were false positives and incorrectly detected from the target data set.

2) **Recall** ($R$): Measures the ability of the system to detect a high number of explicitly relevant digital artifacts. Low scores (e.g., 0.40) indicate that a low number of the digital artifacts from the target data set were detected, while high scores (e.g., 0.95) indicate that a high proportion of digital artifacts were correctly detected; for example, a recall score of 0.50 dictates that only half of the relevant digital artifacts were detected, while a recall score of 1.00 dictates that all relevant digital artifacts were detected in the target data set.

3) **$F_1$–measure** ($F_1$): Provides an equally weighted average (harmonic mean) of recall and precision scores.

4) **Accuracy** ($A$): Measures ability of the system to perform digital artifact classification ($tp$, $tn$, $fp$, $fn$) during artifact matching. Low scores (e.g., 0.30) indicates that a high number of digital artifacts were incorrectly classified during processing, while a perfect score (e.g., 1.00) indicates that all digital artifacts were correctly classified during matching.

6.2.6.2 Efficiency Metrics

A total of two efficiency metrics were previously outlined (see Section 4.3.5), including: 1) absolute speed; and 2) relative speed. To provide a baseline on which to evaluate system efficiency the following score interpretations have been provided:

1) **Absolute speed**: Measures the time taken to process and analyse the target data set, represented in time (minutes). Absolute speed only provides an indication of computational efficiency when comparison to a similar solution for the same data set is available;
for example, comparing the absolute speed of another tool to the solution presented in this research using the same data set. When available, efficiency results from the solution presented in this research will be compared to other forensic analysis tools.

2) **Relative speed:** Measures the average rate that the tool can analyse the target compared to the rate at which data can be read from the source device. Relative speed provides a useful measurement of the system efficiency without any comparative results from other research. A relative speed value of 1.00 dictates that the system can perform complete processing and analysis of the target data set at the same speed at which the data can be read from the source device. A relative speed of 2.00 dictates that the system can perform complete processing at the twice the speed at which data can be read, while a relative speed of 0.50 dictates that the system can perform complete processing at half the speed at which data can be read. Overall, the higher the relative speed, the more computationally efficient.

### 6.2.7 Testing Environment Summary

This section has outlined and discussed experimental testing for system demonstration (and later evaluation) of the design with the use of VMs as a reliable and secure testing environment. Data generation and collection to create application profiles was specified including a documented procedure for recreating specific application life cycle phases. It was deemed that immutable VM disks would ensure that the same testing system would be in place for all application profile creation. The data generation and collection method for creating known data sets was then specified and points of difference from the application profile creation method were outlined. A method to establish ground truth for each data set was detailed including data requirements and a method to determine digital artifacts from application software on a target data set. Finally, an overview of effectiveness and efficiency metrics were revisited and examples of score interpretation provided to help determine system performance.

### 6.3 System Demonstration: Creating Application Profiles

The first stage of the system design is focussed on creating application profiles. System demonstration of the implementation of the LiveDiff tool and APXML data abstraction for this stage is now needed to test if the lower-level research objectives one and two can be met (see Section 4.2.2.2 and Section 4.2.2.3). To briefly reiterate, this involves: 1) Effective and efficient automated identification of relevant digital artifacts from anti-forensic tools; and 2) An effective data abstraction to store, distribute and automate processing of digital artifacts from anti-forensic tools. This section demonstrates the use of LiveDiff to reverse engineer a selection of three anti-forensic tools. The goal is automated identification of digital artifacts and subsequent population into an application profile.
6.3.1 Anti-forensic Tool Selection

Anti-forensic tools are especially relevant to this study and were therefore chosen as the input application software. Three anti-forensic tools have been previously selected for system demonstration (see Section 4.3.2), which are: CCleaner, Eraser and TrueCrypt. Testing a number of different tools is essential to ascertain whether the proposed solution is suitable for different scenarios, in this case, different application software. Table 6.5 displays an overview of the selected anti-forensic tools including the tool name, version number, tool installer file name and the Message Digest version 5 (MD5) cryptographic hash value of the installer file.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>File name</th>
<th>MD5 hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCleaner</td>
<td>5.09.5343</td>
<td>ccsetup509.exe</td>
<td>f119524883af4bac56581ed77cee828</td>
</tr>
<tr>
<td>Eraser</td>
<td>6.2.0.2970</td>
<td>Eraser 6.2.0.2970.exe</td>
<td>adac90074e56f36f8b51ea5fa5eb86</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>7.1a</td>
<td>TrueCrypt Setup 7.1a.exe</td>
<td>7a23ac83a0856c352025a6f7c9ec1526</td>
</tr>
</tbody>
</table>

6.3.2 Overview of Created Application Profiles

LiveDiff version 1.0.0 was used to create 25 application profiles for each of the anti-forensic tools. This resulted in a total of 75 application profiles. Multiple profiles were required in anticipation of this number being needed as input for later testing of the application profile intersection theory to filter irrelevant digital artifacts (see Section 3.2.3.3). The application life cycle phases (see Table 6.4 and Figure 6.4) were recreated for each anti-forensic tool and data collected using LiveDiff. Table 6.6 displays an overview of average counts for the generated application profiles including the average clock time taken to author each profile and the average APXML document size in kilobytes (KB). All counts are averaged and rounded to integer values.

Table 6.6: Average digital artifact counts for created application profiles (count values have been rounded to whole numbers)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Directories</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>Time (sec)</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCleaner</td>
<td>19</td>
<td>391</td>
<td>96</td>
<td>453</td>
<td>959</td>
<td>243</td>
<td>2,413 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>25</td>
<td>333</td>
<td>201</td>
<td>2,117</td>
<td>2,676</td>
<td>260</td>
<td>3,436 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>15</td>
<td>226</td>
<td>154</td>
<td>523</td>
<td>918</td>
<td>240</td>
<td>2,251 KB</td>
</tr>
<tr>
<td>Total</td>
<td>950</td>
<td>59</td>
<td>451</td>
<td>3,093</td>
<td>4,553</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In terms of performing data collection, LiveDiff proved to be very effective at generating application profiles. However, to learn how efficient the system design is, a comparison to a similar solution needs to be presented. The NIST Diskprint project creates reference sets for application software using a post-mortem data collection method accomplished by collecting

---

6LiveDiff version 1.0.0 was used for demonstration of application profile creation and is publicly available from: [https://github.com/thomaslaurens/LiveDiff/releases/tag/LiveDiff-1.0.0](https://github.com/thomaslaurens/LiveDiff/releases/tag/LiveDiff-1.0.0)
forensic images of VM disks (see Section 3.3.4). While there is no information regarding efficiency of the Diskprint generation method, it can be estimated based on the documented procedure. According to Microsoft (2016), the smallest hard drive size for Windows 7 32-bit is 16 GB. Using the Diskprint method a forensic disk image is collected for each application life cycle phase. This would entail creating 6 disk images (one for each application life cycle phase and a ground truth image), a total of 96 GB. Solely copying the data, without any analysis, would take 1600 seconds, or 26.67 minutes. This assumes a hard drive read/write speed of 60 MB/s. In comparison, the clock time to create an application profile using LiveDiff was 248 seconds, or 4.13 minutes (averaged from the 75 generated profiles). This includes generating and collecting data for the five different application life cycle phases (install, open, close, uninstall and reboot). It highlights the speed and efficiency at which LiveDiff can author profiles using an automated data collection procedure on a live operating system.

The total number of identified digital artifacts varied for each anti-forensic tool. The Eraser tool has approximately three times the total number of digital artifacts (2,676) when individually compared to CCleaner and TrueCrypt. As seen in Table 6.6, this was mainly due to Eraser creating over 2,100 Registry value entries. Based on these results, Eraser can be considered a more complex application that creates a higher number of digital artifacts. Even so, the application profile creation time was very similar compared to the other anti-forensic tools. CCleaner created the highest number of data files (391) and manual analysis determined this was primarily caused by the number of language packs (e.g., lang-1025.dll) that are included with the application (approximately 50). In comparison to the numbers of identified data files created by all of the tools (950), the number of identified file system directories was low (between 15 and 25). The same general trend was observed for Registry entries, where a much higher total number of Registry values were identified (3,093) when compared to Registry keys (451).

In terms of application profile size, all profiled anti-forensic tools had a very similar file size, between 2.3 and 3.4 MBs. It can be concluded that Eraser had a higher file size due to storing more than twice as many digital artifacts (2,676), compared to CCleaner and TrueCrypt which had a total of 959 and 918 respectively.

Manual review of the created application profiles discovered digital artifacts that were not unique to the profiled anti-forensic tools. This content was classified as operating system artifacts caused by system noise. In an attempt to remove unrelated digital artifacts, the proposed data reduction methods of application profile intersection and dynamic blacklisting were next subjected to experimental testing and evaluated.

### 6.4 Filtering Irrelevant Artifacts from Application Profiles

Previous research has found that system-level reverse engineering typically produces a high number of irrelevant results, usually operating system files and Registry entries, that are not uniquely associated with an application (see Section 3.2.3 and Section 4.2.1.1). These irrelevant entries are primarily due to the complexity of modern operating systems. Even
when a system appears idle there are many processes and services running in the background. This simply means that created application profiles will be littered with a high number of irrelevant operating system artifacts that are not unique to the anti-forensic tool causing false positive matches when invoked against a target data set. This is problematic when attempting to detect digital artifacts of forensic interest. However, the issue can potentially be solved by implementing data reduction techniques to filter irrelevant known operating system content.

Similar research studies have implemented set intersection to remove irrelevant operating system artifacts but such a strategy has not been exhaustively examined for effectiveness (see Section 3.2.3). Set intersection will now be referred to as application profile intersection, as each profile is treated as a unique set. The system design also proposed the use of dynamic blacklisting in an attempt to filter known operating system files and Registry entries (see Section 5.2.4.1). In order to evaluate both of these proposed solutions the created application profiles need to be demonstrated by analysing both techniques to determine overall effectiveness and efficiency.

6.4.1 Application Profile Intersection

Set theory is a fundamental mathematical concept, where a set is a collection of unique objects. In set theory, the intersection \((\cap)\) of two sets \((A \cap B)\) is a new set of objects that reside in both sets. Simply, the intersection of two sets contains the unique objects that reside in both sets. Application profile intersection uses this concept to determine the objects that reside in both profiles (APXML documents); for example, when performing intersection of two application profiles the output is a profile which only contains digital artifacts that are in both input profiles. Application profile intersection was performed in this research with the aim of removing irrelevant digital artifacts, the premise being that only digital artifacts that are present in every application profile are unique to the anti-forensic tool.

6.4.1.1 Application Profile Intersection Method

Intersection was performed by comparing application profiles from each independent anti-forensic tool. The input ranged from 2 to 25 application profiles. Intersection was accomplished by authoring a Python script, named APXMLIntersection.py, which harnesses the automated processing capabilities of the APXML API (See Section 5.3.4). The full APXMLIntersection.py script is available in Appendix C.3. The script ingests two or more APXML documents, performs intersection of all digital artifacts and produces three outputs: 1) A newly constructed APXML document containing only digital artifacts present in all application profiles; 2) A Comma Separated Values (CSV) file of all file system entries and associated metadata; and 3) A CSV file of all Registry entries and associated metadata. Both CSV files include the observed number of occurrences that each digital artifact appeared in the input profiles. The set intersection strategy implemented in the APXMLIntersection.py script follows simple logic:

1) Parse each application profile to an APXMLObject using apxml.py
2) First pass: Iterate each FileObject and CellObject from the first profile, add a count for each object

3) Subsequent passes: Iterate each subsequent profile:
   a) For each FileObject and CellObject perform a comparison of all digital artifact properties (e.g., for files this includes file name, size, hash value, allocation status and application life cycle phase) to all previously processed objects
   b) If a match is found, increase the count
   c) Or else if the object is new, add a count for the new object

4) After processing all application profiles, return entries in a newly constructed APXML document, but only when entries are present in all input profiles

6.4.1.2 Application Profile Intersection Results

Application profile intersection was performed using the 75 created application profiles (25 for each of the three anti-forensic tools). Table 6.7 displays the percentage decrease to the total digital artifact count when performing intersection of 2, 5, 10, 15, 20 and 25 profiles; for example, the five (5) column is the percentage decrease after performing intersection of five APXML documents. In this example, the percentage of digital artifacts from the CCleaner application profile was reduced by approximately 38%.

Table 6.7: Summary of the percentage decrease to total digital artifact count for the intersection of varying numbers of application profiles

<table>
<thead>
<tr>
<th>Tool</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCleaner</td>
<td>30.85%</td>
<td>38.07%</td>
<td>40.04%</td>
<td>40.59%</td>
<td>40.70%</td>
<td>42.12%</td>
</tr>
<tr>
<td>Eraser</td>
<td>11.15%</td>
<td>12.14%</td>
<td>12.14%</td>
<td>12.22%</td>
<td>12.22%</td>
<td>12.71%</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>23.39%</td>
<td>26.67%</td>
<td>27.57%</td>
<td>27.57%</td>
<td>28.93%</td>
<td>28.93%</td>
</tr>
</tbody>
</table>

Overall, the results show a decrease in the total digital artifact count when performing intersection of two application profiles for all tools (between approximately 11% and 30% total reduction in digital artifacts). Although profile intersection continues to decrease the total digital artifact count as input of profiles increases, the method yields marginal differences with each additional application profile; for example, in the TrueCrypt tool, intersection of two application profiles reduces the total digital artifact count by approximately 23%, while intersection of 25 application profiles reduces the total digital artifact count by approximately 29%. This is a further reduction of around 6%, requires authoring 23 additional application profiles for TrueCrypt as input, taking an extra 92 minutes based on presented figures. A similar trend was observed for all three tools.

Overall, performing application profile intersection was efficient due to the automated APXMLIntersection.py script which does not require user intervention, operation or manual decision making. However, authoring multiple application profiles is time consuming, as each single application profile takes approximately four minutes to create (see Table 6.6). The
following example and discussion uses the results from profiling the TrueCrypt tool. Creating two (2) application profiles takes approximately 8 minutes and performing intersection removes approximately 23% irrelevant digital artifacts. Creating five (5) application profiles takes approximately 20 minutes and performing intersection removes approximately 27% irrelevant digital artifacts. Creating twenty five (25) application profiles takes approximately 100 minutes (or 1 hour and 40 minutes) and performing intersection removes approximately 29% irrelevant digital artifacts. The reduction of irrelevant digital artifacts plateaus when additional profiles are used and minimal data reduction is achieved in comparison to the time factor involved. Figure 6.7 is a visual representation of the total digital artifact count for all three anti-forensic tools when performing application profile intersection from a single profile to all 25 profiles.

![Figure 6.7: Summary of application profile intersection results displaying the total digital artifact count for the three anti-forensic tools using from 1 to 25 application profiles](image)

Figure 6.7 illustrates the diminishing returns when more application profiles are used as input to intersection, the major trend being that the number of filtered irrelevant artifacts decreases. Nevertheless, the indication is that application profile intersection is capable of removing unrelated results. However, further analysis of the data reduction technique is required to give insight to the artifact types (directories, files, keys and values) that were removed to determine those that are predominantly associated with operating system noise; for example, do Registry value changes encompass the majority of operating system noise.

Further investigation was performed using TrueCrypt as an example. Figure 6.8 presents a summary of intersection results for TrueCrypt while also identifying the outcome for each of the four different digital artifact types.

A series of important trends are evident. Firstly, there was only one directory removed from the intersection of all TrueCrypt profiles and this occurred when using only two profiles.

---

7The purpose of Figure 6.7 is to display the overall trend resulting from application profile intersection. Full digital artifact counts are displayed in Appendix C.4 including Tables C.4, C.5 and C.6.

8Similar to Figure 6.7, the purpose of Figure 6.8 is to display the overall trend resulting from application profile intersection. Full digital artifact counts are displayed in Appendix C.4 specifically Table C.6.
as input. Therefore, it can concluded that directories are less dynamic in terms of creation due to operating system noise. Secondly, a high number of data files were removed when performing intersection. The first profile had 210 data files and intersection of 25 profiles reduced this to 167 data files (a reduction of 43 irrelevant data files). However, 18 of these were removed with the intersection of only two profiles, 24 were removed with the intersection of five profiles, and thereafter only one was removed. Thirdly, in comparison to data files, fewer Registry keys were removed, from 153 to 140. A total of 10 of these were removed from the intersection of only two profiles. Again, this result highlights the effectiveness of intersecting only two profiles. Lastly, Registry values were determined to be highly dynamic and constituted a large number of irrelevant entries. The first profile contained over 500 Registry values and intersection of two profiles reduced this number to approximately 300, a reduction of 200 irrelevant values. Most of these irrelevant results were due to modification of Registry value data and not creation of a new Registry value. Overall, the results dictate that data files and Registry values proved to account for the majority of irrelevant digital artifacts in the TrueCrypt application profiles. Further analysis of the output from application profile intersection was then deemed necessary to manually examine the profile content.

6.4.1.3 Analysis of Application Profile Intersection Output

Analysis was performed on the output from application profile intersection to investigate how the contents of the application profiles were effected by intersection. This was accomplished by reviewing the CSV output files from the APXMLIntersection.py script to determine the following: 1) The digital artifacts removed; and 2) The digital artifacts remaining after performing set intersection. Analysis was first conducted by individually investigating each digital artifact. The author’s knowledge of the Microsoft Windows operating system was
utilised to classify digital artifacts; for example, if a specific digital artifact was the result of a running Windows service. In addition, a variety of Internet sources and digital forensic resources were also used to obtain information about specific file system and Windows Registry entries. However, the primary source was the Microsoft Developer Network (MSDN) which documents information for Microsoft product developers, testers and programmers.

The first step of analysis is to determine the digital artifacts that were removed by intersection and if they were correctly or incorrectly removed. Analysis revealed that very few digital artifacts that were unique to the profiled anti-forensic tools were removed, while a high number of irrelevant operating system artifacts were discovered in all authored application profiles. This is a good result, as irrelevant operating system artifacts were removed while still retaining unique application software artifacts. However, two data file types and one Registry value type, all of forensic interest, were incorrectly removed during profile intersection.

**Directories:** Analysis revealed that no file system directories were incorrectly removed and the low number that were removed were manually classified as correct. This is explained by the fact that file system directories have no associated variable content (e.g., data file content represented by the hash value) which reduces system-level modification changes, ultimately resulting in low (in this research none) false positive results.

**Data files:** Two file types were incorrectly removed after intersection. Windows Shortcut files (e.g., TrueCrypt.lnk) were incorrectly removed because they have a variable hash value and, therefore, the correlation of all digital artifact properties cannot be achieved. Windows Shortcut files have a variable hash caused by dynamic content in the file metadata; for example, embedded timestamp information which changes on every application installation (Metz 2016b). Windows Prefetch files (e.g., TRUECRYPT.EXE-33CC2C25.pf) were also incorrectly removed due to a variable hash value, as well as a variable file name caused by a random suffix value (e.g., -33CC2C25.pf) (Metz 2016c).

**Registry keys:** Analysis revealed that no Registry keys were incorrectly removed and the low number that were removed were manually classified as correct. These results are similar to file system directory results and derive from the same behaviour, that Registry keys have no associated variable content which results in limited false positive system-level modifications.

**Registry values:** One Registry value type, UserAssist entries, were incorrectly removed after profile intersection. This was caused by similar content-based problems observed during incorrect data file removal. UserAssist Registry entries are designed to track application usage and have dynamic content which contains a last execution timestamp. Therefore, the data content varies based on when the application was executed. This results in variable data content and failure to match all properties when performing intersection.

**Remaining incorrect digital artifacts:** Although intersection did remove a high number of irrelevant digital artifacts, there were many remaining digital artifacts that were identified
during analysis as not being uniquely associated with the specific anti-forensic tool. These operating system artifacts were not removed as they were present in every authored profile. A primary example is Registry hive files. Since an anti-forensic tool made Registry changes, the actual Registry hive file (e.g., C:\Windows\System32\config\SOFTWARE) also appeared in identified file system changes. However, performing digital artifact matching using a Registry hive file is not appropriate, as every Windows operating system will have the same hive file. Another example is a high number of irrelevant operating system artifacts were found from Windows services; for example, the Windows Search service. A number of Windows Prefetch files were also not removed from intersection as they appeared in every profile; for example, the Windows Command Prefetch file (e.g., C:\Windows\Prefetch\CMD.EXE-4A81B364.pf) was found in every TrueCrypt profile authored because the tool utilises the Windows command prompt during application installation.

Profile intersection testing revealed both positive and negative effects on performing data reduction and removing irrelevant digital artifacts. Three digital artifact types were incorrectly removed due to variable file/value data content: 1) Windows Shortcut files; 2) Windows Prefetch files; and 3) UserAssist Registry values. A variety of known operating system artifacts (not unique to the profiled application) were not removed as they appeared in all application profiles, including: 1) Registry hive files; and 2) Irrelevant Windows Prefetch files. Testing and results achieved from the second data reduction technique of dynamic blacklisting follows.

6.4.2 Application Profile Blacklisting

A blacklisting technique was designed and implemented based on the specified system requirements (see Section 5.2.4.1) and experimental testing of the solution was undertaken to determine the capability of removing irrelevant operating system artifacts. Functionality was included in LiveDiff to perform a system snapshot and generate a blacklist of known operating system artifacts prior to data collection. The blacklist was implemented to include the full logical path of: 1) Data files; and 2) Registry values. Directories and Registry keys were not included as LiveDiff performs recursive scanning and as such, blacklisting directories or Registry keys would also remove their contents (nested data files or Registry values). Previous results from application profile intersection showed that both directories and Registry keys are much less dynamic and produce minimal false positive results included with the authored application profiles. This reflects the design decision of blacklisting to only include data files and Registry values. The method to determine blacklisting effectiveness follows with a summary of results from testing and analysis of the results achieved.

6.4.2.1 Application Profile Blacklisting Method

Usually LiveDiff stores the dynamically generated blacklist in volatile memory. However, a function was written and included to save a text file of all blacklist entries to enable post-mortem analysis of experimental testing results without adversely affecting run-time performance. The earlier selection of 75 application profiles used to perform application profile intersection were used (a total of 25 application profiles for each of the three anti-forensic
tools). A simple Python script, named `APXMLBlacklist.py`, was authored to perform post-mortem blacklisting which was implemented using the APXML API (`apxml.py`). The full script is available in Appendix C.5. The script ingests an APXML document and the associated text file containing blacklisted entries. The `APXMLBlacklist.py` script is similar to the set intersection script in that it produces three outputs: 

1) A newly constructed APXML document containing non-blacklisted digital artifacts; 
2) A CSV file of all file system entries, associated metadata and a blacklist status for each entry; and 
3) A CSV file of all Registry entries, associated metadata and a blacklist status for each entry. The blacklist strategy implemented in the script follows simple logic:

1) Parse the application profile to an `APXMLObject` using `apxml.py` 
2) Parse the blacklist into a Python dictionary mapping full path to the actual object 
3) Iterate each `FileObject` and `CellObject` in the profile 
4) Perform a dictionary lookup using the object’s full path. If the path is found the object is deemed blacklisted and irrelevant

### 6.4.2.2 Application Profile Blacklisting Results

The `APXMLBlacklist.py` script implements the same logic as dynamic blacklisting implemented in LiveDiff, the only difference being it is not conducted during run-time. Blacklist testing was performed using all 75 generated application profiles. However, each profile was individually used as input to the `APXMLBlacklist.py` script and the results averaged for each anti-forensic tool. Table 6.8 displays an overview of the data files and Registry value counts before and after performing blacklisting as well as the percentage decrease resulting from blacklisting. As stated, all counts are averaged for each tested tool.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data files</th>
<th></th>
<th></th>
<th></th>
<th>Registry values</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Decrease</td>
<td>Before</td>
<td>After</td>
<td>Decrease</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>391</td>
<td>234</td>
<td>40%</td>
<td>453</td>
<td>240</td>
<td>47%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eraser</td>
<td>333</td>
<td>213</td>
<td>36%</td>
<td>2117</td>
<td>1008</td>
<td>52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>225</td>
<td>172</td>
<td>24%</td>
<td>523</td>
<td>227</td>
<td>57%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results from blacklisting showed a consistent reduction in the total number of digital artifacts for each anti-forensic tool profile. For all three tools the number of data files reduced after blacklisting was between 24% and 40%, while the number of Windows Registry values also reduced between 47% and 57%. Therefore, it can be concluded that blacklisting filtered a large proportion of irrelevant application profile entries. From the 75 created application profiles the total number of data files was 25,183; of these, 15,460 were blacklisted and a total of 9,723 data files remained, an overall reduction of a little over 61%. In terms of Registry value entries, there were a total of 62,986 Registry values from all 75 application profiles; of these, 25,212 were blacklisted and a total of 37,774 Registry values remained, an overall
reduction of approximately 40%. This largely demonstrates that blacklisting reduced the number of irrelevant data files and Registry values by classifying many as known operating system artifacts.

Figure 6.9 displays a visual representation of the results from implementing dynamic blacklisting. The average for each anti-forensic tool profile before performing blacklisting is displayed (blue) and the average after performing blacklisting is shown (orange). Error bars are included which specify the standard deviation based on the difference observed from the averaged profile contents for each anti-forensic tool.

![Graph showing digital artifact counts before and after blacklisting](image)

Figure 6.9: Averaged application profile counts before and after dynamic blacklisting

It is visually evident that the digital artifact count is reduced when performing blacklisting. The number of digital artifacts for each anti-forensic tool was reduced by over approximately 50%. However, analysis of the blacklisted digital artifacts is now required to determine if any entries were incorrectly removed and the type of digital artifacts that were filtered.

### 6.4.2.3 Analysis of Application Profile Blacklisting

Although the presented data reduction statistics illustrate effective removal of digital artifacts, analysis of the blacklisted entries is essential to determine which application profile entries were removed and if entries were removed that are actually unique to the application. This section analyses and discusses correctly and incorrectly removed digital artifacts observed from testing the dynamic blacklisting design. Detailed tables are included which outline blacklisted
digital artifact types and classification based on the operating system functionality that each provides.

Blacklisted data files: Analysis of blacklisted data files revealed that no digital artifacts that were unique to an application were removed. The results indicate that dynamic blacklisting removed a large number of irrelevant data files from a variety of Windows services and processes. Prominent examples of blacklisted data files include: 1) Windows Updates that had been previously downloaded, installed and were removed during application profile creation time; 2) Normal Windows event log activity; 3) Modified Registry hive files; 4) Data files created by the Windows Search service; 5) The Windows CryptoAPI certificate cache (CryptnetURLCache); 6) Windows Prefetch files (e.g., TASKHOST.EXE-7238F31D.pf) and Prefetch databases (e.g., AgGlFgAppHistory.db); and 7) Windows log files. Each of these classifications (except log files) had in excess of 1,000 data files blacklisted from all of the 75 created application profiles. A variety of other operating system files were also discovered but most had fewer entries. Table 6.9 presents an overview of the most prominent blacklisted data files including classification, a total count of occurrence and an example of the file system location.

Table 6.9: Blacklisted data files classified on logical file system location

<table>
<thead>
<tr>
<th>Classification</th>
<th>Count</th>
<th>Example location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows Updates</td>
<td>1,965</td>
<td>C:\Windows\SoftwareDistribution\DataStore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C:\Windows\SoftwareDistribution\Download</td>
</tr>
<tr>
<td>Event Logs</td>
<td>1,902</td>
<td>C:\Windows\System32\winevt</td>
</tr>
<tr>
<td>Registry hives</td>
<td>1,809</td>
<td>C:\Windows\System32\config\SOFTWARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C:\Users%USERNAME%\NTUSER.DAT</td>
</tr>
<tr>
<td>Search service</td>
<td>1,446</td>
<td>C:\ProgramData\Microsoft\Search\Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C:\Users\All Users\Microsoft\Search\Data</td>
</tr>
<tr>
<td>CryptnetURLCache</td>
<td>1,307</td>
<td>AppData\LocalLow\Microsoft\CryptnetUrlCache</td>
</tr>
<tr>
<td>Prefetch files</td>
<td>1,288</td>
<td>C:\Windows\Prefetch</td>
</tr>
<tr>
<td>Log files</td>
<td>887</td>
<td>C:\Windows\System32\LogFiles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C:\Windows\Logs</td>
</tr>
<tr>
<td>RACAgent</td>
<td>386</td>
<td>C:\ProgramData\Microsoft\RAC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C:\Users\All Users\Microsoft\RAC</td>
</tr>
<tr>
<td>Windows Defender</td>
<td>99</td>
<td>C:\Users\All Users\Microsoft\Windows Defender</td>
</tr>
<tr>
<td>Pagefile</td>
<td>75</td>
<td>C:\pagefile.sys</td>
</tr>
</tbody>
</table>

Blacklisted Registry values: Analysis of blacklisted Registry entries revealed no entries were removed that were unique to the anti-forensic tools. Similar to the results from blacklisting data files, the results illustrated that the blacklisted Registry values were known Windows

10 The displayed CryptnetUrlCache path has been shortened. The full CryptnetURLCache path is: C:\Users\%USERNAME%\AppData\LocalLow\Microsoft\CryptnetUrlCache

180
operating system processes and services. Table 6.10 presents an overview of blacklisted Registry value entries including classification, a count of occurrence and an example of the full Registry entry path.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Count</th>
<th>Example path</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUI Cache</td>
<td>3,775</td>
<td>HKU.DEFAULT\Software\Classes\Local Settings\MuiCache</td>
</tr>
<tr>
<td></td>
<td>3,700</td>
<td>HKU\S-1-5-18\Software\Classes\Local Settings\MuiCache</td>
</tr>
<tr>
<td></td>
<td>2,984</td>
<td>HKU%SID%\Classes\Local Settings\MuiCache</td>
</tr>
<tr>
<td></td>
<td>2,750</td>
<td>HKU%SID%\Software\Classes\Local Settings\MuiCache</td>
</tr>
<tr>
<td>Control Set</td>
<td>1,968</td>
<td>SYSTEM%CONTROLSET\Control</td>
</tr>
<tr>
<td></td>
<td>1,691</td>
<td>SYSTEM%CONTROLSET\Enum</td>
</tr>
<tr>
<td></td>
<td>1,834</td>
<td>SYSTEM%CONTROLSET\services</td>
</tr>
<tr>
<td>Current Version</td>
<td>1,834</td>
<td>HKU%SID%\Software\Microsoft\Windows\CurrentVersion</td>
</tr>
<tr>
<td></td>
<td>979</td>
<td>SOFTWARE\Microsoft\Windows\CurrentVersion</td>
</tr>
<tr>
<td></td>
<td>490</td>
<td>SOFTWARE\Microsoft\Windows NT\CurrentVersion</td>
</tr>
<tr>
<td>WMI</td>
<td>1,372</td>
<td>SOFTWARE\Microsoft\WBEM</td>
</tr>
<tr>
<td>Windows Search</td>
<td>1,114</td>
<td>SOFTWARE\Microsoft\Windows Search</td>
</tr>
</tbody>
</table>

**Potential blacklist failure:** After designing, implementing and testing the dynamic blacklisting technique various lessons were learned regarding operation and functionality. Although not witnessed during testing, it is hypothesised that blacklisting may fail in the following scenario; for example, where a data file or Registry value exists, is included in the blacklist, but later modified by the application. Since the entry is already blacklisted it will not appear in differential analysis results. A good example is Windows Registry hive files. Each hive file (e.g., C:\Windows\System32\config\SOFTWARE) exists on all Windows system and will be present when initially running LiveDiff, therefore, will be populated into the dynamic blacklist. Most applications will modify the SOFTWARE hive and write Registry entries to provide application configuration. However, since the SOFTWARE hive file was blacklisted, it will not appear in differential analysis results. In this example, the fact that the hive file itself is blacklisted in of no consequence because the Registry contents are processed during differential analysis and the associated data file is not of interest. However, take the example of a log file. If a log file exists during LiveDiff blacklist generation, it is added to the blacklist and then the application modifies the log file and will not be included in differential analysis results. This is because no lower-level processing is conducted by LiveDiff (like Registry differencing), and thus, potential system-level differences are not included. Another data file example is the Windows pagefile.sys. To include pagefile information in the application profile, the pagefile would need to be independently processed at a high-level of abstraction, similar to Windows Registry processing.

The implementation and testing of dynamic blacklisting revealed a number findings in terms of data reduction capabilities. Although blacklisting removed no unique digital artifacts, it can be concluded that the technique may fail in some scenarios. However, blacklisting
successfully removed a high proportion of irrelevant digital artifacts that were manually classified as irrelevant operating system noise caused by known operating system services and processes.

6.4.3 Discussion of Data Reduction Findings

Two data reduction techniques to remove irrelevant digital artifacts were tested, namely application profile intersection and dynamic blacklisting. Both have the same goal, to remove digital artifacts not unique to the anti-forensic tool being reverse engineered; for example, known operating system files created by constant system noise. Two different techniques were implemented and tested to determine the effectiveness of each strategy. Each technique can be used individually or in conjunction with each other.

The findings from application profile intersection and dynamic blacklisting revealed that multiple digital artifact types were present in the application profiles that were not unique to the anti-forensic tools. It is important to note that many of these digital artifacts were actually created by the anti-forensic tool itself, but are not unique to the tool so are not useful when creating a reference set to compare against a target; for example, every anti-forensic tool made changes to the pagefile.sys. However, if it was included in the application profile it would be found on every target running Microsoft Windows as the actual pagefile.sys file is present on every Windows NT system.

Both techniques used to filter irrelevant operating system artifacts proved effective at performing data reduction of the created application profiles. However, both had some disadvantages especially the removal of digital artifacts that were manually classified as being unique to the profiled application. The major drawback from performing application profile intersection is that the method requires multiple profiles and thus extra time. However, the testing results show that LiveDiff is efficient at creating application profiles. The average real elapsed time to author an application profile was approximately 4 minutes, compared to an estimated minimum 26 minutes for similar application reference set solutions. This speed is primarily due to data collection being performed on a live system using an automated procedure with minimal user intervention. Therefore, the efficiency of LiveDiff lends itself to creating multiple profiles for comparison.

Overall, blacklisting known operating system artifacts (known data files and Registry values) proved more effective than profile intersection at removing irrelevant results. However, there is one possible scenario (not witnessed during testing) that may irreversibly affect results when an existing file or Registry value is blacklisted and then modified by an application.

After manual review of the results from both techniques it can be concluded that to achieve the best effectiveness at removing irrelevant results, both application profile intersection and dynamic blacklisting should be used in conjunction when creating application profiles. This is because each technique was capable of removing different types of irrelevant digital artifacts; for example, while intersection failed to remove Registry hive files, blacklisting correctly removed these entries.

This section has demonstrated, tested and discussed data reduction techniques using the 75
created application profiles. Irrelevant digital artifacts were found when using intersection and
dynamic blacklisting which removed a proportion of these operating system artifacts. These
techniques aided in automating the creation of application profiles with unique application
software artifacts. A revised method now needs to be developed and tested for creating
application profiles based on the findings achieved.

6.5 System Demonstration: Recreated Application Profiles

The investigation and outcome of filtering irrelevant artifacts from application profiles has
now initiated a new method of profile creation. This method is now tested to determine
effectiveness by creating and analysing the contents of three recreated application profiles. A
final iteration is then performed and a new technique called static blacklisting is added to LiveDiff so that the small number of irrelevant operating system artifacts still remaining
may be removed. The final result is three application profiles, one for each anti-forensic tool,
ready for demonstration of the digital artifact matching stage of the system design against
the known data set.

6.5.1 Revised Application Profile Generation Method

The demonstration of application profile creation provided findings on which to reliably further
design an improved method. Specifically, results were obtained that highlight the applicability
of the data generation method used and the effectiveness of various data reduction techniques.
These findings led to a revised application profile creation method.

**Application profile intersection** proved to be effective at removing irrelevant digital
artifacts and experimental results determining the most effective number of profiles to achieve
the best outcome was tested. However, a balance between effectiveness and time efficiency
is needed. The findings aided in specifying that five (5) application profiles is an optimum
number of profiles needed to reduce irrelevant profile entries while still maintaining efficiency.
Results from implementing and testing **dynamic blacklisting** also proved an effective data
reduction technique and did not remove any relevant application profile entries. Additionally,
the results from dynamic blacklisting also removed all digital artifacts from the reboot appli-
cation life cycle, meaning none were unique to the anti-forensic tool. Therefore, the reboot
life cycle phase has been removed from any further experimental testing. These results have
led to the following revised method for application profile generation:

1) Create five application profiles using LiveDiff in application profile mode with dy-
namic blacklisting and MD5/SHA1 file hashing enabled
2) Perform data collection for the following application life cycle phases: install, open,
close, uninstall
3) Perform intersection of the five application profiles using APXMLIntersection.py
4) Perform a manual analysis of the final application profile to verify document contents
   and potentially identify irrelevant operating system artifacts
The revised application profile creation method needs to be demonstrated to reassess the modifications. Furthermore, the effectiveness of combining dynamic blacklisting with application profile intersection also needs to be investigated.

### 6.5.2 Overview of Recreated Application Profiles

The findings presented thus far resulted in modifications to the system design. These modifications required that application profiles needed to be recreated using the revised method (see Section 6.5.1). Therefore, three new application profiles were authored for the three selected anti-forensic tools (CCleaner, Eraser, and TrueCrypt). Table 6.11 displays an overview comparing the originally created and revised application profiles. The original application profiles (top) are the profiles created without intersection or dynamic blacklisting (taken from Table 6.6), while the revised application profiles (bottom) were created using the revised method. The table includes counts for each digital artifact type (directory, files, keys and values) as well as the total digital artifact count. The file size column provides the file size in kilobytes (KB) for each application profile.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>19</td>
<td>391</td>
<td>96</td>
<td>453</td>
<td>959</td>
<td>2,413 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>25</td>
<td>333</td>
<td>201</td>
<td>2,117</td>
<td>2,676</td>
<td>3,436 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>15</td>
<td>226</td>
<td>154</td>
<td>523</td>
<td>918</td>
<td>2,251 KB</td>
</tr>
<tr>
<td><strong>Revised</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>7</td>
<td>130</td>
<td>62</td>
<td>113</td>
<td>312</td>
<td>353 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>15</td>
<td>72</td>
<td>88</td>
<td>189</td>
<td>364</td>
<td>431 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>6</td>
<td>31</td>
<td>117</td>
<td>169</td>
<td>323</td>
<td>367 KB</td>
</tr>
</tbody>
</table>

When compared to the original unfiltered application profiles, the revised method using dynamic blacklisting coupled with intersection resulted in profiles with a much lower digital artifact count. The resultant file size of all profiles was also much smaller than that from initial creation, due to the removal of irrelevant digital artifacts.

As per the specified revised application profile creation method, the final step entailed that the resultant contents of application profiles be manually reviewed for potential erroneous digital artifacts. Even after application profile intersection and dynamic blacklisting, a number of known operating system artifacts were still present. Out of the total 999 digital artifacts from all three revised application profiles, approximately 60 operating system artifacts were located. All of these entries were previously observed in data reduction results (see Table 6.9 and Table 6.10). However, these entries were not removed due to the following problems: 1) The entries were present in every profile, resulting in profile intersection not removing the entry; and 2) The irrelevant entries were created after dynamic blacklist generation, resulting in the entry not being blacklisted. To overcome the problem of continual irrelevant operating
system artifacts not being filtered, new functionality was added to LiveDiff in the form of a static blacklist populated with known irrelevant file system and Registry locations discovered during data reduction testing.

6.5.3 Further Data Reduction: Static Blacklist Implementation

In order to combat the remaining irrelevant operating system artifacts the dynamic blacklisting technique was extended to provide the functionality to populate a selection of static entries defined by the user. Therefore, a static blacklist was authored and support added to LiveDiff to include these entries in the blacklisting procedure. This was a relatively straightforward task as blacklisting was already a component of the LiveDiff tool. The static blacklist is a text file that contains a list of known irrelevant file system and Registry locations. The static blacklist was manually populated by entering the irrelevant file system and Registry paths discovered from the revised application profiles and results obtained from experimental testing. Listing 6.8 displays the format of the static blacklist with a collection of example entries. The full static blacklist is provided in Appendix C.6.

<table>
<thead>
<tr>
<th>LIVEDIFF STATIC BLACKLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td># Any line starting with a &quot;#&quot; is a comment</td>
</tr>
<tr>
<td># The following prefixes are defined:</td>
</tr>
<tr>
<td># DIR= Specify a directory and contents to be excluded</td>
</tr>
<tr>
<td># FILE= Specify a file to be excluded</td>
</tr>
<tr>
<td># KEY= Specify a Registry key, sub-key and values to be excluded.</td>
</tr>
<tr>
<td># VALUE= Specify a Registry value to be excluded.</td>
</tr>
</tbody>
</table>

########## FILE SYSTEM ENTRIES
# Windows Defender program data
DIR=C:\ProgramData\Microsoft\Windows Defender
DIR=C:\Users\All Users\Microsoft\Windows Defender
# Entries related to the Windows Search service
DIR=C:\ProgramData\Microsoft\Search
DIR=C:\Users\All Users\Microsoft\Search

########## WINDOWS REGISTRY ENTRIES
# MUICache entries
KEY=HKU\.DEFAULT\Software\Classes\Local Settings\MuiCache
KEY=HKU\S-1-5-18\Software\Classes\Local Settings\MuiCache
# CurrentControlSet duplicates
KEY=HKLM\SYSTEM\CurrentControlSet\001
KEY=HKLM\SYSTEM\CurrentControlSet\002

Listing 6.8: Example of LiveDiff static blacklist structure and entries

The static blacklist uses a prefix for any specified paths to provide support to blacklist directories (DIR=), files (FILE=), Registry keys (KEY=) and Registry values (VALUE=). Each prefix is followed by the absolute path of each file system or Registry entry; for example, DIR=C:\ProgramData\Microsoft\Windows Defender specifies the Windows
Defender application data directory and all child contents to be excluded during the snapshot process. A command line argument was added to LiveDiff to provide support to include a static blacklist. The -f argument can be specified, followed by the static blacklist name. If specified, the static blacklist is parsed and populated into the appropriate blacklist already implemented in LiveDiff (see Section 5.2.4.1).

Given the changes to the system implementation, application profiles were created again to test the final method. To briefly reiterate, the finalised method involves creating five application profiles for each tool with dynamic blacklisting enabled, inclusion of static blacklisting and then performing intersection using all five profiles as input. Table 6.12 displays a further review of profile contents comparing the original, revised and final application profile creation methods. Included are the counts for each of the digital artifact types, a total count of all digital artifacts and the final profile size (in KB).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>19</td>
<td>391</td>
<td>96</td>
<td>453</td>
<td>960</td>
<td>2,413 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>25</td>
<td>333</td>
<td>201</td>
<td>2,117</td>
<td>2,676</td>
<td>3,436 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>15</td>
<td>226</td>
<td>154</td>
<td>523</td>
<td>918</td>
<td>2,251 KB</td>
</tr>
<tr>
<td>Revised</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>7</td>
<td>130</td>
<td>62</td>
<td>113</td>
<td>312</td>
<td>353 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>15</td>
<td>72</td>
<td>88</td>
<td>189</td>
<td>364</td>
<td>431 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>6</td>
<td>31</td>
<td>117</td>
<td>169</td>
<td>323</td>
<td>367 KB</td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCleaner</td>
<td>6</td>
<td>123</td>
<td>36</td>
<td>92</td>
<td>257</td>
<td>310 KB</td>
</tr>
<tr>
<td>Eraser</td>
<td>15</td>
<td>72</td>
<td>76</td>
<td>187</td>
<td>350</td>
<td>419 KB</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>6</td>
<td>30</td>
<td>116</td>
<td>169</td>
<td>321</td>
<td>364 KB</td>
</tr>
</tbody>
</table>

A manual analysis of application profile contents was, again, performed. The inclusion of the static blacklist was deemed successful at removing the remaining irrelevant operating system artifacts as there were no operating system-specific entries discovered. The successful finalised application profile method has resulted in three application profiles that are ready for testing against known data sets.

6.5.4 Application Profile Creation Summary

The system demonstration phase of the DSRM process model has worked towards a thoroughly tested first stage of the system design, that is, creation of application profiles. Firstly, the original method for the creation of application profiles was performed (see Section 6.3), investigation regarding two filtering techniques to remove irrelevant operating systems entries from application profiles was then conducted (see Section 6.4) and, finally, recreation of application profiles was achieved based on the findings, lessons learned and associated system.

\footnote{Full application profile metadata contents are provided in Appendix C.7}
A final selection of profiles were thus recreated using all of the proposed filtering techniques, namely: application profile intersection, dynamic blacklisting and static blacklisting. This has resulted in three application profiles that are completely free of irrelevant operating system artifacts and ready for the next stage of system testing. Demonstration of the second main stage of the system design; that is, digital artifact matching using the created application profiles, is now carried out.

6.6 System Demonstration: Digital Artifact Matching

Digital artifact matching is the process of correlating digital artifacts between the application profile(s) and the target data set. The purpose is to automate detection of known digital artifacts and determine the presence, or absence, of anti-forensic tools. The aim of this section is to outline the data set used for the experimental testing method to demonstrate digital artifact matching. The section begins with an overview of a specifically authored data set with known content created in a laboratory controlled environment. A brief summary of the method used to perform experimental testing is followed by the establishment of the ground truth contents of the known data set to enable calculation of the prescribed effectiveness metrics and determine system performance at detecting digital artifacts of forensic interest.

6.6.1 Overview of Known Data Set

A method to author known data sets was previously specified (see Section 6.2.4). The only change to the prescribed data generation and collection method is that the reboot application life cycle phase was removed because no unique application software artifacts were discovered in this life cycle phase during demonstration of application profile creation. Therefore, each anti-forensic tool has a total of four application life cycle phases that are to be tested: 1) Install; 2) Open; 3) Close; and 4) Uninstall. Additionally, one ground truth system was also included in the known data set corpus. This target data set was installed with Microsoft Windows 7 and used to create the additional VMs as needed and will be used as an experimental testing baseline. The result is a total of 13 target data sets; that is, four forensic disk images for each anti-forensic tool and one ground truth disk image. Table 6.13 displays an overview of the forensic disk images that comprise the known data set corpus including the scenario (anti-forensic tool), disk naming convention and application life cycle phase.

Prior to performing experimental testing, the known data set was analysed to determine the size and complexity of the content in terms of file system and Registry entries. Each of the thirteen scenarios that comprise the known content data set are a forensic disk image (bit-by-bit copy) of a 10 GB virtual disk. This equates to a total data set size of 130 GB of uncompressed data stored in the raw disk format.

Additional analysis of the file system and Registry entry content was performed to determine the number and type of digital artifacts in the known data set. This was accomplished using two Python scripts authored during the course of this research. FileSystemStats.py parses a forensic disk image and counts all file system entries (and also extracts Registry hive
### Table 6.13: Overview of the created forensic disk images from the known data set

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Name</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground Truth</td>
<td>GT-01</td>
<td>Fresh operating system install</td>
</tr>
<tr>
<td>CCleaner</td>
<td>CC-01</td>
<td>Installed</td>
</tr>
<tr>
<td></td>
<td>CC-02</td>
<td>Opened</td>
</tr>
<tr>
<td></td>
<td>CC-03</td>
<td>Closed</td>
</tr>
<tr>
<td></td>
<td>CC-04</td>
<td>Uninstalled</td>
</tr>
<tr>
<td>Eraser</td>
<td>ER-01</td>
<td>Installed</td>
</tr>
<tr>
<td></td>
<td>ER-02</td>
<td>Opened</td>
</tr>
<tr>
<td></td>
<td>ER-03</td>
<td>Closed</td>
</tr>
<tr>
<td></td>
<td>ER-04</td>
<td>Uninstalled</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>TC-01</td>
<td>Installed</td>
</tr>
<tr>
<td></td>
<td>TC-02</td>
<td>Opened</td>
</tr>
<tr>
<td></td>
<td>TC-03</td>
<td>Closed</td>
</tr>
<tr>
<td></td>
<td>TC-04</td>
<td>Uninstalled</td>
</tr>
</tbody>
</table>

files). RegistryStats.py parses the extracted Registry hive files and counts all Registry entries. Both scripts are available in Appendix C.8 and C.9. Table 6.14 displays an overview of the contents of the disk images from the known data set. The total count and average count per disk image is displayed for both file system and Registry hive artifacts, as well as information regarding the data set size. Full statistics for each disk image from the known data set is available in Appendix C.10.

### Table 6.14: Overview of the M57-Patents scenario content

<table>
<thead>
<tr>
<th>Property</th>
<th>Total</th>
<th>Average per disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set size</td>
<td>130 GB</td>
<td>10 GB</td>
</tr>
<tr>
<td><strong>File System</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Directories</td>
<td>382,760</td>
<td>29,443</td>
</tr>
<tr>
<td>Data files</td>
<td>641,082</td>
<td>49,314</td>
</tr>
<tr>
<td>Other</td>
<td>38,128</td>
<td>2,933</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>1,021,679</td>
<td>78,591</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>40,291</td>
<td>3,099</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,061,970</strong></td>
<td><strong>81,690</strong></td>
</tr>
<tr>
<td>Hive files</td>
<td>130</td>
<td>10</td>
</tr>
<tr>
<td><strong>Registry</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registry keys</td>
<td>1,545,232</td>
<td>118,864</td>
</tr>
<tr>
<td>Registry values</td>
<td>2,738,568</td>
<td>210,659</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>4,178,359</td>
<td>321,412</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>105,441</td>
<td>8,111</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4,283,800</strong></td>
<td><strong>329,523</strong></td>
</tr>
</tbody>
</table>
6.6.2 Experimental Testing Method

Each of the 13 known data set scenarios were used for experimental testing. To accomplish this, each forensic disk image was used as input to the Vestigium tool. Listing 6.9 displays an example of how Vestigium was invoked to perform automated digital artifact matching. The listing displays the command invoked to perform testing against the CCleaner disk image where the tool had been installed; that is, the known data set scenario named CC-01. Tool output is saved to the user-specified output directory (TC-01-output). Testing was performed using all three application profiles for the selected anti-forensic tools. Again, the listing specifies new line characters (\^) and comments (REM) using Windows command conventions.

```
$ C:\Python34\python.exe Vestigium.py ^
D:\KDS\CC-01.RAW ^ REM Forensic disk image
D:\KDS\CC-01-output ^ REM Output folder
CCleaner-5.09-6.1.7601.apxml ^ REM CCleaner profile
Eraser-6.2.0.2970-6.1.7601.apxml ^ REM Eraser profile
TrueCrypt-7.1a-6.1.7601.apxml ^ REM TrueCrypt profile
```

Listing 6.9: Command line example to invoke Vestigium against the known data set

The code snippet above invokes Vestigium to process the CCleaner forensic disk image (CC-01.RAW) using all three (3) application profiles as input. Testing each scenario using all three application profiles was undertaken to establish if any digital artifacts were incorrectly detected from a different anti-forensic tool than that present on the specified known data set scenario; for example, if an entry from the Eraser profile was detected on the CCleaner installation scenario it can be concluded that the detected Eraser artifact is a false positive. Additionally, all three application profiles were used as input since, in most digital investigations, analysts will want to search for the existence of any anti-forensic tool wherever present.

6.6.3 Establishing Known Data Set Ground Truth

The experimental testing method specified that the ground truth needs to be established so that the digital artifacts of forensic interest on the data sets used for testing can be determined (see Section 6.2.5). This information provides a baseline on which to evaluate the effectiveness of the digital artifact matching stage of the system design. Simply put, the ground truth specifies what digital artifacts are present, and should be detected, on each data set. A method was specified for determining ground truth (see Section 6.2.5.3), but is not required for known data set testing as the content of the data sets are already known. Ground truth can be established based on the application profile contents because the same operating system (Microsoft Windows 7), the same anti-forensic tool, and the same tool version has been used for both application profile creation and known data set generation. This culminates in the fact that the digital artifacts present in the application profile and the associated known data
set should be the same; for example, the Eraser profile should have the same installation footprint (and associated digital artifacts) as the ER-01 scenario. However, before testing was undertaken an informal check was performed to confirm the data set contents and ensure that the data was generated correctly.

A manual analysis was conducted on each forensic disk image to determine the digital artifacts present on the system. Various de facto computer forensic tools were used to analyse the forensic images and browse the file system and Windows Registry for known application software artifacts. Manual analysis was performed using two widely accepted computer forensic tools from the well known Access Data\textsuperscript{12} company: 1) FTK Imager version 3.4.2 to browse and search file system entries; and 2) Registry Viewer version 1.8.1.3 to browse and search the extracted Windows Registry hive files. The manual analysis procedure involved reviewing the contents of the application profile for each associated known data set scenario and manually searching for digital artifacts. The known data set scenario was confirmed correct if the entries in the application profile were locatable. Using this procedure there were no errors found in the generated known data set and, thus, were deemed ready for experimental testing. Table 6.15 displays an overview of digital artifact counts for the ground truth information on the created known data set. For each testing scenario (forensic disk image, anti-forensic tool and application life cycle phase) a count is provided for each digital artifact type, including a total count of all digital artifacts.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tool</th>
<th>Phase</th>
<th>File System</th>
<th>Registry</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT-01</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CC-01</td>
<td>CCleaner</td>
<td>install</td>
<td>65</td>
<td>50</td>
<td>115</td>
</tr>
<tr>
<td>CC-02</td>
<td>CCleaner</td>
<td>open</td>
<td>66</td>
<td>73</td>
<td>139</td>
</tr>
<tr>
<td>CC-03</td>
<td>CCleaner</td>
<td>close</td>
<td>66</td>
<td>79</td>
<td>145</td>
</tr>
<tr>
<td>CC-04</td>
<td>CCleaner</td>
<td>uninstall</td>
<td>66</td>
<td>79</td>
<td>145</td>
</tr>
<tr>
<td>ER-01</td>
<td>Eraser</td>
<td>install</td>
<td>41</td>
<td>127</td>
<td>168</td>
</tr>
<tr>
<td>ER-02</td>
<td>Eraser</td>
<td>open</td>
<td>42</td>
<td>136</td>
<td>178</td>
</tr>
<tr>
<td>ER-03</td>
<td>Eraser</td>
<td>close</td>
<td>45</td>
<td>136</td>
<td>181</td>
</tr>
<tr>
<td>ER-04</td>
<td>Eraser</td>
<td>uninstall</td>
<td>48</td>
<td>139</td>
<td>187</td>
</tr>
<tr>
<td>TC-01</td>
<td>TrueCrypt</td>
<td>install</td>
<td>16</td>
<td>142</td>
<td>158</td>
</tr>
<tr>
<td>TC-02</td>
<td>TrueCrypt</td>
<td>open</td>
<td>18</td>
<td>144</td>
<td>162</td>
</tr>
<tr>
<td>TC-03</td>
<td>TrueCrypt</td>
<td>close</td>
<td>19</td>
<td>145</td>
<td>164</td>
</tr>
<tr>
<td>TC-04</td>
<td>TrueCrypt</td>
<td>uninstall</td>
<td>22</td>
<td>146</td>
<td>168</td>
</tr>
</tbody>
</table>

| Total | 514 | 1,396 | 1,910 |

The pertinent experimental testing environment and the method to be performed for known data set testing has now been set out. An overview of the created known data set was provided, followed by the method used to invoke the Vestigium tool against the forensic...
disk images from the known data set. Finally, the ground truth information was established and digital artifact counts provided to aid in calculating the prescribed effectiveness metrics. Coupled with the finalised application profiles, this information puts forward all the necessary variables to execute experimental testing and demonstrate the effectiveness of the digital artifact matching stage of the system design.

6.6.4 Known Data Set Testing

Experimental testing on the known data set was performed by invoking Vestigium against each of the thirteen disk image scenarios. For each disk image, all three application profiles were used as input. Digital artifact matching was accomplished by matching the file system entries and Registry entries from the input application profiles against all the corresponding entries from each disk image.

This section proceeds to outline the effectiveness results from known data set testing with reference to the calculated metrics. Efficiency results are not included in known data set testing as there are no comparative results available. Instead, efficiency results are presented for two data sets in the evaluation chapter (see Chapter 7). A discussion of the results outlines the successes, failures and lessons learned from experimental testing in the laboratory controlled environment. This leads to system design and implementation refinement before proceeding to system evaluation as per the cyclical nature of the design science research methodology.

6.6.4.1 Matching Effectiveness: File System Entries

Matching file system entries incorporates the correlation of file system directories and data files between the application profile(s) and the target data set. Table 6.16 displays an overview of the results from file system matching. The table conveys all file system entries from the target data set that have been categorised as one of the four digital artifacts classifiers: tp, tn, fp, fn. This was achieved by comparison to ground truth information. The binary matrix classification system provides the capability to calculate each of the four prescribed effectiveness metrics ($P$, $R$, $F_1$, $A$) which are outlined in Section 4.3.4. Further information regarding the digital artifact classification scheme is available in Section 4.3.4.2 and score interpretation examples are available in Section 6.2.6.

The Vestigium tool with input from the finalised application profiles proved highly effective when executed against the selection of disk images from the known data set. Digital artifact matching was effective in terms of detection of relevant digital artifacts from the application profiles, while also reporting no false positive matches. There were no digital artifacts detected in the ground truth (GT-01) disk image, a good result meaning that no anti-forensic tool artifacts were present on the disk image, a default installation of Microsoft Windows 7 without added software. Furthermore, no digital artifacts from an anti-forensic tool were detected on a data set where the tool was not introduced; for example, no CCleaner entries were detected on a TrueCrypt disk image.
Table 6.16: Effectiveness results for file system matching for the known data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Phase</th>
<th>( \text{tp} )</th>
<th>( \text{tn} )</th>
<th>( \text{fp} )</th>
<th>( \text{fn} )</th>
<th>( P )</th>
<th>( R )</th>
<th>( F_1 )</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT-01</td>
<td>N/A</td>
<td>0</td>
<td>80,551</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CC-01</td>
<td>install</td>
<td>65</td>
<td>80,629</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-02</td>
<td>open</td>
<td>66</td>
<td>80,629</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-03</td>
<td>close</td>
<td>66</td>
<td>80,636</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-04</td>
<td>uninstall</td>
<td>59</td>
<td>80,602</td>
<td>0</td>
<td>7</td>
<td>1.00</td>
<td>0.89</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-01</td>
<td>install</td>
<td>41</td>
<td>83,494</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-02</td>
<td>open</td>
<td>42</td>
<td>83,489</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-03</td>
<td>close</td>
<td>45</td>
<td>83,523</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-04</td>
<td>uninstall</td>
<td>35</td>
<td>83,524</td>
<td>0</td>
<td>13</td>
<td>1.00</td>
<td>0.73</td>
<td>0.84</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-01</td>
<td>install</td>
<td>16</td>
<td>81,092</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-02</td>
<td>open</td>
<td>18</td>
<td>81,094</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-03</td>
<td>close</td>
<td>19</td>
<td>81,096</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-04</td>
<td>uninstall</td>
<td>20</td>
<td>81,099</td>
<td>0</td>
<td>2</td>
<td>1.00</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Average \( 1.00 \quad 0.96 \quad 0.98 \quad 1.00 \)

The overall \( F_1 \)-measure \((F_1)\) score for all disk images used for known data set testing was 0.98. This score is a harmonic mean of the combined precision \((P)\) and recall \((R)\) scores. A perfect accuracy score \((1.00)\) was achieved, meaning that no digital artifacts were incorrectly classified by \textit{Vestigium}. A perfect precision score \((1.00)\) was also achieved for all 13 target disk images, meaning that no false positive matches were erroneously detected. This is important as the results presented from the \textit{Vestigium} tool contain no incorrect matches that would require manual analysis to determine correctness. In terms of detection capability, \textit{Vestigium} automated the detection of 492 file system entries out of a possible total of 514 entries from the application profiles. This resulted in an average recall score of 0.96, or 96\% of all digital artifacts present in the target data set. The only reductions to recall scores were observed in the uninstallation phase for all three anti-forensic tools. Further analysis of the uninstallation phase is therefore required.

Manual analysis of \textit{Vestigium} output was performed on each disk image for the uninstallation life cycle phase (CC-04, ER-04 and TC-04). Testing revealed that a number of false negatives were present on each disk image, meaning that a digital artifact that was present in the application profile was not found in the target data set. Investigation showed that the inability to detect all digital artifacts was caused by some file system entries on the target data set having incomplete metadata; for example, files of interest were discovered with missing file size, meta\_type and hash values (both \textit{MD5} and \textit{SHA1}). Furthermore, some directories were discovered with an incorrect meta\_type; for example, the \textit{TrueCrypt} user configuration folder was discovered with a meta\_type of 3 (an input/output file type), when in fact it is a directory with a meta\_type of 2. Listing 6.10 displays two \texttt{FileObjects} that were present but not detected due to the discussed missing metadata properties.
The same problems were observed for the CCleaner and Eraser tools. However, a higher number of digital artifacts from both tools was not detected in the target disk images. A total of 7 and 13 file system entries were not detected on the CCleaner and Eraser uninstallation disk images respectively. From these results it can be concluded, that in some scenarios metadata correlation is not a robust method for detection of deleted digital artifacts. The reason for missing metadata properties is caused by incomplete file system metadata after deletion, where 100% of the content is not recoverable by the fiwalk tool. Nevertheless, a high number of digital artifacts were still detected from all uninstallation disk images, with recall scores ranging from 0.73 to 0.91. This means that 73% to 91% of all deleted file system entries were still detected even after tool uninstallation. Since each digital artifact is unique to the anti-forensic tool, this is compelling digital evidence that reliably supports the presence of an anti-forensic tool on the target system and that it had also been uninstalled.

6.6.4.2 Matching Effectiveness: Registry Entries

Matching Registry entries incorporates the correlation of Registry entries (keys and values) between the application profile(s) and the target data set. Table 6.17 displays an overview of the results from Registry artifact matching. Similar to file system matching, the results table conveys the classified digital artifacts (tp, tn, fp, fn) and each of the four prescribed effectiveness metrics (P, R, F1, A).

Once again, the Vestigium tool with input from the finalised application profiles proved highly effective when executed against the selection of disk images from the known data set.
Digital artifact matching was effective in terms of detection of relevant Registry entries from the application profiles, while also reporting no false positive matches. Again, no digital artifacts were detected in the ground truth (GT-01) disk image and no entries from an anti-forensic tool were detected on a data set where the tool was not introduced. These are expected results. The overall $F_1$–measure ($F_1$) score for all disk images used for known data set testing was 0.96. The result is slightly lower than that of file system matching (0.98). However, this is still a good result. Again, a perfect accuracy score was obtained. Registry analysis resulted in automated detection of 1,271 entries, while only missing 125. Again, a perfect precision score (1.00) was achieved, meaning that no false positive matches were erroneously detected. The average recall score for all disk images was 0.92, meaning that 92% of all Registry entries were detected successfully. Similar to the results from file system analysis, perfect recall scores were observed in the install, open and close life cycle phases, apart from TrueCrypt which had six false negative results.

Further analysis was conducted and unusual Registry entry behaviour was found for the six entries for TrueCrypt that were not detected, even though they were present in the application profile. Manual analysis revealed these entries did not exist in the target data set. The following listing outlines the six non-existent Registry entries.

| Key: | %controlset%\enum\root\legacy_truecrypt\0000\control |
| Value: | %controlset%\enum\root\legacy_truecrypt\0000\control\*newlycreated* |
| Key: | %controlset%\services\truecrypt\enum |
| Value: | %controlset%\services\truecrypt\enum\0 |
| Value: | %controlset%\services\truecrypt\enum\count |
| Value: | %controlset%\services\truecrypt\enum\nextinstance |

Table 6.17: Effectiveness results for Registry matching for the known data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Phase</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>$P$</th>
<th>$R$</th>
<th>$F_1$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT-01</td>
<td>N/A</td>
<td>0</td>
<td>322,070</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CC-01</td>
<td>install</td>
<td>50</td>
<td>321,680</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-02</td>
<td>open</td>
<td>73</td>
<td>321,657</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-03</td>
<td>close</td>
<td>79</td>
<td>321,651</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>CC-04</td>
<td>uninstall</td>
<td>67</td>
<td>321,589</td>
<td>0</td>
<td>12</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-01</td>
<td>install</td>
<td>127</td>
<td>319,563</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-02</td>
<td>open</td>
<td>136</td>
<td>319,558</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-03</td>
<td>close</td>
<td>136</td>
<td>319,562</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ER-04</td>
<td>uninstall</td>
<td>88</td>
<td>319,654</td>
<td>0</td>
<td>51</td>
<td>1.00</td>
<td>0.63</td>
<td>0.78</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-01</td>
<td>install</td>
<td>136</td>
<td>348,774</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.96</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-02</td>
<td>open</td>
<td>138</td>
<td>348,777</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.96</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-03</td>
<td>close</td>
<td>139</td>
<td>348,866</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.96</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>TC-04</td>
<td>uninstall</td>
<td>102</td>
<td>349,003</td>
<td>0</td>
<td>44</td>
<td>1.00</td>
<td>0.70</td>
<td>0.82</td>
<td>1.00</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
<td>0.92</td>
<td>0.96</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Research revealed that the six Registry entries were all related to starting a new service or initialisation of a new device on the Windows operating system (Carvey, 2011). Further informal testing revealed that the six Registry entries are created when installing TrueCrypt but deleted soon after installation (usually after a set time or system reboot). Therefore, the entries were present in the application profile but not in the target data set. Since the entries have a short life-span, their inclusion in the application profile is not required.

Further analysis of Vestigium output was performed on each disk image from the uninstallation life cycle phases (CC-04, ER-04 and TC-04). Similar to file system matching, this was conducted in an attempt to determine the reasoning for the high number of false negatives (entries that were not detected). Once again, the reason was due to incomplete metadata for deleted entries. However, the missing metadata properties were different from that observed in file system matching. The main problem with deleted Registry entry correlation was due to incomplete Registry path information; for example, the TrueCryptFormat Registry key was recovered, but the full Registry path (software\classes\truecryptformat) was not included in the recovered information. Since only the key name was recovered, Registry path matching could not be achieved. In this example, it could be possible to detect the recovered entry as the key name (TrueCryptFormat) is unique and known to be associated with TrueCrypt. However, not all recovered entries have the same unique properties available. Listing 6.11 displays two recovered Registry entries with incomplete metadata. The second CellObject provides an example on an entry that would not be able to be correlated, as the recovered value name (AccessPermission) is not unique to TrueCrypt. Therefore, the situation of incomplete metadata for Registry entries is a difficult matching scenario with no robust solution available in the system implementation. Listing 6.11 does not include the byte_runs element for each CellObject due to space restrictions.

```xml
<cellobject>
  <cellpath>TrueCryptFormat</cellpath>
  <basename>TrueCryptFormat</basename>
  <name_type>k</name_type>
  <alloc>0</alloc>
  <mtime>2015-12-25T22:56:49.1262510Z</mtime>
</cellobject>

<cellobject>
  <cellpath></cellpath>
  <basename>AccessPermission</basename>
  <name_type>v</name_type>
  <alloc>0</alloc>
  <data_type>RegBinary</data_type>
  <data>01 00 04 80 30 00 00 00 40 00 00 00 00 00 00 00 14 00 00 00 02 00 1C 00 01 00</data>
</cellobject>
```

Listing 6.11: Example CellObjects with incomplete metadata properties
6.6.5 Discussion of Findings

The overall outcome from known data set testing in a laboratory controlled environment proved a successful system design and implementation has been achieved. A high recall score was observed for both file system matching and Registry matching, 0.96 and 0.92 respectively. Simply put, approximately 92-96% of all possible digital artifacts were successfully detected. Furthermore, a perfect precision score (1.00) was achieved in every testing scenario. The only observed weakness in the system implementation was seen in the uninstallation phase for all anti-forensic tools in both file system and Registry matching. Even so, the recall scores were still quite high, ranging from 0.63 to 0.91. Given the uniqueness of the detected anti-forensic tool artifacts, even 63% recovery of deleted digital artifacts provide compelling digital evidence of both anti-forensic tool presence on the target data set and evidence of tool uninstallation by the potential suspect. This evidence is particularly useful in a scenario where a suspect may attempt to hide evidence of malicious tool usage by removing and/or deleting the tool.

Previous reference set solutions commonly use file hashes to detect application software usage. This research uses rich metadata correlation to perform automated digital artifact detection; for example, the inclusion of digital artifact name, path, size and allocation status to aid detection. This technique has proved to be functionally capable of aiding detection and able to determine the state of detected digital artifacts; for example, the ability to automatically know if a digital artifact has been deleted and report this useful information to the investigator.

There are no reference set solutions are available to automate forensic analysis, in particular to detect application software. The solution implemented in this research has proven to be effective at detecting a high number of relevant and interesting digital artifacts. Furthermore, there have been no robust solutions previously developed or tested to achieve recovery and matching of deleted Registry entries. The CellXML-Registry tool was specifically authored in this project to recover deleted Registry entries. Based on the results achieved, the implementation can be considered a success by providing an automated analysis technique to detect a high proportion of relevant digital artifacts.

The results gained from the known data set testing in a laboratory controlled environment has demonstrated the functionality of the system implementation. However, during experimental testing a number of problems were encountered that require system modification before progressing to the next phase of the design science process model; that is, system evaluation.

6.6.6 Design Modifications to Digital Artifact Matching

Feedback on the system implementation was achieved through experimental testing on the disk images that comprised the known data set. The input data was approximately 130 GB which contained over 1 million file system entries and over 4 million Registry entries. Testing digital artifact matching on such a large number of file system and Registry entries provided insight on a number of programming bugs in the Vestigium tool, as well as the underlying
CellXML-Registry tool. During known data set testing the encountered programming bugs were resolved.

A variety of programming bugs in Vestigium arose which caused the software to unexpectedly crash during processing. A number of these problems were triggered by lack of error checking when processing digital artifact metadata properties; for example, it was expected that all FileObjects would have a file size property and error checking neglected to be included. Testing revealed that some unallocated (deleted) data files lacked a file size property, causing Vestigium to crash when attempting to query the file size. To combat this problem, extensive error checking was included when a problem was witnessed during testing by simply checking that a FileObject or CellObject has data populated in a metadata property before performing any action. This was achieved by checking the metadata property available; for example, the Python statement: if datafile.filesize checks that the datafile variable has content.

A variety of XML bugs were also discovered in Vestigium, most of which caused the program to crash during target data set processing. XML processing is very strict. Any malformed XML element or any XML element with the incorrect data type (e.g., string instead of integer) caused Vestigium to crash. XML processing errors were not encountered during file system process using fiwalk. However, a number of errors did arise during Registry processing using CellXML-Registry. One example was special characters found in the Registry value basename property. Although special character checks were included for the cellpath property, the basename property was overlooked. All encountered errors were then fixed and the updated version of CellXML-Registry was able to process every Registry hive file without encountering any processing errors.

6.7 Overview of Achieved Research Objectives

System demonstration has progressed towards solving the high-level and various lower-level objectives specified in this research project. The experimental testing results have solved lower-level research objectives one and two, while also providing preliminary results for lower-level research objective three and four. This section has outlined the progress made thus far for each stated research objective.

**Lower-level Research Objective One:** To enable effective and efficient automated identification of relevant digital artifacts from application software

Demonstration has produced experimental testing results to prove that the LiveDiff tool was functionally capable of automated identification of relevant digital artifacts. Section 6.3 created a selection of application profiles using the implemented software based on the system design. Initial application profile creation judged the solution highly efficient when compared to other solutions presented in similar research (see Section 6.3.2). Section 6.4 worked towards identification of only relevant digital artifacts that are unique to each anti-forensic tool using two filtering techniques, both of which proved effective at removing
irrelevant application profile entries. Section 6.5.1 outlined further refinements to the design and resulted in a finalised enhancements to LiveDiff, which was again tested via application profile creation. The outcome was three application profiles containing only relevant digital artifacts, suitable for identification of known content on a target data set.

**Lower-level Research Objective Two:** To design an effective data abstraction for storing, distributing and automating the processing of different digital artifact types from application softwares.

This chapter demonstrated the design and implementation of the Application Profile XML (APXML) data abstraction to store, distribute and automate processing of identified digital artifacts from reverse engineering. The APXML data abstraction provides the functionality to store both file system and Registry entries while at the same time has the ability to distribute to other researchers or practitioners. Furthermore, the associated APXML API (apxml.py) proved effective at providing automated processing functionality of application profile contents. This was demonstrated in filtering irrelevant digital artifacts from the created application profile using simple Python scripts to perform automated intersection (see Section 6.4.1.1) and automated blacklisting (see Section 6.4.2.1). Furthermore, the APXML data abstraction was again leveraged to aid automated digital artifact matching when parsing the application profile for comparison against the target data set.

Lower-level research objectives one and two are now considered solved. In addition, lower-level research objectives three and four have, in part, been addressed in the known data set testing portion of system demonstration (see Section 6.6.4). Lower-level objective three specified an effective and efficient automated technique to generate a metadata representation of the target data set. This chapter has demonstrated the functionality of Vestigium to achieve this, coupled with the fiwalk and CellXML-Registry tools. However, further experimental testing and a thorough evaluation is still required to determine overall effectiveness of the developed solution. In addition, efficiency results have not yet been presented but will be covered in detail in the system evaluation chapter (see Chapter 7). Lower-level objective four specified for effective and efficient automated methods to correlate different digital artifact types between an application profile and target data set. Again, this chapter has demonstrated the implemented Vestigium tool is able to achieve the desired functionality in a controlled laboratory environment. Nevertheless, further experimental testing is still required followed by a thorough evaluation of the system implementation.

### 6.8 Conclusion

The goal of the fourth element of the DSRM process model is to demonstrate the functionality of the designed and implemented system in a controlled laboratory environment. Operation of all tools (computer software) authored in this research was detailed, each used as an individual component in the implemented system. An experimental testing method prescribed an environment using virtual machines to perform data generation and collection while a method to
establish the ground truth digital artifacts on the prescribed data sets was outlined. Testing metrics were revisited and score interpretation examples provided.

The system design was demonstrated firstly, to test the creation of application profiles using LiveDiff, including the use of filtering techniques to only identify relevant file system and Registry entries. A revised creation method was specified, refinements to LiveDiff made and three finalised application profiles were created. Secondly, demonstration was used to test digital artifact matching functionality. A data set was created, populated with known content and the digital artifact matching stage tested using the Vestigium tool. Effectiveness metrics proved the functionality of the implemented system and led to refinements. Finally, a selection of lower-level research objectives were reviewed, some were solved in full, others in part, leading to further experimental testing. Comprehensive evaluation of the refined system against a public and real-world data set will now take place.
The final element in the Design Science Research Methodology (DSRM) process model is the evaluation of the implemented system. According to Peffers et al. (2007), evaluation includes observing and measuring how well the designed artifact supports a solution by comparing the research objectives to the observed results using relevant analysis techniques and metrics. Therefore, this chapter describes the experimental testing performed leading to system evaluation. Two data sets were selected to perform evaluation: 1) A publicly available data set designed for digital forensics research to enable reproducible findings; and 2) A real-world data set comprised of second-hand hard drives with diverse and unpredictable data. Each data set was used as input to the Vestigium tool, digital artifact matching executed and metrics calculated to evaluate the system effectiveness and efficiency. Comparisons are made to de facto forensic analysis techniques to evaluate the system compared to existing solutions.

The first half of the chapter is dedicated to public data set testing using the M57-Patents scenario. An overview is provided including the testing method and establishment of ground truth. Testing is conducted and results presented for effectiveness and efficiency of the overall system. The findings are discussed and refinements made to the system based on the problems and limitations. The second half of the chapter is dedicated to real-world data set testing using a selection of second-hand hard drives. An overview of the data set is provided including a selection criteria and a summary of the experimental testing method. Findings are again, presented in terms of effectiveness and efficiency. Finally, a section is dedicated to a discussion on the evaluation of the system to solve the prescribed research objectives.
7.1 Public Data Set Overview: M57-Patents Scenario

The first data set used for system evaluation is the publicly available M57-Patents scenario (see Section 4.3.3.2). A public data set was selected because it offers research reproducibility; specifically the capability for other researchers, tool developers and forensic practitioners to test, evaluate, verify and/or build on the findings presented in this research. An overview of the M57-Patents scenario, the implemented experimental testing method and a summary of the established ground truth of the data set is outlined. The information provides insight into the data set contents and leads to the next two sections which convey the results from experimental testing.

7.1.1 Overview of M57-Patents Scenario

According to Digital Corpora (2011), the M57-Patents scenario was created in 2009 by the Naval Postgraduate School in Monterey, California, United States. The M57-Patents scenario is freely available for download from the Digital Corpora website. The scenario features a fictitious patent research company, m57.biz that consists of four employees: 1) Pat McGoo, the company CEO; 2) Terry Johnson the IT administrator; 3) Jo Smith a patent researcher; and 4) Charlie Brown another patent researcher. The data set was created over a 17-day period between November 16th, 2009 (2009-11-16) and December 11, 2009 (2009-12-11) which excludes weekends and holidays.

This research aims to detect application software artifacts from desktop computer systems running versions of the Microsoft Windows operating system. Therefore, the material from the M57-Patents scenario that is now of use for the evaluation of this system design is the 79 forensic images from desktop computer systems of the four users running different Microsoft Windows operating systems. Table 7.1 provides an overview of desktop systems in the M57-Patents scenario based on the computer user. The user is displayed with the associated number of disk images in the data set, the size of the hard disk (HDD) and the Microsoft Windows version.

<table>
<thead>
<tr>
<th>User</th>
<th>Disk Images</th>
<th>HDD Size</th>
<th>Windows Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pat</td>
<td>19</td>
<td>12 GB</td>
<td>Windows XP</td>
</tr>
<tr>
<td>Terry</td>
<td>20</td>
<td>19-38 GB</td>
<td>Windows Vista</td>
</tr>
<tr>
<td>Jo</td>
<td>21</td>
<td>12-14 GB</td>
<td>Windows XP</td>
</tr>
<tr>
<td>Charlie</td>
<td>19</td>
<td>9.5 GB</td>
<td>Windows XP</td>
</tr>
</tbody>
</table>

1See: http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario
2Any discussion regarding individual disk images from the M57-Patents scenario uses the user name and date to identify the specific disk images from the data set; for example, charlie-2009-11-12 is a disk image from the user Charlie which was collected on the 12th of November, 2009. ISO 8601 date conventions (YYYY-MM-DD) are used as they correlate to the naming method used by the M57-Patents scenario.
3The initial size of Terry’s hard drive was 19 GB, which increased to 38 GB on 2009-11-19.
4The initial size of Jo’s hard drive was 12 GB, which increased to 14 GB on 2009-11-20.
The M57-Patents scenario includes two Microsoft Windows operating system versions: Windows XP and Windows Vista. This is important to consider during system evaluation as the created application profiles were authored on a system running Microsoft Windows 7. It presents a significant challenge for the application profiling system and will aid in producing findings to determine achievement, or otherwise, of lower-level research objectives five and six (see Section 4.2.2.4).

Prior to performing experimental testing, the M57-Patents scenario was analysed to determine the size and complexity of the data set. This was achieved by using the same two Python scripts authored to determine known data set content (see Section 6.6.1). The file system statistics script (FileSystemStats.py) and Windows Registry statistics script (RegistryStats.py) are available in Appendix C.8 and C.9. Table 7.2 presents an overview of the M57-Patents scenario including the size of the data set (disk images) and summary information regarding the number and type of file system and Windows Registry entries. The average per disk (of the 79 disks) for each property is also provided. Appendix D.1 provides more in-depth information regarding M57-Patents scenario content.

<table>
<thead>
<tr>
<th>Property</th>
<th>Total</th>
<th>Average per disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set size</td>
<td>1467.71 GB</td>
<td>18.58 GB</td>
</tr>
<tr>
<td>Directories</td>
<td>1,402,729</td>
<td>17,756</td>
</tr>
<tr>
<td>Data files</td>
<td>3,711,083</td>
<td>46,976</td>
</tr>
<tr>
<td>Other</td>
<td>124,688</td>
<td>1,578</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>4,890,757</td>
<td>61,908</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>347,743</td>
<td>4,402</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5,238,500</strong></td>
<td><strong>66,310</strong></td>
</tr>
<tr>
<td>Hive files</td>
<td>1,267</td>
<td>16</td>
</tr>
<tr>
<td>Registry keys</td>
<td>10,382,107</td>
<td>131,419</td>
</tr>
<tr>
<td>Registry values</td>
<td>18,956,843</td>
<td>239,960</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>28,699,932</td>
<td>363,290</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>639,018</td>
<td>8,089</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>29,338,950</strong></td>
<td><strong>371,379</strong></td>
</tr>
</tbody>
</table>

The 79 forensic disk images from the M57-Patents scenario equate to approximately 1.5 Terabytes (TB) of raw data. The data set is distributed in the Expert Witness Format (EWF), commonly known by the .E01 file extension. The ewfinfo tool from the libewf project revealed that the forensic images were compressed using the deflate compression method and best compression option selected. The total size of the compressed forensic disk images equates to approximately 454 GB. Compression of the target data set is important to note as it may adversely affect the performance (computational efficiency) achieved during testing since the

---

5See: https://github.com/libyal/libewf/
forensic image needs to be decompressed on the fly when processing.

Each of the 79 forensic disk images from the M57-Patents scenario will be used in different aspects of testing the digital artifact matching component of the system design (the Vestigium tool). However, before undertaking experimental testing, the ground truth of the data set needs to be established to determine the anti-forensic tool presence, or absence, on the disk images from the data set.

### 7.1.2 Experimental Testing Method

The experimental testing method used for the M57-Patents scenario is the same as implemented for known data set testing (see Section 6.6.2). The Vestigium tool is used to correlate digital artifacts between the application profiles (APXML documents) and the target data set (each forensic disk image from the M57-Patents scenario). Listing 7.1 provides an example of the command to invoke Vestigium using one disk image (charlie-2009-11-12.E01) and a user-specified output folder (charlie-2009-11-12-output). All three application profiles are included for every disk image tested. The listing specifies new line characters (\n) and comments (REM) using Windows command conventions.

```bash
$ C:\Python34\python.exe Vestigium.py ^
D:\M57\charlie-2009-11-12.E01 ^ REM Forensic disk image
D:\M57\charlie-2009-11-12-output ^ REM Output folder
CCleaner-5.09-6.1.7601.apxml ^ REM CCleaner profile
Eraser-6.2.0.2970-6.1.7601.apxml ^ REM Eraser profile
TrueCrypt-7.1a-6.1.7601.apxml REM TrueCrypt profile
```

Listing 7.1: Command example to invoke Vestigium against the M57-Patents scenario

Although the M57-Patents scenario provides DFXML reports generated by fiwalk for all forensic disk images, the reports were not used as input during testing. The reasoning is that a DFXML report is not always available for investigations. Reports were provided for this data set because the authors of the fiwalk tool are from the same academic institution as the authors of the M57-Patents data set. Therefore, DFXML metadata generation should be performed during tool run-time to provide an accurate representation of tool efficiency.

### 7.1.3 Establishing Ground Truth for the M57-Patents Scenario

The only official documentation included with the M57-Patents scenario are student exercises, associated instructor material and fictitious legal documents. Therefore, in terms of installed application software, the M57-Patents scenario has no documentation that specifies the content of the data set; for example, information relating to applications that are present on each user’s system, dates the applications were installed, a list of actions performed, and, if or when applications were uninstalled. However, previous research output does provide

---


7See: [http://digital.corpora.org/corpora/scenarios/m57-patents-scenario](http://digital.corpora.org/corpora/scenarios/m57-patents-scenario)
some information regarding the presence of specific applications; for example, TrueCrypt is known to be installed on at least one user’s system: Roussev and Quates (2012) stated that, using the sdhash tool, they discovered a user (Jo) had used the TrueCrypt software. However, Roussev and Quates (2012) were attempting to discover illegal digital images and their analysis was not focused on detecting anti-forensic tools. The available documentation of the M57-Patents scenario is therefore incomplete and lacks sufficient detail to establish reliable and accurate information regarding anti-forensic tool presence, or even absence. As a consequence, ground truth needs to be established to provide a baseline count of relevant digital artifacts to determine effectiveness of the implemented system.

The process of establishing ground truth (see Section 6.2.5) involves examining each disk image and: 1) Determining the Windows operating system version (already known); 2) Determining anti-forensic tool presence; and 3) Determining the digital artifacts uniquely associated with each anti-forensic tool and documenting the findings. Using this information a ground truth application profile can be created with which results can be compared.

### 7.1.3.1 Determining Anti-forensic Tool Presence

Initial analysis was conducted by performing a keyword search on each forensic disk image in the M57-Patents scenario. Performing keyword searching aids in identifying disk images that may have a specific anti-forensic tool on the system; for example, searching for the keyword `truecrypt` will likely yield search hits if the tool is present on the target system. To accomplish this, each forensic image was first converted to the raw disk image format using the ewfexport utility from the libewf project. Each converted disk image was then subjected to a keyword search using the strings and grep utilities. Three keywords were used: 1) ccleaner; 2) eraser; and 3) truecrypt. The following commands demonstrate how the disk images were converted and subsequently searched. In the example, the keyword `truecrypt` was explicitly searched using the `-i` flag to perform a case insensitive search.

```
$ ewfexport -f raw D:\M57\charlie-2009-11-12.E01
$ strings D:\M57\charlie-2009-11-12.raw | grep -i "truecrypt"
```

The search output from each disk image was then analysed to determine positive or negative identification. This was achieved by manually analysing the context surrounding each keyword search hit as well as the number of search hits returned. It should be noted that searching raw data does not provide any results to indicate which file system entry or Registry entry the search hit was discovered from. It is also important to note that this search technique will only work with a unique application name. The result from the initial analysis identified thirteen (13) potential disk images (out of a total of 79) that may have at least one of the three selected anti-forensic tools. Table 7.3 displays the interesting forensic images from the

---

8 As previously noted, all the forensic images in the M57-Patents scenario are distributed as compressed EWF files, meaning a disk image cannot be directly searched as any potential keywords (strings) will be compressed. Therefore, each forensic image needs to be uncompressed and then searched.

9 A total of eight potentially interesting forensic images were discovered from the user Terry. However, three disk images contained keyword search hits from both CCleaner and Eraser tools. This results in a
M57-Patents scenario including the username, operating system version, inclusive dates of interesting content, count of forensic images and the associated keyword search hit.

Table 7.3: Potentially interesting forensic disk images from the M57-Patents scenario

<table>
<thead>
<tr>
<th>User</th>
<th>OS</th>
<th>Dates</th>
<th>Count</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jo</td>
<td>XP</td>
<td>2009-12-03 to 2009-12-11</td>
<td>8</td>
<td>truecrypt</td>
</tr>
<tr>
<td>Terry</td>
<td>Vista</td>
<td>2009-12-08 to 2009-12-11</td>
<td>5</td>
<td>ccleaner</td>
</tr>
<tr>
<td>Terry</td>
<td>Vista</td>
<td>2009-12-10 to 2009-12-11</td>
<td>3</td>
<td>eraser</td>
</tr>
</tbody>
</table>

7.1.3.2 Determining and Documenting Anti-forensic Tool Artifacts

Further investigation was conducted on the 13 interesting disk images from keyword searching. DFXML and RegXML reports were generated for all 13 disk images using the specified ground truth method (see Figure 6.6). Once again, the grep utility was invoked, this time to search all metadata reports for the specified keywords, as displayed in the command example below.

```
$ grep -r -i -n "truecrypt" D:\M57\charlie-2009-11-12\*.xml
```

The output from metadata searching provided additional evidence of anti-forensic tool presence, as well as helping to determine the tool version found on the disk images. Tool version information was found by manually examining the version number embedded in tool installer files (e.g., TrueCrypt Setup 6.3a.exe), as well as the version number stored in varying Registry values. Table 7.4 provides a comparison overview of the anti-forensic tools used for the created application profiles versus the anti-forensic tool versions discovered on the M57-Patents scenario. The anti-forensic tool name, version and operating system is displayed for all three tools.

Table 7.4: Anti-forensic tool versions for application profiles versus the M57-Patents scenario

<table>
<thead>
<tr>
<th>Created</th>
<th>M57-Patents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>Version</td>
</tr>
<tr>
<td>CCleaner</td>
<td>5.09.5343</td>
</tr>
<tr>
<td>Eraser</td>
<td>6.2.0.2970</td>
</tr>
<tr>
<td>TrueCrypt</td>
<td>7.1a</td>
</tr>
</tbody>
</table>

Knowledge of the operating system and anti-forensic tool version provides the ability to create a new application profile (using LiveDiff) that matches the identified anti-forensic tool, the tool version and operating system version; for example, Eraser version 5.7.8 was discovered on the M57-Patents scenario (on Terry’s system) running Microsoft Windows Vista. Therefore, a new application profile was created for Eraser version 5.7.8 using Windows Vista as the host operating system. This new application profile should have the same contents as total of 13 interesting disk images.
appear on the target data set, but comparison should be made to confirm this. Therefore, the entries from the newly created application profile were compared to the contents of the target data set. Any entries deemed a match are then copied to a new application profile (APXML document) which retains the digital artifact metadata taken directly from the target data set. This document contains the actual file system and Registry entries that appear in the disk image and provide an accurate metadata representation of ground truth for the selected anti-forensic tool. Table 7.5 displays a summary of digital artifact counts for each of the identified disk images where one of the three anti-forensic tools were present. The disk image name, anti-forensic tool name and anti-forensic tool version are displayed. The digital artifact count is also provided for directories, files, Registry keys and Registry values and a total digital artifact count.

Table 7.5: Overview of ground truth information for the M57-Patents scenario

<table>
<thead>
<tr>
<th>Name</th>
<th>Tool</th>
<th>Version</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-03</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>11</td>
<td>26</td>
<td>55</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td>jo-2009-12-08</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>40</td>
<td>73</td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>40</td>
<td>73</td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>TrueCrypt</td>
<td>6.3a</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>40</td>
<td>73</td>
</tr>
<tr>
<td>terry-2009-12-08</td>
<td>CCleaner</td>
<td>2.26</td>
<td>3</td>
<td>50</td>
<td>14</td>
<td>25</td>
<td>92</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>CCleaner</td>
<td>2.26</td>
<td>3</td>
<td>50</td>
<td>14</td>
<td>25</td>
<td>92</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>CCleaner</td>
<td>2.26</td>
<td>3</td>
<td>49</td>
<td>14</td>
<td>25</td>
<td>91</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>CCleaner</td>
<td>2.26</td>
<td>3</td>
<td>48</td>
<td>14</td>
<td>25</td>
<td>90</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>CCleaner</td>
<td>2.26</td>
<td>3</td>
<td>47</td>
<td>14</td>
<td>25</td>
<td>89</td>
</tr>
<tr>
<td>terry-2009-12-12-10</td>
<td>Eraser</td>
<td>5.7.8</td>
<td>3</td>
<td>15</td>
<td>49</td>
<td>86</td>
<td>153</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>Eraser</td>
<td>5.7.8</td>
<td>3</td>
<td>14</td>
<td>49</td>
<td>86</td>
<td>152</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser</td>
<td>5.7.8</td>
<td>3</td>
<td>14</td>
<td>49</td>
<td>86</td>
<td>152</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>56</td>
<td>399</td>
<td>333</td>
<td>685</td>
<td>1,473</td>
</tr>
</tbody>
</table>

The information obtained has resulted in accurate documentation of ground truth for the M57-Patents scenario for the three selected anti-forensic tools. For each disk image and associated tool, an APXML document has been manually created which contains a metadata representation of the unique digital artifacts associated with each anti-forensic tool. This results in a machine-readable document to aid experimentation and system evaluation. These generated reports have been made publicly available for other researchers and practitioners to aid future development.\[10\]

\[10\]See: [https://github.com/thomaslaurenson/Vestigium/tree/master/M57-GroundTruth](https://github.com/thomaslaurenson/Vestigium/tree/master/M57-GroundTruth)
7.2 M57-Patents Scenario: Effectiveness Testing and Results

This section presents the effectiveness results from public data testing using the M57-Patents scenario. Firstly, the three created application profiles from the selected anti-forensic tools are executed against the selected 13 interesting disk images from the M57-Patents scenario and effectiveness measured. Comparisons of file system matching and Registry matching success are compared to de facto forensic analysis techniques to provide insight into system effectiveness. Secondly, the three application profiles are executed against the remaining non-interesting disk images (a total of 66) from the M57-Patents data set to determine potential false positive detection rates where all three anti-forensics tool are known to be absent. Thirdly, due to the lack of uninstalled anti-forensic tools present in the initial experiments, an additional test is conducted on an anti-forensic tool known to be installed and subsequently uninstalled during the scenario. A newly created application profile for the XP Advanced Keylogger tool is additionally used against the 19 disk images from the user Pat. This provides the ability to evaluate the implemented system design using a longitudinal data set to determine residual digital artifact detection. The results are compared against a similar research study by Jones et al. (2015) that also performed testing using the M57-Patents scenario. The proposed and implemented path normalisation technique is then investigated to determine the effectiveness in regards to aiding file system path and Registry path correlation on different Windows operating system versions. Finally, the Windows Registry processing technique, specifically the CellXML-Registry tool, is analysed to determine effectiveness at representing the original Registry evidence source, as well as the ability to recover deleted Registry entries. Comparisons are made to a similar solution by Nelson (2012), which also performed testing using the M57-Patents scenario.

7.2.1 Detecting Anti-forensic Tools using Vestigium

The first experiment using the M57-Patents data set was conducted to determine the overall effectiveness of the implemented system to detect the three selected anti-forensic tools, namely: 1) CCleaner; 2) Eraser; and 3) TrueCrypt. A total of 13 disk images from the M57-Patents scenario were selected for testing as they were previously manually analysed and deemed interesting, having at least one of the three selected anti-forensic tools installed (see Table 7.5). Each disk image was used as input to the Vestigium tool.

Output from Vestigium indicated the presence of all three anti-forensic tools on at least one of the selected disk images. The user Jo was found to have installed TrueCrypt on 2009-12-03 and the tool was present on Jo’s system until 2009-12-11 (the end of the scenario). The user Terry was found to have installed CCleaner on 2009-12-08 and Eraser on 2009-12-10 (again, both tools were present until the end of the scenario). These results indicate successful detection of all anti-forensic tools on a total of 13 disk images from two users. Table 7.6 provides a high-level overview of the results from the M57-Patents scenario including the user, operating system, date range of detected tools, count of the number of disk images with anti-forensic tools, the tool detected and the application life cycle phases found.
Table 7.6: High-level overview of Vestigium results for the M57-Patents scenario

<table>
<thead>
<tr>
<th>User</th>
<th>OS</th>
<th>Dates</th>
<th>Count</th>
<th>Profile</th>
<th>Detected Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jo</td>
<td>XP</td>
<td>12-03 to 12-11</td>
<td>8</td>
<td>TrueCrypt</td>
<td>install, open, close</td>
</tr>
<tr>
<td>Terry</td>
<td>Vista</td>
<td>12-08 to 12-11</td>
<td>5</td>
<td>CCleaner</td>
<td>install, open, close</td>
</tr>
<tr>
<td>Terry</td>
<td>Vista</td>
<td>12-10 to 12-11</td>
<td>3</td>
<td>Eraser</td>
<td>install, open, close</td>
</tr>
</tbody>
</table>

In order to establish the overall performance of Vestigium, the 13 interesting forensic images were analysed further and findings reported in detail. The following subsections outline and discuss the effectiveness of the implemented system divided into file system matching effectiveness followed by Windows Registry matching effectiveness.

### 7.2.1.1 File System Matching Effectiveness

Matching file system entries incorporates the correlation of file system directories and data files between the application profile(s) and the target data set. Table 7.7 displays the results from file system matching for target disk images from the M57-Patents scenario. The disk image name is provided with the associated application profile name and version number. The four classifiers for digital artifacts are included: 1) True positive \((t_p)\); 2) True negative \((t_n)\); 3) False positive \((f_p)\); and 4) False negative \((f_n)\). The four prescribed effectiveness metrics are also provided: 1) Precision \((P)\); 2) Recall \((R)\); 3) \(F_1\)-measure \((F_1)\); and 4) Accuracy \((A)\). Further information regarding the digital artifact classification scheme is available in Section 4.3.4.2, effective metrics are outlined in Section 4.3.4 and score interpretation examples are available in Section 6.2.6.

The overall results from file system matching indicate an effective system implementation in terms of automated detection of relevant file system entries. All 13 interesting disk images were discovered to have file system entries uniquely associated with one of the three anti-forensic tool profiles. For all 13 disk images a perfect precision score \((1.00)\) was achieved, meaning that no digital artifacts were incorrectly detected and reported to the user. This means that Vestigium did not detect any digital artifacts that were not directly relevant to the application profile and, ultimately, the associated anti-forensic tool. This is an excellent result as there is no need for a practitioner to perform additional manual analysis on tool output to determine correct and incorrect matches. The complete lack of false positive results (perfect precision score) was achieved due to the refined and accurate contents of the created application profiles. Data reduction techniques (see Section 6.4 and Section 6.5) proved effective at identifying and subsequently populating each application profile with only digital artifacts deemed unique to the anti-forensic tool. This in turn resulted in a clean application profile without erroneous entries. The matching methods implemented in Vestigium also aided in returning no false positive rates. Matching multiple metadata properties has proven effective at detecting relevant entries while those digital artifacts that are not a match are not detected.
Table 7.7: Effectiveness results for file system matching for the M57-Patents Scenario

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>P</th>
<th>R</th>
<th>F1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-03</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>33485</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>33454</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>33434</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-08</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>33422</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>33326</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>35317</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>35341</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>TrueCrypt-7.1a</td>
<td>17</td>
<td>35400</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-08</td>
<td>CCleaner-5.09</td>
<td>50</td>
<td>118113</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>CCleaner-5.09</td>
<td>50</td>
<td>118231</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.94</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>CCleaner-5.09</td>
<td>50</td>
<td>118017</td>
<td>0</td>
<td>2</td>
<td>1.00</td>
<td>0.96</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>CCleaner-5.09</td>
<td>50</td>
<td>118019</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>CCleaner-5.09</td>
<td>50</td>
<td>117209</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>Eraser-6.2.0</td>
<td>3</td>
<td>118051</td>
<td>0</td>
<td>15</td>
<td>1.00</td>
<td>0.17</td>
<td>0.29</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>Eraser-6.2.0</td>
<td>2</td>
<td>118053</td>
<td>0</td>
<td>15</td>
<td>1.00</td>
<td>0.12</td>
<td>0.21</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser-6.2.0</td>
<td>2</td>
<td>117242</td>
<td>0</td>
<td>15</td>
<td>1.00</td>
<td>0.12</td>
<td>0.21</td>
<td>1.00</td>
</tr>
</tbody>
</table>

| Average      | 1.00 | 0.80 | 0.84 | 1.00 |

Overall, the average recall score for all disk images was 0.80, meaning that 80% of all ground truth digital artifacts were detected. For CCleaner and TrueCrypt recall was higher, an average of 0.95, or 95% of the total digital artifacts. In contrast, the average recall for Eraser was 0.14. This means that only 14% of the known ground truth digital artifacts were detected (reasoning for this result is discussed below in detail). The overall average F1-measure score was 0.84 and represents a balanced score of recall and precision. All accuracy scores achieved a perfect score (1.00) as no digital artifacts were incorrectly classified by Vestigium which would alter the accuracy score outcome.

The high recall score dictates that Vestigium was able to detect digital artifacts from an application profile authored on a different operating system using a different anti-forensic tool version. This was achieved by the functionality of Vestigium to correlate multiple digital artifact properties (e.g., file system path, hash value, allocation status) and not relying directly on the data file hash value for correlation. In addition, path normalisation proved effective at detecting digital artifacts with a file system path that had slight variations between the application profile and target data set (see Section 5.5.3.1). Listing 7.2 displays three data files that were only detectable using the implemented path normalisation technique. In each example the full path of the file system entry is displayed for both the target data set (jo-2009-12-03.E01), the application profile (TrueCrypt-7.1a-6.1.7601.apxml) and the associated normalised file system path. As displayed, there were variations between each file system path for the target and application profile. However, in each case the file path was able to be correlated using path normalisation. This technique resulted in providing
a generalised value that is able to be correlated between different targets. A good example of the flexibility of path normalisation was the ability to detect a file system path with an embedded username (see the first example in the listing). A full analysis of path normalisation effectiveness is presented later in Section 7.2.4.

<table>
<thead>
<tr>
<th>Target Data Set</th>
<th>Application Profile</th>
<th>Normalised Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\Users\forensic\AppData\Roaming\TrueCrypt</td>
<td>C:\documents and settings\jo\application data\truecrypt</td>
<td>%appdata%/truecrypt</td>
</tr>
<tr>
<td>C:\documents and settings\all users\desktop\truecrypt.lnk</td>
<td>C:\users\public\desktop\truecrypt.lnk</td>
<td>%allusersprofile%/desktop/truecrypt.lnk</td>
</tr>
<tr>
<td>C:\windows\prefetch\truecrypt.exe-3a2a0f93.pf</td>
<td>C:\windows\prefetch\truecrypt.exe-009a2e5a.pf</td>
<td>%prefetch%/truecrypt.exe.pf</td>
</tr>
</tbody>
</table>

Listing 7.2: Comparison of file system path differences between the target data set and application profiles

**CCleaner:** The overall results for CCleaner showed a high recall and perfect precision score (0.96 and 1.00 respectively). The results indicate the capability to detect the CCleaner tool using an application profile authored on a different Windows-based operating system (Window 7 versus Windows XP). Furthermore, the CCleaner version was also different (version 5.09 versus version 2.26). As stated, these results were achievable due to the advanced digital artifact matching methods and path normalisation implemented in the Vestigium tool. A total of three file system entries (out of 53) were not detected in the target data set: 1) A Windows Prefetch file with a different embedded application version number; 2) A Windows Shortcut file that existed in the target data set but not in the application profile; and 3) Another Windows Shortcut file that was stored in a different file system location. Both Windows Shortcut file differences were caused by the way CCleaner created Shortcut files between each application version. In CCleaner version 2.26, the Shortcut file was created for the user who installed the application, while in version 5.09, the Shortcut file is automatically created for every user (using the All Users user profile). An example of the varying normalised file path for this scenario is displayed below.

<table>
<thead>
<tr>
<th>Target Data Set</th>
<th>Application Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>%userprofile%/desktop/ccleaner.lnk</td>
<td>%allusersprofile%/desktop/ccleaner.lnk</td>
</tr>
</tbody>
</table>

**Eraser:** The overall results for Eraser showed a low recall score (average of 0.14). The results were investigated and research conducted in an attempt to determine the cause of the low scores. It was discovered that major changes to Eraser had occurred between the version used for creating the application profile (version 6.2.0) versus the version on the target data set (version 5.7.8). Basically, Eraser was rewritten between the two versions;
version 5.7.8 was authored in Visual C++, while version 6.2.0 is authored in C# using the .NET framework\footnote{The only reference found by the author regarding the changes made to Eraser development is available from: \url{http://eraser.heidi.ie/forum/threads/for-the-greater-good-of-eraser.4302/}}. The dramatic changes to application redesign and operation resulted in a considerable difference in digital artifacts created by the software. Ultimately, these changes effected a low recall rate as the digital artifacts present in the application profile were not present on the target data set. An additional experiment was conducted using an application profile authored on Windows 7 using \texttt{Eraser} version 5.7.8. The results are displayed in Table 7.8.

\begin{table}[h]
\centering
\begin{tabular}{lrlrrrr}
\hline
\textbf{Name} & \textbf{Profile} & \textbf{tp} & \textbf{tn} & \textbf{fp} & \textbf{fn} & \textbf{P} & \textbf{R} & \textbf{F}_1 & \textbf{A} \\
\hline
terry-2009-12-10.E01 & Eraser-5.7.8 & 18 & 118051 & 0 & 0 & 1.00 & 1.00 & 1.00 & 1.00 \\
terry-2009-12-11-001.E01 & Eraser-5.7.8 & 17 & 118053 & 0 & 0 & 1.00 & 1.00 & 1.00 & 1.00 \\
terry-2009-12-11-002.E01 & Eraser-5.7.8 & 17 & 117242 & 0 & 0 & 1.00 & 1.00 & 1.00 & 1.00 \\
\hline
\end{tabular}
\caption{Overview of file system matching results for \texttt{Eraser} version 5.7.8}
\end{table}

The results from the newly authored \texttt{Eraser} profile using version 5.7.8 returned perfect scores for recall and precision, 1.00 and 1.00 respectively. All file system entries were discovered in each of the three data sets from the user Terry that were known to have \texttt{Eraser} installed (established from ground truth data set information). Creating an application profile for every application version is not feasible. However, in this case \texttt{Eraser} had undergone such dramatic changes that the application could be classified as a different tool. Nevertheless, in circumstances such as this, an application profile needs to be created for each tool version. This is an excellent example of real-world application development and the impact it can have on digital forensic techniques.

\textbf{TrueCrypt:} The overall results for \texttt{TrueCrypt} showed a high recall and perfect precision scores, 0.94 and 1.00 respectively. Detailed investigation was performed to determine the only digital artifact (out of 18) that was not detected, a Windows Prefetch file. The listing below displays the actual and normalised file path for both the target data set file and application profile file:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Target Data Set: & C:\WINDOWS\Prefetch\TRUECRYPT SETUP 6.3A.EXE-05F795CC.pf \\
Normalised path: & %PREFETCH%/TRUECRYPT SETUP 6.3A.EXE.pf \\
\hline
Application Profile: & C:\Windows\Prefetch\TRUECRYPT SETUP 7.1A.EXE-9732BAC6.pf \\
Normalised path: & %PREFETCH%/TRUECRYPT SETUP 7.1A.EXE.pf \\
\hline
\end{tabular}
\caption{Actual and normalised file paths for \texttt{TrueCrypt}}
\end{table}

The comparison identifies that even with path normalisation, the Windows Prefetch file could not be automatically detected due to the embedded version number in the anti-forensic tool installer file (7.1a version 6.3a). The same problem was previously discovered and discussed for \texttt{CCleaner}. Apart from the identified prefetch file all other file system entries for \texttt{TrueCrypt} were detected correctly.
In summary, the results from file system matching dictate that the implemented system design proved effective at detecting file system entries in a target data set. Furthermore, detection was possible even where the application version and host operating system were different. The variation between file system paths were observed and correlation achieved using path normalisation. The only low score was the recall rate for Eraser. Investigation revealed that the application had been completely rewritten and resulted in a considerable difference between the tool versions and the digital artifacts created by each version. The following subsection provides a comparison of file system entry detection to a similar solution, that is, file hashing.

7.2.1.2 File System Matching Effectiveness Comparison: Hash Sets

The file system matching method presented in this research can be compared to the industry standard forensic analysis technique to automate detection of known data files, namely: **hash analysis**, also known as **known file filtering** (see Section 3.3.1). Hash analysis involves generating a set of known interesting file hashes and comparing them to every data file on the target data set. In this research the application profile has a hash value for every data file known to be associated with each of the three anti-forensic tools. A list of hashes can be extracted and used to compare to the target data set to attempt to identify entries of forensic interest.

In order to practically accomplish this, a simple Python script was authored to ingest an APXML document, create a set of known file hashes (`known_md5s`) and compare these to all the data files on the target data set. Listing 7.3 displays the script used to locate hash value matches.

```
# Import required Python modules
import sys, apxml, Objects

# Read Application Profile XML document
apxml_object = apxml.iterparse(sys.argv[1])

# Create a dictionary of known MD5 hashes from application profile
known_md5s = { f.md5: f
              for f in apxml_object
              if isinstance(f, Objects.FileObject) and
              f.meta_type == 1 }

# Locate known files from target disk image
for (event, obj) in Objects.iterparse(sys.argv[2]):
  if isinstance(obj, Objects.FileObject):
    if (obj.md5 in known_md5s):
      print(obj.filename, obj.md5)
```

Listing 7.3: Example of the script used to locate MD5 hash matches

213
The authored script was executed against the same 13 *interesting* disk images from the M57-Patents scenario. The results from testing revealed only a single data file was detected from the TrueCrypt application profile; the same file was detected on 8 disk images, all from the user Jo. The detected data file was deemed a true positive based on the ground truth data. The following listing displays the file name (truncated due to length) and associated MD5 hash value of the discovered file.

<table>
<thead>
<tr>
<th>File name: Start Menu/Programs/TrueCrypt/TrueCrypt Website.url</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5 hash: 9cda3a91cf04824781bbebf415fe5211</td>
</tr>
</tbody>
</table>

Only one file (out of total of 455) was discovered due to the different versions of the anti-forensic tool present on the M57-Patents scenario compared to the version used to create the application profile (see Table 7.4). The solution presented in this research was able to detect a total of 393 file system entries, while hash set analysis was only able to detect 8 file system entries. Nevertheless, no false positives were detected from hash analysis and the only result was the 8 occurrences of the same data file across 8 disk images. The one data file that was detectable was because the file content did not change between tool versions, resulting in the same hash value. As indicated by the results, this is not a common occurrence (based on the tested applications) and will only happen when a file is not modified between versions.

The findings from performing a review comparing detection capabilities using hash analysis clearly demonstrates the effectiveness of the system design implemented in this research. The ability to detect a different application software version than that present on the target data set is evident in the comparative results. Hash analysis fails to detect sufficient evidence. This is important, as creating an application profile (or hash set) for every software version is not feasible. The next stage in evaluating the effectiveness of the matching component of the system design is Windows Registry matching.

### 7.2.1.3 Registry Matching Effectiveness

Matching Windows Registry entries incorporates the correlation of Registry keys and values between the application profile(s) and the target data set. Table 7.9 displays the results from Registry matching for target disk images from the M57-Patents scenario. The disk image name is provided with the associated application profile name and version number. The four digital artifacts classifiers are included \( (tp, tn, fp, fn) \) as well as the four effectiveness metrics \( (P, R, F_1, A) \).

The overall results from Registry matching indicate an effective system design to automate detection of Windows Registry entries directly relevant to the selected anti-forensic tools. The same 13 *interesting* disk images used for file system matching were identified to have Registry entries from one of the three anti-forensic tool profiles. The results were very similar to those achieved for file system matching. However, the average recall scores were lower: 0.72 for Registry matching compared to 0.80 for file system matching. Once again, recall scores were lowered by the inability to detect entries from Eraser, even though they were present on three disk images from the user Terry. This was caused by the same problem faced in file system
Table 7.9: Effectiveness results for Registry matching for the M57-Patents Scenario

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>P</th>
<th>R</th>
<th>F1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-03</td>
<td>TrueCrypt</td>
<td>34</td>
<td>290268</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.92</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>TrueCrypt</td>
<td>49</td>
<td>290523</td>
<td>0</td>
<td>5</td>
<td>1.00</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>TrueCrypt</td>
<td>49</td>
<td>290718</td>
<td>0</td>
<td>5</td>
<td>1.00</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-08</td>
<td>TrueCrypt</td>
<td>49</td>
<td>290732</td>
<td>0</td>
<td>5</td>
<td>1.00</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>TrueCrypt</td>
<td>49</td>
<td>291461</td>
<td>0</td>
<td>5</td>
<td>1.00</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>TrueCrypt</td>
<td>49</td>
<td>291555</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.89</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>TrueCrypt</td>
<td>49</td>
<td>291555</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.89</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>TrueCrypt</td>
<td>49</td>
<td>291702</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.89</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-08</td>
<td>CCleaner</td>
<td>33</td>
<td>206943</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>CCleaner</td>
<td>33</td>
<td>207279</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>CCleaner</td>
<td>33</td>
<td>207595</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>CCleaner</td>
<td>33</td>
<td>207608</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>CCleaner</td>
<td>33</td>
<td>208429</td>
<td>0</td>
<td>6</td>
<td>1.00</td>
<td>0.85</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>Eraser</td>
<td>0</td>
<td>207499</td>
<td>0</td>
<td>135</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>Eraser</td>
<td>0</td>
<td>207512</td>
<td>0</td>
<td>135</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser</td>
<td>0</td>
<td>208333</td>
<td>0</td>
<td>135</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Average 1.00 0.72 0.76 1.00

matching, in that Eraser was rewritten and the changes resulted in completely different digital artifacts present in the application profile compared to the target data sets. Again, a perfect precision score (1.00) was achieved for all tested disk images. The overall average $F_1$-measure score was 0.76, representing a harmonic mean of precision and recall scores.

A limitation of the Vestigium tool was discovered when attempting to match UserAssist Registry values. Attempts were made to correlate UserAssist entries using path normalisation for the full path and basename of Registry entries (see Section 5.5.3.1). However, problems were discovered during M57-Patents testing due to discrepancies between the normalised path properties. The following listing illustrates discrepancies between normalised Registry entries from Windows XP and Windows 7. The first comparison displays differences in the UserAssist key between Windows versions, while the second comparison displays differences in the basename (the actual Registry value name) between Windows versions.

Example of UserAssist key path differences:
Windows XP: ...\UserAssist\{75048700-EF1F-11D0-9888-006097DEACF9}\Count
Windows 7: ...\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count

Example of UserAssist value name differences:
Windows XP: HRZR_EHACNGU:P:\Hfref\VRHfre\Qrfxgbc\ppfrghc226.rkr
Windows 7: P:\Hfref\VRHfre\Qrfxgbc\ppfrghc226.rkr

As illustrated, the UserAssist key varies between Windows XP and Windows 7; this is a hard-
coded Globally Unique Identifier (GUID) value. Additionally, a Windows XP UserAssist value name (basename) has a hard-coded prefix (HRZR_EHACNGU:), while a Windows 7 UserAssist value name has no prefix. Both of these inconsistencies have been discovered in previous research (Stevens, n.d.), but were not known to the author at the time of system design and software development. Modifications can be made to the path normalisation implementation (CellPathNormaliser.py) to resolve these problems and normalise the key path and value names to allow correlation between Windows versions. A summary of findings for Windows Registry matching for each tool is outlined in the following paragraphs.

**CCleaner:** The overall results for CCleaner showed a high recall score and perfect precision score (0.85 and 1.00 respectively). This means that 85% of all Registry entries present in the 5 disk images where CCleaner was installed were correctly detected. However, a total of six Registry entries (out of 39) were not detected correctly. One was a UserAssist Registry value (as discussed above), while five other Registry entries were not discovered (not in the application profile), as displayed in the following listing:

```
SOFTWARE\CCleaner
SOFTWARE\CCleaner\UpdateCheck
SOFTWARE\CCleaner\(Default)
SOFTWARE\Classes\CLSID\{645FF040-5081-101B-9F08-00AA002F954E}\Shell
SOFTWARE\Software\Piriform\CCleaner\UpdateKey
```

Further investigation of the missing Registry entries concluded that the newer version of CCleaner (version 5.09) did not create these entries when installing and were unique to older versions of the application on the target data set (version 2.26). Therefore, this problem was not caused by the Windows operating system version difference (Vista vs. 7), rather were due to tool version differences and changes that have been made to the CCleaner tool.

**Eraser:** Similar to file system matching, the results for Eraser did not align with the other anti-forensic tool results. A total 135 Registry entries were determined to be ground truth for the Eraser tool, but none were detected during testing. It resulted in an average recall score of 0.00 for all 3 disk images from the user Terry. As previously stated, this was caused by a complete re-write of Eraser between versions 5.7.8 (present on the data set) and 6.2.0 (used for creating the application profile). An additional test was again conducted using the new profile created for additional file system matching testing using Eraser version 5.7.8. Table 7.10 displays the results from the newly created application profile.

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>P</th>
<th>R</th>
<th>F1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>terry-2009-12-10</td>
<td>Eraser-5.7.8</td>
<td>132</td>
<td>117917</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>Eraser-5.7.8</td>
<td>132</td>
<td>118051</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser-5.7.8</td>
<td>132</td>
<td>117917</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>132</td>
<td>117917</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 7.10: Overview of Windows Registry matching results for Eraser version 5.7.8

216
The results from the newly authored Eraser profile proved to be much more effective, with a high recall score (0.98) and perfect precision score (1.00). A total of only three Registry entries were not discovered (out of 135), all UserAssist Registry values. Each UserAssist entry was not able to be detected due to the discrepancies between the versions of Windows for the data set and application profile (as previously discussed).

**TrueCrypt:** The overall results from TrueCrypt showed a high recall and perfect precision score (0.90 and 1.00 respectively). A total of six Registry entries were not detected (this number varied based on the disk image date, see Table 7.9). Two were UserAssist entries, present due to the problem detecting UserAssist keys on different Windows versions. The other four were, in fact, only two Registry values that were twice present on the data set. Two Registry entries were created by TrueCrypt version 6.3a that are not present in version 7.1a. A duplicate of each entry was discovered as both entries are present in the SYSTEM Registry hive under different ControlSet keys, as shown in the listing below.

```
SYSTEM\ControlSet001\Services\truecrypt\Security
SYSTEM\ControlSet001\Services\truecrypt\Security\Security
SYSTEM\ControlSet002\Services\truecrypt\Security
SYSTEM\ControlSet002\Services\truecrypt\Security\Security
```

The results of Windows Registry matching indicates an effective system implementation to automate the detection of Registry entries using the authored application profiles. Again, the system was effective when correlating Registry entries between different operating system versions and anti-forensic tool versions. The very low score of the Eraser tool was attributed to the tool being completely re-written between the tool version of the data sets and the tool version used to create the application profile. Nevertheless, an average recall score of 0.72, or 72% of all relevant Registry entries was still discovered. If Eraser results are not included, the recall score is increased to 0.88, or 88% of all possible Registry entries.

### 7.2.1.4 Registry Matching Effectiveness Comparison: Searching

There are no similar solutions available as a comparison for the detection of Registry entries. The Vestigium tool extracts, processes and searches multiple hive files using previously established known Registry entries (from the application profile). This is a reference set solution. Existing forensic tools for Registry analysis do not perform such an examination, rather they provide an interface for the user to carry out manual investigation by locating known entries of interest; for example, an analyst can investigate values under the UserAssist key to determine recently run applications or analyse the Users key to determine the last user logon time. Normally, these keys are manually analysed using a Registry tool that have a similar interface provided by the Microsoft Windows `regedit` utility. In contrast, this research uses an automated metadata correlation searching function. Nevertheless, most of the currently available Registry analysis tools do have a search function which provides the ability to query Registry content using user-defined keywords.

Since the forensic analysis approach used in this research is fundamentally different to
previous approaches it is difficult to perform a direct comparison to other solutions. However, to provide a baseline to establish effectiveness, a comparison was conducted between the results from this research solution and the use of keyword searching to identify relevant content. To achieve this, the reglookup tool was used to parse Registry hive files and the output searched using the grep tool. The keywords specified are case insensitive names of each anti-forensic tool selected for testing, as follows:

- **CCleaner**: ccleaner
- **Eraser**: eraser
- **TrueCrypt**: truecrypt

The Registry hive files were sourced from the output from Vestigium which had previously extracted all hives from the 13 interesting target disk images. The following method provides an example of the processing technique implemented. The reglookup tool is used to parse each Registry hive file and produce a Comma Separated Value (CSV) file format written to standard output. This output is redirected (using the pipe character), processed by grep and keyword matching is performed. In the following example, the keyword truecrypt is searched for and matches appended to a text file for later manual analysis.

```
$ reglookup m57_allhives\jo-2009-12-11-002\WINDOWS-system32-config-software ^
| grep -i -n truecrypt >> jo-2009-12-11-002_TC_matches.txt
```

The prescribed method was executed against the selected disk images. The output was manually analysed and compared to the previously established ground truth information. Table 7.11 displays the results for the four prescribed information retrieval metrics.

The overall results for Registry keyword searching proved effective at detecting Registry entries of forensic interest. The average recall score was 0.81, or 81% of all present Registry entries that were correctly detected. A perfect precision score was achieved (1.00), meaning no false positive results were detected and the F$_1$-measure score was 0.87. In general, the effectiveness metric results were high. The F$_1$-measure score was higher for Registry keyword searching (0.87) compared to the solution presented in this research (0.76). The lower score for the results from this research was caused by the anomaly of the low recall scores for the Eraser tool. This result reflects that differences in tool versions did not effect Registry keyword searching results. When including the results from the revised Eraser profile (version 5.7.8), the average F$_1$-measure score for this research then increased to 0.95, higher than that achieved for the Registry keyword searching method (0.87).

It is proposed that the results from Registry keyword searching were high due to the uniqueness of the keywords used. All three keywords (truecrypt, ccleaner and eraser) can be considered unique; that is, it is unlikely that the keyword would appear on a default Windows operating system installation or with Registry entries created by other application software that may be installed on the system. This is a scenario that other application software may not share; for example on a default Windows operating system install, a number of Registry entries would most likely contain the microsoft office or simply office keywords, even when
### Table 7.11: Overview of Registry keyword search analysis on the M57-Patents scenario

<table>
<thead>
<tr>
<th>Name</th>
<th>Keyword</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>P</th>
<th>R</th>
<th>F1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-03</td>
<td>truecrypt</td>
<td>35</td>
<td>290302</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.97</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>truecrypt</td>
<td>54</td>
<td>290572</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>truecrypt</td>
<td>54</td>
<td>290767</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-08</td>
<td>truecrypt</td>
<td>54</td>
<td>290781</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>truecrypt</td>
<td>54</td>
<td>291510</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>truecrypt</td>
<td>54</td>
<td>291604</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>truecrypt</td>
<td>54</td>
<td>291604</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>truecrypt</td>
<td>54</td>
<td>291751</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-08</td>
<td>ccleaner</td>
<td>30</td>
<td>206976</td>
<td>0</td>
<td>9</td>
<td>1.00</td>
<td>0.77</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>ccleaner</td>
<td>30</td>
<td>207312</td>
<td>0</td>
<td>9</td>
<td>1.00</td>
<td>0.77</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>ccleaner</td>
<td>30</td>
<td>207628</td>
<td>0</td>
<td>9</td>
<td>1.00</td>
<td>0.77</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>ccleaner</td>
<td>30</td>
<td>207641</td>
<td>0</td>
<td>9</td>
<td>1.00</td>
<td>0.77</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>ccleaner</td>
<td>30</td>
<td>208462</td>
<td>0</td>
<td>9</td>
<td>1.00</td>
<td>0.77</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>eraser</td>
<td>54</td>
<td>207499</td>
<td>0</td>
<td>81</td>
<td>1.00</td>
<td>0.40</td>
<td>0.57</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>eraser</td>
<td>54</td>
<td>207512</td>
<td>0</td>
<td>81</td>
<td>1.00</td>
<td>0.40</td>
<td>0.57</td>
<td>1.00</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>eraser</td>
<td>54</td>
<td>208333</td>
<td>0</td>
<td>81</td>
<td>1.00</td>
<td>0.40</td>
<td>0.57</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>1.00</strong></td>
<td><strong>0.81</strong></td>
<td><strong>0.87</strong></td>
<td><strong>1.00</strong></td>
</tr>
</tbody>
</table>

The results indicate that Registry keyword searching does have limitations for detection of anti-forensic tools as well as other application software. Ultimately, the solution is not as

---

The application is not installed. A further test is needed to prove this premise.

The ground truth data set (GT-01.RAW) from known data set testing was used to search for a variety of anti-forensic tool names in an attempt to discover false positive Registry keyword search results. The ground truth disk image was used as it contains a default Microsoft Windows 7 operating system install, without any additional application software; meaning that if an application keyword is detected, it can be assured that it is a false positive. The same Registry keywords search technique was leveraged using the following keywords that represent either anti-forensic tools or other common application software names: `putty`, `sdelete`, `chrome`, `safari`, `office`, `microsoft office`, `perl` and `python`.

Five out of eight keywords were detected on the ground truth image using the prescribed software names, including: `putty`, `sdelete`, `office`, `microsoft office` and `perl`. The number of results for each keyword varied. The listing below highlights some interesting keyword search results, where the keyword is in bold face (all Registry paths have the root key removed).

```
Microsoft\Windows\CurrentVersion\App Management\Icon Hints\Microsoft Office
Microsoft\Multimedia\TV\Tuning Spaces\Antenna\Input\Type{871C380-42A0-1069-A2EA-08002B30309D}\ShellFolder\Hide\sDeletePerUser
CurrentVersion\Schedule\Configuration\Tasks\PerLeastPrivEngine
```

The results indicate that Registry keyword searching does have limitations for detection of anti-forensic tools as well as other application software. Ultimately, the solution is not as
robust as the system implementation in this research, because the system design herein detects Registry entries using multiple metadata properties, providing a more reliable and accurate correlation method.

7.2.2 False Positive Detection Rate using Vestigium

An effective system design and implementation has been established for detecting anti-forensic tool presence from known interesting disk images from the M57-Patents scenario; that is, disk images that are known to have at least one of the selected anti-forensic tools installed on them. All the results presented thus far show a perfect precision score (1.00), meaning that no false positive digital artifacts were erroneously detected. However, experimental testing was only conducted for 13 interesting disk images from the M57-Patents scenario, while there are a total of 79 disk images available in the data set. The next stage in experimental testing using the M57-Patents scenario is to again execute the Vestigium tool, this time against the remaining 66 disk images available to determine if any false positive digital artifacts are detected. Since ground truth has already been established (see Section 7.1.3), any digital artifact detected on the non interesting disk images can be classified as a false positive result.

Table 7.12 displays the results from file system matching using Vestigium against the remaining 66 non-interesting disk images. The same four digital artifact classifiers are provided (tp, tn, fp, fn). However, effectiveness metrics are not included as true positive and false negative detection are absent, both of which are required for the specified metrics. Due to space restrictions, the name column provides a range of disk images where false positives were detected; for example, terry-2009-11-19 to terry-2009-12-11-002 represents a total of 15 disk images from the user Terry during that date range.

Table 7.12: Cumulated false positive results for file system matching using the non-interesting M57-Patents scenario disk images

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>terry-2009-11-19 to terry-2009-12-11-002</td>
<td>CCleaner</td>
<td>0</td>
<td>2,188,432</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

Testing file system matching against the remaining non-interesting disk image resulted in detection of a total of 30 false positive matches, all from the user Terry. These matches stemmed from two false positive data files which were detected on a total of 15 different disk images, ranging from 2009-11-19 to 2009-12-11. Both false positive files were from the CCleaner application profile. An example of the file system path for both erroneously detected data files are displayed below:

Windows/Prefetch/AU-.EXE-746367D6.pf
Users/terry/AppData/Local/Temp/~nsu.tmp

Further investigation regarding the detected false positives was performed and discovered that both data files could be attributed to the Nullsoft Install System (NSIS), an
open source system to create application installers for Microsoft Windows systems\textsuperscript{12} Further analysis revealed that CCleaner used the NSIS tool to build the tool installer and was therefore included in the application profile. However, numerous other applications also use the NSIS tool to package installer files and was, therefore, discovered on additional systems from the M57-Patents scenario. Manual analysis could not determine the exact application which created the two false positive data files on Terry’s system. However, a collection of well known applications use the NSIS system including Mozilla for Firefox, Thunderbird and FileZilla, as well as the Open Office suite and Flickr Uploader application\textsuperscript{13}.

Analysis of the non-interesting disk images was also examined in terms of Registry artifact matching. Table \ref{tab: registy-matching} displays the results from Registry matching using Vestigium against the remaining 66 non-interesting disk images. Again, the same digital artifact classifiers are provided, effectiveness metrics not included and the name column provides a range of disk images where false positives were detected.

Table 7.13: Cumulated false positive results for Registry matching using the non-interesting M57-Patents scenario disk images

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>charlie-2009-11-12 to</td>
<td>CCleaner</td>
<td>0</td>
<td>5,535,932</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>charlie-2009-12-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jo-2009-11-12 to</td>
<td>CCleaner</td>
<td>0</td>
<td>1,989,885</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>jo-2009-12-20-oldComputer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-12 to</td>
<td>CCleaner</td>
<td>0</td>
<td>5,485,186</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>pat-2009-12-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>0</td>
<td>13,011,003</td>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

As witnessed in the false positive results from file system matching, the erroneously detected Registry entries were, again, from the CCleaner tool. No other false positives were detected from either Eraser or TrueCrypt. The listing below displays the two Registry keys that were detected on a total of 45 different disk images from the M57-Patents scenario.

| SOFTWARE\Google\No Toolbar Offer Until |

Investigation revealed that the two Registry keys are related to the Google Chrome web browser which is bundled with the CCleaner tool. During installation of CCleaner, the user is presented with the option to also install the Chrome browser. During application profile creation, the input box for installing Chrome was unselected and, consequently, resulted in the two Registry keys being created. The reasoning for the Registry key appearing in systems without CCleaner installed is that the entry is not specific to the tool. Other applications also use the same Registry keys to store information if Chrome is bundled with the installer.

\textsuperscript{12}See: \url{http://nsis.sourceforge.net/Main_Page}

\textsuperscript{13}See: \url{http://www.scratchpaper.com/home/shameless-promotion}
Overall, a low number of false positive results were detected when testing included all 79 disk images from the M57-Patents scenario. Of the interesting disk image no false positives were detected during experimental testing. However, four unique digital artifacts were discovered when testing the remaining 66 non-interesting disk images. Two data files and two Registry keys were found, both from the CCleaner tool. It should be noted that the resultant false positive digital artifacts are not an issue with Vestigium, rather they are digital artifacts included in the application profile which are not unique to the anti-forensic tool.

All four entries were manually removed from the CCleaner profile. However, this issue has a simple automated solution. The LiveDiff static blacklist was updated to include all four discovered false positive results so that any additionally created profiles would not include these entries. Nevertheless, this result provides further evidence of potential difficulties in identification of unique and relevant digital artifacts during application profile creation. Such issues, however, can be solved and overcome by the robust design and functionality of the LiveDiff tool.

7.2.3 Detecting Residual Digital Artifacts using Vestigium

The results presented thus far have evaluated the functionality to correlate digital artifact metadata to enable automated detection of anti-forensic tools on a target data set using application profiles. However, a limitation of the experimental testing presented in Section 7.2.1 is centred around the lack of residual digital artifacts; for example, file system or Registry entries that are associated with the uninstallation phase of the application life cycle. This is because each anti-forensic tool was found to be installed (specifically: installed, opened and closed) during the course of the M57-Patents scenario, but none were uninstalled. This means no results were found to realise the capability to detect deleted, or residual, digital artifacts. This is important to evaluate as a suspect may have carried out a malicious action with a tool, then uninstalled the tool in an attempt to remove the evidence of this activity. Therefore, additional testing is required to investigate this context.

The primary requirement for additional testing is, therefore, to select and perform detection of an anti-forensic tool (or another application software type) known to be installed and then removed from a system (uninstalled) at some point during the scenario. In previous research, Jones et al. (2015) investigated past activity of application software from partial digital artifacts using the M57-Patents scenario as one of the testing data sets. More specifically, the authors performed block hashing of data files uniquely associated with an application and attempted to detect the same blocks in the disk images from the M57-Patents scenario. An experiment outlined in the research investigated the presence and persistence after uninstallation of an anti-forensic tool named XP Advanced Keylogger, a keylogger tool produced by XP Tools. The results presented by Jones et al. (2015) illustrated that XP Advanced Keylogger had been installed by the user Pat on 2009-12-03 and subsequently uninstalled on 2009-12-07. After uninstallation, there were still four days remaining in the scenario and a disk image was captured for each of these days. This provides an excellent experimental testing scenario to determine the effectiveness of the solution implemented in this research to
detect deleted digital artifacts using metadata correlation.

The creation of a new application profile for the XP Advanced Keylogger tool now follows with a brief overview of the testing methodology to be used. Testing will be performed using the new application profile against selected disk images of the user Pat from the M57-Patents scenario.

7.2.3.1 Application Profile Creation: XP Advanced Keylogger

Detecting residual and deleted artifacts using Vestigium on selected disk images of the M57-Patents scenario required a new application profile to be authored for the XP Advanced Keylogger tool. The tool installer was sourced with the following details:

- Tool name: XP Advanced Keylogger
- Tool version: 2.1
- Installer file name: xpadvancedkeylogger.exe
- Installer MD5 hash value: f53bafdf255f5505f5350fd91b97e0664

For reasons of simplicity, the remainder of this section uses the term XP Keylogger to refer to the XP Advanced Keylogger tool. The finalised application profile creation method was implemented (see Section 6.5.1 and Section 6.5.3). To reiterate, this involved creating five application profiles with dynamic and static blacklisting enabled, performing application profile intersection using the five profiles as input and finally a manual review to assess profile contents. Table 7.14 displays a summary of the application profile created for XP Keylogger including counts for each digital artifact type, the time (in seconds) to create all five profiles used for intersection and the final file size of the application profile in kilobytes (KB).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>Time</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>XP Keylogger</td>
<td>9</td>
<td>54</td>
<td>574</td>
<td>687</td>
<td>1,321</td>
<td>472</td>
<td>1,530 KB</td>
</tr>
</tbody>
</table>

The output from application profile creation was a new APXML document to represent the digital artifacts uniquely associated with the XP Keylogger tool. The efficiency of the application profile creation method using LiveDiff was evident when generating this new profile, taking 472 seconds, or just under 8 minutes, to create all five profiles. The profiles were ingested to the APXMLIntersection.py script which resulted in a finalised application profile ready for testing. The real-world elapsed time was measured which resulted in a total time of 14 minutes to create a finalised application profile for the XP Keylogger tool. This includes starting virtual machines, performing data collection using LiveDiff and performing post-processing tasks including profile intersection. Full application profile metadata contents are provided in Appendix C.7.
7.2.3.2 Testing Methodology

The experimental testing method used to investigate residual digital artifacts follows the same process as specified for all M57-Patent scenario testing (see Section 7.1.2). The Vestigium tool is used to correlate digital artifacts between the target data set and application profile(s). In this experiment, only the XP Keylogger profile was used as input. Following the information specified by Jones et al. (2015), the selected disk images from the M57-Patents scenario are a total of 19 disk images from the user Pat which have the Microsoft Windows XP operating system installed.

7.2.3.3 Residual Digital Artifact Detection Results

Table 7.15 presents an overview of the results from residual digital artifact detection on the 19 disk images from the user Pat from the M57-Patents scenario. Only disk images with one or more detected digital artifacts are displayed, a total of seven disk images. The disk image name is provided with the accompanying application profile name and the four digital artifacts classifiers are included ($tp$, $tn$, $fp$, $fn$) as well as the four effectiveness metrics ($P$, $R$, $F_1$, $A$). To reiterate, previous research states that XP Keylogger was installed on 2009-12-03 and uninstalled before the disk image was collected on 2009-12-07 (Jones et al., 2015).

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>$tp$</th>
<th>$tn$</th>
<th>$fp$</th>
<th>$fn$</th>
<th>$P$</th>
<th>$R$</th>
<th>$F_1$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat-2009-12-03</td>
<td>XPKeylogger-2.1</td>
<td>27</td>
<td>40296</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.90</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-04</td>
<td>XPKeylogger-2.1</td>
<td>27</td>
<td>40279</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>0.90</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-07</td>
<td>XPKeylogger-2.1</td>
<td>9</td>
<td>44364</td>
<td>0</td>
<td>25</td>
<td>1.00</td>
<td>0.26</td>
<td>0.41</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-08</td>
<td>XPKeylogger-2.1</td>
<td>9</td>
<td>44371</td>
<td>0</td>
<td>25</td>
<td>1.00</td>
<td>0.26</td>
<td>0.41</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-09</td>
<td>XPKeylogger-2.1</td>
<td>7</td>
<td>44441</td>
<td>0</td>
<td>27</td>
<td>1.00</td>
<td>0.21</td>
<td>0.35</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-10</td>
<td>XPKeylogger-2.1</td>
<td>7</td>
<td>44398</td>
<td>0</td>
<td>27</td>
<td>1.00</td>
<td>0.21</td>
<td>0.35</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-11</td>
<td>XPKeylogger-2.1</td>
<td>7</td>
<td>44402</td>
<td>0</td>
<td>27</td>
<td>1.00</td>
<td>0.21</td>
<td>0.35</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>1.00</strong></td>
<td><strong>0.42</strong></td>
<td><strong>0.54</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td>pat-2009-12-03</td>
<td>XPKeylogger-2.1</td>
<td>625</td>
<td>286133</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-04</td>
<td>XPKeylogger-2.1</td>
<td>626</td>
<td>285795</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-07</td>
<td>XPKeylogger-2.1</td>
<td>383</td>
<td>287322</td>
<td>0</td>
<td>283</td>
<td>1.00</td>
<td>0.58</td>
<td>0.73</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-08</td>
<td>XPKeylogger-2.1</td>
<td>383</td>
<td>287299</td>
<td>0</td>
<td>283</td>
<td>1.00</td>
<td>0.58</td>
<td>0.73</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-09</td>
<td>XPKeylogger-2.1</td>
<td>32</td>
<td>287809</td>
<td>0</td>
<td>634</td>
<td>1.00</td>
<td>0.05</td>
<td>0.10</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-10</td>
<td>XPKeylogger-2.1</td>
<td>32</td>
<td>287994</td>
<td>0</td>
<td>634</td>
<td>1.00</td>
<td>0.05</td>
<td>0.10</td>
<td>1.00</td>
</tr>
<tr>
<td>pat-2009-12-11</td>
<td>XPKeylogger-2.1</td>
<td>32</td>
<td>288088</td>
<td>0</td>
<td>634</td>
<td>1.00</td>
<td>0.05</td>
<td>0.10</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>1.00</strong></td>
<td><strong>0.47</strong></td>
<td><strong>0.54</strong></td>
<td><strong>1.00</strong></td>
</tr>
</tbody>
</table>

The results from residual digital artifact detection provide significant insight to the effectiveness of the implemented system to detect deleted and residual digital artifacts from application
software using a metadata correlation approach. The findings confirm that XP Keylogger was installed by the user Pat on 2009-12-03 and subsequently uninstalled on 2009-12-07. These dates correspond with the results reported by Jones et al. (2015). The following discussion details the results of the longitudinal analysis of Pat’s disk image starting with the installation of XP Keylogger until the end of the M57-Patents scenario.

**pat-2009-12-03:** The XP Keylogger tool was installed. A total of 27 file system entries and 625 Registry entries were detected. Vestigium returned matches for install, open and close application life cycle phases. Therefore, the tool had been installed and executed by the user Pat. Registry entries were discovered in Pat’s NTUSER.DAT hive file to support that Pat (or someone using Pat’s user account) executed the tool.

**pat-2009-12-04:** XP Keylogger is still present on Pat’s system. A MUICache Registry entry was detected indicating the tool was executed again on this date.

**pat-2009-12-07:** The XP Keylogger tool was uninstalled. A total of 9 file system entries still remained on the system. However, only one was found to be in an unallocated state, meaning that almost all metadata linking the deleted file system entries had already been overwritten by new data. This means that 8 of the file system entries were residual (not removed during uninstallation and still in an allocated state in the file system). A Windows Prefetch file (TOOLKEYLOGGER.EXE-2986F3DF.pf) was still present. The tool licensing component also still existed in the Windows system32 directory. A total of 383 Registry were detected. Of these, 351 were found in an unallocated state (deleted) which demonstrates the functionality of the system (CellXML-Registry) to detect deleted Registry artifacts. An additional 32 residual Registry entries were detected, 31 allocated entries in the SOFTWARE hive file and one MUICache entry in Pat’s NTUSER.DAT hive file. These 32 residual entries were not removed by the tool uninstaller.

**pat-2009-12-08:** One day after uninstallation of the XP Keylogger tool. Results found that the same number of file system and Registry entries were present. This shows that residual digital artifacts persist in the file system and Registry a day after tool uninstallation, even with the user performing regular daily activities.

**pat-2009-12-09:** Two days after tool uninstallation. The Windows Prefetch file and one unallocated file ($OrphanFiles/Password.gif) no longer exist in the file system metadata. The tool licensing component is still present and still in an allocated state. It appears that the tool uninstaller does not remove these entries. A total of seven file system entries are still detected. All previously detected unallocated Registry entries (a total of 351) are no longer recoverable. The residual 32 Registry entries are still present on the system, again, the tool uninstaller appears to not remove these entries correctly. One UserAssist Registry entry still exists in the NTUSER.DAT hive file for the user Pat. This is remaining evidence that the tool was executed by the user.

**pat-2009-12-10 to pat-2009-12-11-002:** Three to five days after tool uninstallation. The same residual file system and Registry entries still exist that were recorded on
2009-12-09. These entries are still allocated and will remain on the system until manually deleted by a user.

The longitudinal analysis of the XP Keylogger tool on the disk images from the user Pat has resulted in useful and interesting findings. Vestigium was able to detect deleted (unallocated) and residual (remained after uninstallation) digital artifacts. Detection of unallocated Registry entries was only possible using the CellXML-Registry tool authored specifically for this research. However, the findings from the experiment indicate that the life span of unallocated Registry entries can be short, in this case two days of normal system activity. The life span of the file system metadata can be even shorter. Only one unallocated file system entry was detected using file system metadata, even though it is highly likely that residual fragments still exist. Results presented by Jones et al. (2015) support this premise, where some data file fragments were still able to be detected after uninstallation.

In this experiment it was also noted that the tool uninstaller failed to delete all file system and Registry entries associated with the tool. A collection of file system and Registry entries remained in an allocated state even after uninstallation for the remainder of the scenario, five days after tool uninstallation. These entries will remain until manually removed by the user, an event which may be unlikely to occur. Overall, the results obtained are potentially useful evidence in a digital investigation. The Vestigium tool provided evidential proof that XP Keylogger was installed and executed on a specific date, linked a specific user account with the tool (through the NTUSER.DAT hive file), was able to detect unallocated entries indicating the tool was uninstalled on a specific date and was able to detect residual digital artifacts that remained five days after tool uninstallation. All of these findings are extremely useful in practice when determining application software usage in a digital investigation.

Comparison to a similar study, that by Jones et al. (2015), is next conducted as it is advantageous to determine the effectiveness of the implemented system design when compared to other forensic analysis methods.

7.2.3.4 Comparison of Findings to Jones et al. (2015)

To establish the relative effectiveness of the implemented system design a comparison should be made to similar approaches to automate detection of digital artifacts from application software. A comparison of findings to perform residual digital artifact detection is therefore conducted between this research solution (which uses a metadata matching approach) and the research conducted by Jones et al. (2015) (which uses a binary block-based hashing matching approach).

Figure 7.1 illustrates the results obtained in this research when compared to a similar research solution presented by Jones et al. (2015). Each disk image from the user Pat is shown with the corresponding percentage of the profile contents that were detected. The three different results are: 1) The percentage of the application profile entries detected for

---

14 The purpose of Figure 7.1 is to display the overall trend resulting from testing. Results from this research are available in Table 7.15. Also, see Jones et al. (2015) for additional details. 

226
this research (blue); 2) The percentage of detected blocks from Jones et al. (2015) (orange); and 3) The percentage of detected files from Jones et al. (2015) (green). All results are presented based on the percentage of the application profile (or reference set) detected for all experimental tests; that is, the proportion of the entries in the profile which were detected on the target data set. The results are discussed in the following paragraphs.

![Figure 7.1: Longitudinal detection capabilities for XP Keylogger on the M57-Patents scenario for the user Pat. Results are presented for the Vestigium tool and compared against findings from Jones et al. (2015) including sector and file percentages.](image)

According to Jones et al. (2015), the created profile for XP Keylogger contained a total of 23 data files comprised of 4,716 blocks, each 512 bytes in length. Thus, the profile has 4,716 block hashes generated using the MD5 hashing algorithm. As displayed, the XP Keylogger tool was detected on pat-2009-12-03 with approximately 83% matching blocks, and 67% of the files in the profile.

The results of the solution used in this current research discovered approximately 99% of all profile contents including 27 file system entries (directories and data files) and 625 Registry entries (keys and values). This is a total of 652 digital artifacts. Compared to only 23 data files of the Jones et al. (2015) project, the number detected in the solution for this research is a much higher number of unique application software artifacts. The research and findings from Jones et al. (2015) reported similar results for the next day (2009-12-04) as those observed for 2009-12-03.

All results correspond with the XP Keylogger tool being uninstalled on 2009-12-07. It resulted in a decrease in detected digital artifacts for this research, as well as a decrease in detected blocks and file percentage for the similar solution. This is expected, as data was deleted and subsequently overwritten by new data making 100% recovery highly unlikely. However, both studies continued to report detection of residual evidence. Regarding the
pat-2009-12-07 disk image, this research solution detected 56% of the profile contents with most digital artifacts discovered in an unallocated (or deleted) state. The results from Jones et al. (2015) show a similar decrease in detection rates, approximately 20% of sectors were recovered, and 39% of files were detected.

Results from the next disk image (pat-2009-12-08) illustrates a continual decrease in detected digital artifacts. However, on the following day in the scenario (pat-2009-12-09) a dramatic decrease was found in this research solution (dropping to 5.6% detection), and sector recovery for Jones et al. (2015) dropping to approximately 0.5%. Interestingly, the file percentage detected for Jones et al. (2015) maintained an approximate 35% of detected files for the remainder of the disk images from the scenario. However, it is also evident that the percentage of detected files is skewed by false positive prevalence. Between the dates of pat-2009-11-12 to pat-2009-12-02 (a total of 11 disk images) the file percentage detected was approximately 4-5%. This is seen in Figure 7.1 between the date 2009-11-12 and 2009-12-01. As the XP Keylogger was not present on the system between these dates, it can be concluded that these are false positive results, thus, incorrectly increasing file detection rates by around 4-5% for all dates. Unfortunately, the raw data is unavailable to either prove, or disprove, this premise.

A further analysis was conducted on the final disk image from the user Pat from the M57-Patents scenario (disk image name: pat-2009-12-11). The selected disk image was collected five days after XP Keylogger was uninstalled. For this disk image Jones et al. (2015) detected 8 out of the 23 data files in their profile. However, out of the total 4,716 block hashes only 24 were discovered (a sector detection rate of 0.51%). Furthermore, the file detection percentage was 34.78% which means at least one sector was discovered from a total of 8 out of 24 files. Given a false positive file detection rate of 4–5% on dates without XP Keylogger present, this could mean a lower detection rate of approximately 30% or approximately 20 sector hashes. In comparison, the solution presented in this research detected 7 data files as compared to 8. Furthermore, a total of 32 residual Registry artifacts were also detected on the last disk image from the user Pat. Given the nil false positive rate of the solution implemented in this research, it can be concluded that each detected digital artifact (total of 37) is a true positive and, ultimately, a correctly recovered residual digital artifact.

Overall, both solutions were able to detect the presence of the XP Keylogger tool, before and after uninstallation. The solution presented in this research yielded a much higher count of relevant digital artifacts as it also incorporates four digital artifacts type (directories, data files, Registry keys and values) in comparison to Jones et al. (2015) which only includes data files. Block hashing can only be used for data files and does not encompass other digital artifact types.

An in-depth longitudinal analysis of the detection of anti-forensic tool persistence after uninstallation together with a comparison to a similar solution has hereby been provided. Findings that are relevant to the effectiveness of the path normalisation technique implemented in this research to aid digital artifact metadata matching now follows.
7.2.4 Effectiveness of Path Normalisation Technique

The system design proposed the use of path normalisation to transform the full path, or absolute path, of file system and Windows Registry entries into a generalised value (see Section 5.5.3.1). The goal of path normalisation was to improve digital artifact matching effectiveness in complex matching scenarios; for example, detecting file system entries with a variable *username* or detecting Windows Registry entries that have variations between Microsoft Windows versions. Results have suggested that path normalisation has proved effective at aiding detection of digital artifacts on a different Windows operating system version than that of the created application profile (see Section 7.2.1). However, a targeted and thorough analysis of the designed and implemented path normalisation technique is required to determine overall effectiveness.

Thus far, path normalisation proved effective at detecting digital artifacts with a file system path that had slight variations between the application profile and target data set; for example, the following listing displays the full path of a file system directory from the target data set (*jo-2009-12-03*), application profile (*TrueCrypt*) and the normalised equivalent of both file system paths.

<table>
<thead>
<tr>
<th>Target Data Set:</th>
<th>C:\Documents and Settings\Jo\Application Data\TrueCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Profile:</td>
<td>C:\Users\forensic\AppData\Roaming\TrueCrypt</td>
</tr>
<tr>
<td>Normalised Path:</td>
<td>%APPDATA%/TrueCrypt</td>
</tr>
</tbody>
</table>

In the above example, a file system path match was only achieved by the implementation of path normalisation. In this example, path normalisation removed the variable *username* value (*Jo* vs. *forensic*) and transformed the user profile path difference between Windows XP (C:\Documents and Settings) and Windows 7 (C:\Users). The final result was a transformed path value (%APPDATA%/TrueCrypt).

In-depth analysis of path normalisation to determine the effectiveness of the implemented solution for both file system and Windows Registry entries will be carried out using two selected disk images from the M57-Patents scenario: 1) *jo-2009-12-11-002*; and 2) *terry-2009-12-11-002*. The two disk images chosen for testing have each of the three selected anti-forensic tools installed and provide the ability to test digital artifact detection using the original actual path from the application profile and the normalised path from the target data set. The code was derived and modified from the Vestigium tool to perform this testing scenario. Two Python scripts were authored for additional testing: 1) *FSPPathDetection.py* to detect file system entries using the filename and filename_norm metadata properties; and 2) *REGPathDetection.py* to detect Registry entries using the cellpath, cellpath with the root key removed, and cellpath_norm metadata properties. Both scripts are available in Appendix D.2.

7.2.4.1 Path Normalisation Results for File System Entries

The findings from testing path normalisation effectiveness for file system entries discovered variable results for each anti-forensic tool. However, in all instances an increased number of
detected file system paths was observed.

Table 7.16 displays the comparison of results for path normalisation effectiveness for the two selected disk images from the M57-Patents scenario. The disk image name and associated anti-forensic tool profile are displayed including the number of actual (non-normalised) file system paths detected, the number of normalised file system paths detected and the percentage increase of path detection when normalisation is implemented.

Table 7.16: Overview of file system path detection rates using actual versus normalised path for selected M57-Patents scenario disk images

<table>
<thead>
<tr>
<th>Name</th>
<th>Tool</th>
<th>Actual</th>
<th>Normalised</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-11-002</td>
<td>TrueCrypt-7.1a</td>
<td>8</td>
<td>17</td>
<td>112.50</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>CCleaner-5.09</td>
<td>46</td>
<td>51</td>
<td>10.87</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser-5.7.8</td>
<td>14</td>
<td>17</td>
<td>21.43</td>
</tr>
</tbody>
</table>

All tested anti-forensic tools showed an increase in detection rates using file path normalisation. However, the results were variable for each tool. Analysis of all file system entries from each application profile revealed that the variable results were caused by the differences in anti-forensic tool operation; that is, the file system entries created.

Some tools created more file paths (a total of 9) which had variable user content; for example, TrueCrypt created a high number of file system paths with file paths that had an embedded variable username value (e.g., C:\Users\forensic\AppData\Roaming\TrueCrypt, where forensic is the variable username content). Therefore, for TrueCrypt, a higher number of file system paths were detected with path normalisation when compared to detection using the actual file path. In contrast, CCleaner created very few (a total of 5) file system entries with variable user content. Similar trends were observed for Eraser, which only had 3 file system entries with variable user content.

The differences showed varying results for each tool, with a percentage increase of detection using path normalisation that ranged from approximately 10% for CCleaner to 112% for TrueCrypt. In addition to variable user content, path normalisation also detected one Windows Prefetch file for each anti-forensic tool by removing the variable file name suffix. Listing 7.2 displays the additional file system paths that were only detectable using path normalisation (duplicates have been removed and all paths are displayed after path normalisation was performed on the actual path value).

The results from path normalisation for file system entries indicate an effective design and implementation. This was observed in the increase of the number of detectable file system paths that would not have been matched without path normalisation. One interesting scenario, not witnessed in public data set testing of path normalisation, was the introduction of variations to the Windows Program Files directory. On Microsoft Windows 64-bit systems, an additional Program Files (x86) directory is provided to store 32-bit applications running on a 64-bit system. Path normalisation included functionality to convert both
Program Files directory naming conventions to the normalised value of \%PROGRAMFILES\%. However, this scenario was not evident in the M57-Patents scenario as all systems were running 32-bit operating systems. Additional analysis was performed to investigate this situation and it was discovered that all three anti-forensic tools would install application information into the Program Files (x86) directory if installed on a 64-bit system. In this event, using the actual file system path for detection would not detect any file system entries because the entries that were detected without path normalisation were all located in the Program Files directory. This result highlights the success of file system path normalisation, a solution that provides the functionality to match file system paths with variable user content and variable path locations that differ between a Windows operating system version (e.g., Windows Vista versus Windows 7) and platforms (e.g., 32-bit versus 64-bit).

%ALLUSERSPROFILE%/Desktop/TrueCrypt.lnk
%STARTMENU%/Programs/TrueCrypt
%STARTMENU%/Programs/TrueCrypt/TrueCrypt Website.url
%STARTMENU%/Programs/TrueCrypt/TrueCrypt.lnk
%APPDATA%/TrueCrypt
%APPDATA%/TrueCrypt/Configuration.xml
%PREFETCH%/TRUECRYPT.EXE.pf
%SYSTEMROOT%/drivers/truecrypt.sys
%STARTMENU%/Programs/CCleaner
%STARTMENU%/Programs/CCleaner/CCleaner Homepage.url
%STARTMENU%/Programs/CCleaner/CCleaner.lnk
%PREFETCH%/CCLEANER.EXE.pf
%LOCALAPPDATA%/Eraser
%LOCALAPPDATA%/Eraser/default.ers
%PREFETCH%/ERASER.EXE.pf

Listing 7.4: List of additional file system paths detected using path normalisation

7.2.4.2 Path Normalisation Results for Registry Entries

Similar to the results from path normalisation for file system entries, the findings from testing path normalisation effectiveness for Registry entries discovered variable results for each anti-forensic tool. Again, for all tools it was observed that path normalisation resulted in an increased detection rate. This was due to two Registry path variables: 1) Windows Registry root key values which vary for different Windows operating system versions; and 2) Variables in Registry paths due to user-specific content as well as differences in Registry path between Windows versions. The following listing reiterates the differences in the root key for the SOFTWARE Registry hive file on different Windows operating system versions.

Windows XP: $$$PROTO.HIV
Windows Vista: CMI-CreateHive{29EE1162-53C9-4474-A2B6-D90A7F6B0A7C}
Windows 7: CMI-CreateHive{199DAFC2-6F16-4946-BF90-5A3FC3A60902}
Given the Registry hive root key differences, it is not possible to perform a comparison equality check for the same Registry entry on different operating system versions. Therefore, path normalisation effectiveness testing performed matched Registry path entries using three techniques:

- The actual path (`cellpath`) without normalisation
- The actual path (`cellpath`) with the root key removed
- A fully normalised path (`cellpath_norm`) as implemented in Vestigium

Table 7.17 displays an overview of results from Registry path normalisation. The disk image name and anti-forensic tool name are displayed. The count of detected actual paths, the actual paths with the root key removed and a fully normalised path are displayed. The percentage increase in detection rate between a path with the root key removed to a fully normalised path is also provided.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tool</th>
<th>Actual path</th>
<th>Root key removed</th>
<th>Normalised path</th>
<th>Percentage increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-11-002</td>
<td>TrueCrypt</td>
<td>0</td>
<td>19</td>
<td>49</td>
<td>157.89</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>CCleaner</td>
<td>0</td>
<td>28</td>
<td>34</td>
<td>21.43</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>Eraser</td>
<td>0</td>
<td>100</td>
<td>132</td>
<td>32.00</td>
</tr>
</tbody>
</table>

The results discovered that not a single Registry entry was able to be detected using the actual Registry path from the application profile. This is because the operating system version on the target data set (Windows XP and Vista) is different from the application profile (Windows 7). This means a different root key was present in every Registry path and ultimately resulted in no detected entries being made. When the root key was removed (i.e., stripping `$$PROTO.HIV` from the Registry path) a number of Registry entries were able to be detected. This is a similar technique to how `fiwalk` represents file system paths, by removing the Windows drive letter (e.g., `C:\`) from the file path. When comparing root key removal to fully normalised Registry paths, full path normalisation detected an average of 70% more entries. This demonstrates the effectiveness of path normalisation for detecting Registry entries between operating system versions. However, variable results were again reported for each anti-forensic tool with the percentage increase ranging from approximately 21% for CCleaner to 157% for TrueCrypt. This is similar to the results from file path normalisation caused by more user-specific content in the TrueCrypt application. Furthermore, TrueCrypt had a high number of ControlSet entries which were not detectable without normalisation. A total of 30 additional Registry entries were detected from the SYSTEM Registry hive under one of the ControlSet keys. All entries were unable to be detected without path normalisation as the application profile had a different ControlSet key name compared to the target data set, as seen in the following listing.
An interesting result from CCleaner was a collection of Registry entries from the shell Registry sub-key and the Registry values it stores. Six additional Registry paths were detected using full normalisation due to case inconsistencies between different operating system versions. The target data set (Windows Vista) represents shell Registry values in sentence case (Shell), while the application profile (Windows 7) represents the same sub-key in lower case (shell). A similar case inconsistency was discovered in the Eraser results for the ShellEx Registry key. Path normalisation solved both these inconsistencies by converting the Registry key path to lower case which is possible because Registry key paths are case insensitive (discussed in Section 5.5.3.1). The listing below displays examples of the two case inconsistency scenarios witnessed, and specify the path for the target data set, application profile and the resultant normalised path (some Registry entries have been truncated due to length).

Target Data Set: Classes\CLSID\{645FF0 ... 2F954E\}\Shell\Run CCleaner
Application Profile: Classes\CLSID\{645FF0 ... 2F954E\}\shell\Run CCleaner
Normalised Path: classes\clisd\{645FF0 ... 2F954E\}\shell\Run CCleaner

Target Data Set: Classes\Folder\shellex\ContextMenuHandlers\Erasext
Application Profile: Classes\Folder\ShellEx\ContextMenuHandlers\Erasext
Normalised Path: classes\folder\shellex\contextmenyhandlers\Erasext

7.2.4.3 Discussion of Path Normalisation Results

The overall results from path normalisation effectiveness has demonstrated and evaluated the capabilities of path normalisation to aid in automated detection of file system paths as well as Registry paths. The implementation of path normalisation was able to transform full path values to a generalised value and remove user-specific information, inconsistencies between operating versions and inconsistencies between operating system platform (system architecture).

In terms of file system entries, an average detection rate increase of 48% was achieved using path normalisation. The results for TrueCrypt were generally higher as the tool stores more application configuration data in user directories, all of which require normalisation to remove user-specific information. However, all tested anti-forensic tools benefited from performing path normalisation. Additionally, path normalisation provided the functionality to detect Windows Prefetch files which would have been undetectable without removing the variable file name suffix.

In respect of Registry entries, stripping the Registry hive root key was discovered as essential to correlate path information between different Windows versions. Results from full path normalisation (including root key transformation) indicate an effective implementation,
increasing path detection rates by approximately 70%. Case inconsistencies were discovered
between Windows versions and transforming Registry key paths to lower case solved this
issue. Overall, path normalisation results indicate an effective technique to allow correlation
of file system and Registry entries using a normalised full path value.

Figure 7.2 provides a graphic representation of the combined results from file system and
Registry path normalisation matching results between the actual paths (blue) and normalised
paths (orange). The increased number of combined file system and Registry paths detected
in both scenarios is visually evident.

![Figure 7.2: Overview of actual paths versus normalised paths matching results for file system
and Registry entries](image)

7.2.5 Effectiveness of the Windows Registry Processing Technique

The target data set processing component of the system design required the functionality
to parse and represent Windows Registry entry metadata to allow later correlation against
the application profile(s). This research developed a new technique to represent Registry
entries using a revised RegXML \texttt{CellObject} syntax (see Section 5.3.2.2). The design re-
quired that a new Registry parsing tool be implemented to produce output based on the
revised RegXML structure, as well as providing the functionality to recover deleted Registry
entries to facilitate detection of uninstalled application software. The resultant implemented
solution was the \texttt{CellXML-Registry} tool, authored specifically in this research to provide
the required functionality. Testing and analysis of the new Registry processing technique is
required to determine the overall effectiveness, as well as effectiveness when compared to a similar solution.

Testing was performed using all extractable Registry hives from the 79 forensic disk images that comprise the M57-Patents scenario. CellXML-Registry was executed against all 1,272 Registry hive files extracted from the M57-Patents scenario. Results were analysed and compared to previous findings put forward by Nelson [2012], who created the original RegXML structure using the pre-existing hivexml tool coupled with a Python framework to completely automate Registry processing (regxml_extractor). Table 7.18 presents a comparison of different statistics for extracted Registry hive files and their contents from the M57-Patents scenario. For each descriptive statistic a count is provided for the results obtained using CellXML-Registry in this research and the hivexml tool as reported by Nelson [2012].

Table 7.18: Comparison of Registry parsing between CellXML-Registry and hivexml

<table>
<thead>
<tr>
<th>CellXML-Registry</th>
<th>hivexml</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>79</td>
<td>Number of processed disk images</td>
</tr>
<tr>
<td>1272</td>
<td>1041</td>
<td>Number of extracted hive files</td>
</tr>
<tr>
<td>1270</td>
<td>1016</td>
<td>Number of hives processed without error</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>Number of hives processed with error</td>
</tr>
<tr>
<td>29,339,058</td>
<td>16,238,06</td>
<td>Total entry count (keys and values)</td>
</tr>
<tr>
<td>1,384,137</td>
<td>0</td>
<td>Total unallocated entry count (keys and values)</td>
</tr>
</tbody>
</table>

The comparison between this research and Nelson [2012] highlights disparity in results achieved. Both methods processed the same 79 disk images from the M57-Patents scenario. Differences were discovered in the number of Registry hive files extracted, in this research a total of 1,272 hive file were extracted by Vestigium (more specifically, by the HiveExtractor.py module included in Vestigium), while the regxml_extractor tool extracted only 1,041. Compared to the 25 hive files that were not able to be parsed using the hivexml tool, only two hive files were not able to be parsed by CellXML-Registry. The two hive files only failed when CellXML-Registry attempted to recover deleted entries. Without recovery being specified, both of these hive files were processed without error. The issue is an underlying problem with the Registry parser project and more information regarding this matter can be found on the CellXML-Registry Issues listing.

In addition to the difference in extractable Registry hive files, the results indicate variation in the total number of Registry entries (keys and values) processed. The output from CellXML-Registry resulted in approximately 29 million Registry entries, while hivexml only reported just over 16 million. Further investigation of the discrepancy is beyond the scope of this research project. Finally, CellXML-Registry was able to recover approximately 1.3 million unallocated (deleted) Registry entries. In contrast, hivexml reported no unallocated entries as it does not provide support for recovery of deleted content. Overall,

---

15See: https://github.com/thomaslaurenson/CellXML-Registry/issues/1
the results indicate that Vestigium in conjunction with CellXML-Registry proved more effective at extracting Registry entries compared to results presented by Nelson (2012) using regxml_extractor and hivexml.

This section has described and presented results pertaining to effectiveness of the implemented system design. The Vestigium tool was subjected to numerous experimental testing scenarios using a public data set. The next phase of M57-Patents scenario testing involves evaluating the efficiency of the system design.

7.3 M57-Patents Scenario: Efficiency Testing and Findings

The tools used to perform forensic analysis need high computational efficiency due to the requirements of processing a potentially large number of targets that are increasing in size and complexity (see Section 2.3.1 and Section 2.3.2). Therefore, an efficient system is pivotal to solving the challenges of current digital investigations. Furthermore, the high-level research objective specified for a solution capable of meeting digital forensic triage requirements. To reiterate, triage is a preliminary digital examination to determine the intelligence value of digital devices and prioritisation of further in-depth analysis under significant time and resource constraints.

This section presents efficiency results from public data testing using the M57-Patents scenario. Firstly, the efficiency metrics and testing method are briefly outlined. All 79 forensic disk images from the M57-Patents scenario are then used as input to the implemented system (Vestigium) and metrics calculated to determine system efficiency. It becomes apparent that the initial results do not meet digital forensic triage requirements. Based on the results, therefore, system design enhancements are proposed, implemented and tested. Finally, a comparison to other automated forensic analysis and triage solutions is presented to provide a baseline for system evaluation.

7.3.1 Efficiency Metrics and Testing Method

Two efficiency metrics have been previously proposed: 1) Absolute speed, the clock time required to process the target data set; and 2) Relative speed, the average rate that the tool can analyse data compared to the rate at which data can be read from the source device. Both efficiency metrics have been outlined and discussed (see Section 4.3.5) and score interpretations provided (see Section 6.2.6.2). The following hardware and software specifications were used for all M57-Patents scenario efficiency testing:

- Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz
- 8 GB DDR3 1333 MHz
- Western Digital Green 2.0 TB SATA 3 desktop 3.5-inch hard drive
- Microsoft Windows 7 Professional 64-bit

To determine the relative speed of the Vestigium tool, the read speed of the hard drive containing the actual target data set needs to be established. Using the prescribed testing en-
vironment, the hard disk read speed was ascertained using the Microsoft Windows \texttt{winsat} utility which reports hardware performance capability. The following listing provides an example of the command used to determine the sequential read speed of the target hard disk:

\begin{verbatim}
$ \texttt{winsat disk -seq -read -drive f -count 10}
\end{verbatim}

To establish a reliable hard drive read speed, multiple tests were executed. A total of 10 hard drive benchmark tests were performed and an average value determined. Additional information including hard drive read speeds for each test are documented in Appendix D.3. The benchmark results provide the ability to calculate the relative speed to aid in determining system efficiency. The results showed that the average disk read speed was 119.6 Megabytes per second (MB/s).

7.3.2 System Efficiency Results

The specified metrics were calculated based on the processing times reported by \textit{Vestigium} to: 1) Perform file system metadata generation and matching; and 2) Perform Registry metadata generation and matching. Individual processing times were collected at both processing stages. This was achieved by adding functionality to record the time for each \textit{Vestigium} function using the Python \texttt{timeit} module. Table 7.19 displays an overview of the processing time for each \textit{Vestigium} function. The absolute speed (elapsed clock time) and relative time (rate of data processing compared to sequentially reading the entire disk image) to perform file system and Registry processing are shown, as well as an average for each metric per disk.

Table 7.19: Overview of system efficiency for the M57-Patents scenario (all results are displayed in minutes apart from relative speed which is the fraction of time taken compared to sequential hard drive read speed)

<table>
<thead>
<tr>
<th>Vestigium Processing</th>
<th>Efficiency Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>File System</td>
</tr>
<tr>
<td>Total</td>
<td>350.10</td>
</tr>
<tr>
<td>Average</td>
<td>4.43</td>
</tr>
</tbody>
</table>

As previously stated, the M57-Patents scenario disk images are stored in the EWF format using the \textit{best compression} method, with a total compressed size of 454 GB\textsuperscript{18}. When uncompressed, the data set equates to almost 1.5 terabytes (TB) of raw data, while the average uncompressed size per disk is approximately 19 GB. All following efficiency metrics are reported using the uncompressed data set size.

\textbf{Absolute speed:} \textit{Vestigium} took 474 minutes (7.9 hours) to process all 79 disk images, an average of 6 minutes per disk. File system processing took a total of approximately 350

\textsuperscript{17}See: https://docs.python.org/3.4/library/timeit.html
\textsuperscript{18}Compared to RAW disk images without compression, compressed disk images take longer to process as they need to be decompressed on-the-fly when processing
minutes (5.8 hours), while Registry processing took almost 125 minutes (a little over 2 hours). These results show that file system processing and matching took approximately 74% of the total elapsed time, while Registry processing and matching took approximately 26%.

**Relative speed:** Based on absolute speed and input data set size, the average processing speed (see Equation 4.9) was calculated to be 57.5 MB/s, meaning Vestigium analysed all input data at an average of 57.5 MB/s. The disk read speed was previously calculated at 119.6 MB/s. Given these two variables, the overall relative speed was 0.48. Simply put, it took Vestigium approximately twice as long to analyse all data than the time it would take to simply read the data sequentially from the evidence hard drive. Based on expectations, the relative speed result and the time taken to analyse data is disappointing and not considered to be efficient based on digital forensic triage requirements. The expected relative speed to be achieved in the research objective was at least 1.00, the same time it would take to sequentially read all input data. Modifications and refinements were therefore essential to increase efficiency of the implemented system.

Additional analysis of system efficiency was conducted to ascertain more detailed information regarding each step of file system and Registry processing. To achieve this, the processing performed by Vestigium was further divided into separate steps and metrics recalculated. There are a total of four individual processing steps performed by Vestigium: 1) Parse the target disk image using the fiwalk tool to generate a DFXML representation of the file system; 2) Perform file system matching; 3) Parse extracted Registry hive files using the CellXML-Registry tool to generate a RegXML representation of the Registry; and 4) Perform Registry matching. Individual processing times were determined to establish efficiency at each of the four steps of target data set processing. Table 7.20 provides a summary of the four specified steps of Vestigium processing and the absolute speed (elapsed time) for each step is provided, as well as the percentage of processing time to complete each step.

Table 7.20: Overview of efficiency results of the four separate Vestigium processing steps using the M57-Patents scenario (absolute speed is displayed in minutes)

<table>
<thead>
<tr>
<th></th>
<th>File System</th>
<th>Registry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Matching</td>
</tr>
<tr>
<td>Absolute Speed</td>
<td>333.34</td>
<td>16.77</td>
</tr>
<tr>
<td>Percentage</td>
<td>70.22%</td>
<td>3.53%</td>
</tr>
</tbody>
</table>

The results reveal that 70% of total processing time was taken to generate metadata for file system entries using the fiwalk tool. In comparison, Registry metadata generation tool took a little over 4% of total processing time using the CellXML-Registry tool. File system matching took approximately 3.5% of total processing time, while Registry matching took 22% of the total processing time. The discrepancy of efficiency between the matching steps can be explained by the amount of data that required processing. The input to Registry matching was approximately six times as many entries compared to the input to file system matching (see Table 7.2). Approximately, 30 million Registry entries required processing,
while approximately only 5 million file system entries required processing equating to six times the time taken to complete matching. However, based on the overall processing time, matching methods were deemed to be efficient to perform triage. In contrast, further refinements to improve overall system efficiency and, ultimately, achieve the performance requirements needed for rapid forensic triage. Therefore, the techniques used to process the target data set and generation of metadata requires modification.

7.3.3 Improving Vestigium Efficiency

The initial results from efficiency testing fielded informative feedback on each processing step performed by Vestigium. Enhancing overall system efficiency by improving the processing steps executed by Vestigium and the underlying tools used to parse and generate target data set metadata (fiwalk and CellXML-Registry) is now a priority.

7.3.3.1 Improving File System Metadata Generation Efficiency

The efficiency results revealed that approximately 70% of total processing time was taken to parse the file system of the target data set and generate a DFXML metadata report. File system metadata generation has the potential to provide rapid processing of entries as the data size usually includes only 1-5% of the total hard drive size (depending on the amount and type of data stored). However, the results from efficiency testing indicated that file system metadata generation (which includes file hashing) took longer than the time required to sequentially read the data from the hard drive. Further testing was therefore specified to evaluate the fiwalk tool that is used to generate file system metadata. A range of different scenarios required testing to implement a revised and more efficient system: 1) Reporting data file location of the disk image; 2) Enabling file hashing; 3) Disabling file hashing; and 4) Using one or both hashing algorithms. Generating metadata was performed using fiwalk for five different Test Cases (TC) with the following configuration options:

TC01: Parsing all entries without accessing data location or performing hashing

TC02: Parsing all entries and reporting data location (byte_runs) without file hashing

TC03: Parsing all entries, reporting data location with MD5 file hashing

TC04: Parsing all entries, reporting data location with SHA1 file hashing

TC05: Parsing all entries, reporting data location with MD5 and SHA1 file hashing

Each of the prescribed test cases were executed on a single disk image from the M57-Patents scenario (jo-2009-12-11-002) to provide an indication of efficiency in each test case scenario. Table 7.21 displays an overview of the results from invoking fiwalk to perform metadata generation. Different command line arguments (flags) were specified to achieve the efficiency improvements.

---

19 The fiwalk tool required modification to perform metadata extraction without reporting data location or hashing files. This was caused by a bug found in the tool that incorrectly parsed the supplied command line arguments resulting in fiwalk accessing and reporting data location even when specifically requested not to. Additional information, including source code modification is covered in Appendix D.4
desired functionality as specified by the prescribed test cases. Each test was performed 10 times on the same system to ensure a reliable and accurate result.

Table 7.21: Overview of fiwalk metadata generation performance for the last disk image from the user Jo from the M57-Patents scenario (named jo-2009-12-11-002)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Flags</th>
<th>Hash algorithm</th>
<th>Absolute time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC01</td>
<td>-x -z -g -b</td>
<td>None</td>
<td>7.95</td>
</tr>
<tr>
<td>TC02</td>
<td>-x -z</td>
<td>None (but get byte_runs)</td>
<td>62.23</td>
</tr>
<tr>
<td>TC03</td>
<td>-x -z -M</td>
<td>MD5</td>
<td>73.88</td>
</tr>
<tr>
<td>TC04</td>
<td>-x -z -l</td>
<td>SHA1</td>
<td>81.02</td>
</tr>
<tr>
<td>TC05</td>
<td>-x</td>
<td>MD5 &amp; SHA1</td>
<td>95.79</td>
</tr>
</tbody>
</table>

The results show that including the reporting of data location and/or performing file hashing increases absolute processing time. This was entirely expected as both require additional processing time to determine each property. However, what the testing does provide is informative results of performance thus aiding the selection of an alternative solution. Parsing the file allocation table only is comparatively fast, taking approximately 8 seconds. When accessing the data location, processing time increases by more than 680%, taking approximately 62 seconds. Adding MD5 hashing further increases the processing time by an additional 11 seconds, while including SHA1 hashing adds nearly 20 seconds being computationally more intensive compared to the MD5 algorithm. Calculating both file hashing algorithms and reporting data location took the most time, almost 96 seconds.

Parsing the file system allocation table and not reporting data location or performing file hashing is by far the most computationally efficient option available (approximately 8 seconds for a disk image with a 6 GB compressed file size). However, Vestigium requires data location (the byte_runs element) to extract Registry hive files and to perform file hashing (both are essential for effective digital artifact matching). Nevertheless, modifications to file system processing, especially metadata generation, must be addressed based on triage requirements.

Based on the results achieved thus far and the author’s knowledge of potential solutions, it has been proposed that the fastest file system processing method will be implemented; that is, processing the file system allocation table without fetching data location or performing file hashing (represented by Test Case 01). Obviously, the limitations of this approach are that not reporting file location or performing file hashing will be achieved, but can be overcome with the following two solutions:

1) Since data location will not be available to aid extraction of Registry hive files, the icat tool (from The Sleuth Kit, a pre-existing Vestigium dependency) will be used to extract hive files using the available file system inode number to determine data location.

\[20\] See Appendix D.3 for complete results
2) A modified and selective file hashing method, which the author has dubbed selective file hashing, is proposed. This method will enable more efficient processing by only hashing specific files of potential interest, identified based on a known file size (from the application profile) instead of hashing every file on the target data set.

Extracting file location is easily achievable using the icat tool. This removed the need for fiwalk to extract the byte_runs information for every file. Since the inode number is extracted when only parsing the file allocation table this information is readily available. The revised technique was implemented in Vestigium and informal testing conducted to confirm functionality. This required adding a number of new programming functions, including support to check for icat availability on the running system, and to extract each Registry hive file using the icat tool.

Performing efficient file hashing is a more complex scenario when attempting to minimise processing time and increase efficiency. The original implementation required that every data file was hashed, a normal practice in digital investigations. However, as results indicate, this is a time consuming process compared to solely extracting file system metadata. Results revealed that file hashing takes 820% longer than just parsing file system metadata (see Table 7.21 specifically TC01 compared to TC03). To combat this issue, the author proposes to enhance file system processing performance by specifying the data files that require hashing. This is achieved because knowledge is readily available in terms of the data files that are attempted to be detected on the target data set. The information is, therefore, accessed from the application profile; specifically, the file size of known data files. The author has named the proposed method selective file hashing, as only potentially interesting or useful data files are subjected to hash value calculation. The premise is centred on the following fact of secure hash algorithms:

That when performing hash value correlation, only two data files with exactly the same file size will produce the same cryptographic hash value.

Cryptographic hashing will only find the same hash value for exactly the same data that is input to the algorithm. If the same hash value is found for different sized inputs, the algorithm is inherently flawed. Since the system design is searching for known data files from the application profile, there is ground truth information available that includes a file size metadata property for every data file of interest. Therefore, only data files from the target data set with a matching file size will be subjected to file hashing. Theoretically, this should result in reduced numbers of data files needing to be hashed, ultimately, decreasing processing time and increasing system efficiency. The following procedure outlines a high-level overview of the selective file hashing method:

- Parse application profile and create a Python set of known file sizes
- Parse file system using revised method, specifically, only extract metadata from the file allocation table using fiwalk
• For each target data file, only fetch file data and calculate $MD5$ hash value if the file size is present in the list of known file sizes from the application profile.

The specified functionality to perform selective file hashing is not implemented in fiwalk or any other digital forensic file system metadata generation/hashing tools known to the author. Therefore, additional programming code was written and included in Vestigium to achieve the desired functionality. This was implemented by creating a set of all file sizes when parsing the input application profile(s). This results in a set of known file size of interest. When processing each data file from the target data set, the filesize property of each FileObject is checked for existence in the set of known file sizes. If the file size is found, the data file is subjected to $MD5$ file hashing. Since data location is not reported, the icat tool is leveraged and the file extracted into a Python BytesIO object and hashed using the Python built-in hashlib file hashing library. This method allows for hash value calculations without actually extracting and saving the data file which could potentially increase processing time.

### 7.3.3.2 Improving Registry Metadata Generation Efficiency

The efficiency results showed that approximately 4% of total processing time was taken to generate Registry metadata. This result was comparatively efficient compared to other processing steps performed by Vestigium. However, the author noted performance limitations of CellXML-Registry during informal testing in comparison to other Registry tools using the same data set. Therefore, formal testing was performed and analysed to determine the efficiency of the solution implemented in this research compared to other solutions.

A comparison of tool performance for absolute speed (processing time) was conducted for CellXML-Registry against other similar tools. The hivexml tool was selected as it also generates RegXML. However, it does not include deleted Registry entry support. The reglookup tool was used in conjunction with reglookup-recover to provide a solution with the capability to extract allocated and unallocated (deleted) Registry entries. Similar to file hash performance comparison, each test was conducted ten times to provide an average result. The same disk image from the M57-Patents scenario, that was used for file system efficiency testing, was again used to test Registry processing performance (jo-2009-12-11-002). The system had a total of 18 Registry hive files equating to a total size of 31.9 MB. Table 7.22 displays the comparative efficiency results of the three selected tools to perform Registry processing. The reglookup tools were tested on two operating system types, namely Linux and Windows, to determine any differences in efficiency.

Overall, the hivexml tool was the fastest Registry processing tool tested, taking a total of approximately 4.5 seconds to process all 18 Registry hive files from the jo-2009-12-11-002 disk image. However, hivexml does not have the functionality to recover deleted Registry entries, which would require additional processing time to scan and attempt recovery. The combination of reglookup and reglookup-recover provides functionality for deleted Registry entry recovery, but both tools need to be consecutively invoked against each hive.
Table 7.22: Overview of Registry processing tools for the M57-Patents disk image: jo-2009-12-11-002 (absolute time is displayed in seconds)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Platform</th>
<th>Absolute Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CellXML-Registry</td>
<td>1.2.0</td>
<td>Windows</td>
<td>14.52</td>
</tr>
<tr>
<td>hivexml</td>
<td>1.3.9</td>
<td>Linux</td>
<td>4.45</td>
</tr>
<tr>
<td>reglookup &amp; reglookup-recover</td>
<td>1.0.1</td>
<td>Linux</td>
<td>5.88</td>
</tr>
<tr>
<td>reglookup &amp; reglookup-recover</td>
<td>1.0.1</td>
<td>Windows</td>
<td>7.10</td>
</tr>
</tbody>
</table>

Nevertheless, even with executing two tools on each hive file the reglookup combination proved efficient, taking only slightly longer than hivexml to process all input with the Linux option proving to be the most efficient at just over 5 seconds. The CellXML-Registry tool (authored specifically for this research) took more than twice as long compared to other solutions, a total of approximately 14.5 seconds to process all hive files. This test and the achieved results confirmed the efficiency limitations of the CellXML-Registry tool.

While observing CellXML-Registry performance, the RegXML generation phase (saving XML syntax to an output file) took much longer compared to the other solutions. Interestingly, the other two tools output to standard output (stdout) and redirection is required to save the resultant output. CellXML-Registry does not implement this technique, and instead, by default, saves all entries into an XML file format. This is achieved using the .NET framework StreamWriter.WriteLine() function and the technique was adopted as the Registry parser project uses it for all tool output. Given the performance issue surrounding XML generation the following refinements were proposed to the CellXML-Registry tool based on testing results and the author’s knowledge:

1) Re-write existing tool output to print Registry metadata to standard output instead of writing to an XML file

2) Improve XML writing performance by using a standardised library specifically designed for printing XML syntax

Additional research was conducted to re-implement the existing output method used in CellXML-Registry. Support was added to produce XML syntax to standard output using the .NET framework System.Xml library designed specifically for XML processing. It also includes support for encoding, special character handling and general production of well-formed XML documents\(^21\). The Console.WriteLine() function was used to replace all occurrences of the original output function (StreamWriter.WriteLine()) to achieve redirection of XML to standard output. Functionality was retained to write RegXML metadata to an XML file, but this was also updated to use the newly implemented System.Xml library.

In addition to the revised XML output, Vestigium was updated to now directly ingest CellXML-Registry metadata directly from standard output. The same solution is available for fiwalk in combination with the DFXML Objects.py API. Informal testing indicated

that directly parsing standard output from each tool is significantly faster than creating an XML file and parsing the resultant file.

7.3.3.3 Improving Digital Artifact Matching Efficiency

Thus far, refinements have been proposed, designed and implemented into the tools used to perform target data set processing; specifically, metadata generation for the file system and Registry evidence sources. In terms of digital artifact matching, file system matching took 3.5% of the total time processing time, while Registry matching took approximately 22% of total processing time. A number of trials were made in an endeavour to improve digital artifact matching efficiency for file system and Registry entries. This included implementing and informally testing a collection of different programming looping techniques to iterate objects including Python list comprehension, dictionary comprehension and third-party dictionary libraries. However, no major increase in efficiency was obtained for matching. Additionally, informal testing was conducted using the original DFXML `dfxml.py` API which implemented the Simple API for XML (SAX) parser. This implementation proved to be more efficient, but lacks the required functionality provided by the `Objects.py` solution using the Python `ElementTree` library. Eventually, the originally implemented techniques were made without changes the Vestigium matching framework was left unmodified.

7.3.4 Revised System Efficiency Results

The investigation of efficiency at each stage of target data set processing and digital artifact matching performed by Vestigium was undertaken. Various refinements were proposed, designed and implemented into both the Vestigium tool and the underlying target data set processing tools (`fiwalk` and `CellXML-Registry`). Specifically, a revised technique was implemented to achieve faster file system metadata generation by only extracting file allocation table metadata and performing selective file hashing based on known file sizes extracted from the application profile. Furthermore, modifications were made to `CellXML-Registry` to achieve faster Registry metadata generation by changing output generation to use a specialised XML processing library. Informal testing throughout the various revised developments indicated improved system efficiency. However, further formal testing is now required to determine the improvements gained. Additionally, thorough testing is also required to assess any adverse effects on system effectiveness that may have occurred as a result of system modification; primarily, the introduction of selective file hashing.

The revised version of Vestigium was again subjected to experimental testing using all disk images from the M57-Patents scenario. Table 7.23 displays the results with comparison to the original efficiency results (as previously displayed in Table 7.19).

The original results from efficiency testing determined an absolute speed of 474 minutes (7.9 hours) to process all 79 disk image from the M57-Patents scenario, while relative processing speed was calculated at 0.48, meaning that the system took twice as long to process all data when compared to the time required to solely read the data.
Table 7.23: Overview of revised system efficiency results for the M57-Patents scenario (all results are displayed in minutes apart from relative speed which is the fraction of time taken compared to sequential hard drive read read speed)

<table>
<thead>
<tr>
<th>Efficiency Metrics</th>
<th>Vestigium Processing</th>
<th>Efficiency Metrics</th>
<th>Absolute speed</th>
<th>Relative speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>File System</td>
<td>Registry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (original)</td>
<td>350.10</td>
<td>124.59</td>
<td>474.69</td>
<td>0.48</td>
</tr>
<tr>
<td>Average (original)</td>
<td>4.43</td>
<td>1.58</td>
<td>6.01</td>
<td>0.48</td>
</tr>
<tr>
<td>Total (revised)</td>
<td>95.62</td>
<td>85.62</td>
<td>178.25</td>
<td>1.15</td>
</tr>
<tr>
<td>Average (revised)</td>
<td>1.17</td>
<td>1.08</td>
<td>2.26</td>
<td>1.15</td>
</tr>
</tbody>
</table>

**Absolute speed:** The revised system reduced absolute speed by more than half, taking a total of 190 minutes (a little under 3 hours) to process all input. The results dictate a success in improving efficiency, obtaining a 60% decrease in processing time and, ultimately, an increase in system efficiency. The average absolute speed to process a disk image was reduced from 6 minutes to 2.2 minutes. Furthermore, the results show that the time taken to process each evidence source (file system and Registry) is much closer than the original implementation, taking approximately 85-95 minutes in total for each.

**Relative speed:** The revised system implementation increased average analysis speed from 57.5 MB/s to 137.24 MB/s; meaning Vestigium was able to analyse input data at a rate of approximately 137 MB/s. Given this value, the revised relative speed was calculated to be 1.15. This result exceeds the original system requirement of 1.00 and, ultimately, means that the revised system implementation is capable of analysing all disk images from the M57-Patents scenario slightly faster that the time required to solely read disk contents sequentially. Although this seems logically impossible, processing faster than disk read speed is achievable due to Vestigium utilising only metadata for digital artifact matching and, therefore, removing the requirement to read all disk contents. Overall, relative speed results dictate that the performance enhancements and resultant revised implementation was successful at improving efficiency to the requirements specified by the research objective.

Even with an increase in efficiency due to the modifications, it is important that as a result there was no change to the reported effectiveness findings. Analysis of the output from Vestigium revealed there was no change in digital artifact matching effectiveness. All results maintained the same as previously reported in Section 7.2.

The revised file system processing technique, specifically selective file hashing, produced higher overall system efficiency. However, the implemented technique does have a potential limitation. Since all data files are not hashed by Vestigium, the resultant DFXML report of all file system metadata will lack complete file hash information; that is, only data files with a corresponding file size that was observed in the input application profile will have a hash value. This means that the metadata report is less useful for post-processing, specifically, to perform additional file hash analysis; for example, the DFXML report generated by Vestigium
cannot be used to perform additional file hash searches and each non-hashed file would require additional processing. This is a trade-off between performance and coverage. Since this research specified a solution capable of forensic triage, the author decided that efficiency of the system was more important than providing a complete file system metadata report for post-processing. Nevertheless, the original method of complete data file hashing could be easily re-implemented in Vestigium if required. No other limitations were observed.

7.4 M57-Patents Scenario: Discussion and Observations

The results from M57-Patents scenario testing were presented in terms of effectiveness (see Section 7.2) and efficiency (see Section 7.3) of the implemented system. The testing has resulted in a thorough evaluation using a public data set which is readily available for research reproducibility, allowing other researchers and practitioners to verify or build on the results presented in this research. This section presents a summary discussion of the implemented system in terms of effectiveness and efficiency based on the prescribed research objectives. The results obtained are discussed and conclusions drawn based on the findings from experimental testing. Furthermore, identified system limitations encountered during the course of experimental testing are also noted.

7.4.1 System Effectiveness

Effectiveness indicates the functionality to automate the detection of digital artifacts from a target data set using an application profile of known content. The overall effectiveness of the system design was evaluated in terms of file system entry detection (see Section 7.2.1.1) and Windows Registry entry detection (see Section 7.2.1.3). The results indicate an effective system when compared to similar existing solutions commonly used to perform digital forensic analysis (see Section 7.2.1.2 and Section 7.2.1.4).

To reiterate, recall measures the ability to correctly detect relevant digital artifacts, while precision measures the ability to only detect relevant digital artifacts from the target data set. Overall, a high recall score was achieved for automated detection of digital artifacts between the authored application profiles and disk images from the M57-Patents scenario. Recall scores were approximately 0.92, or 92% of all possible digital artifacts, for all testing scenarios apart from the Eraser tool. Investigation revealed that the low recall rates for Eraser was due to the application being rewritten between the version on the data set (5.7.8) and the application profile (6.2.0).

Perfect precision scores (1.00) were obtained for all interesting disk image testing scenarios, while testing on the non-interesting disk image detected a total of 30 file system entries and 90 Registry entries that were false positive. This only impacts on effectiveness metrics for precision scores. All false positives were caused by non-unique digital artifacts in the CCleaner application profile. This was solved by updating the static blacklist used by the LiveDiff tool. The low false positive rate returned by Vestigium is advantageous to investigators as further manual analysis to determine correctness is not required. High
precision scores were thus obtained using the strict metadata matching methods while still maintaining the effective detection of relevant digital artifacts in a variety of complex matching scenarios (e.g., using different application versions and different operating system versions).

The effectiveness of Vestigium was achievable, in part, due to the created application profiles and the low number of irrelevant or non-unique entries. Data reduction techniques had previously been implemented and tested to remove irrelevant operating system noise observed during profile creation. The effectiveness results further illustrate that the profiles contained mainly unique digital artifacts from each anti-forensic tool.

The methods and associated tools authored to recover deleted (unallocated) entries proved effective at detecting residual digital artifacts (see Section 7.2.3). A longitudinal analysis was performed and results indicated that both deleted and residual file system and Registry entries were detectable even after anti-forensic tool uninstallation. However, the life span of deleted evidence was observed to be short (2 days) in the tested scenario. The CellXML-Registry tool authored in this research proved effective at recovering deleted Registry entries and enabled correlation and detection of unallocated entries.

The technique of path normalisation was designed and implemented to aid detection of file system and Registry paths (absolute paths) to aid matching entries between operating system versions and application versions. File system path normalisation resulted in an increase in detectable entries by removing user-specific variables and normalising path differences between operating system versions. Registry path normalisation was also effective as the technique aided in matching entries where Registry root key values differed for all Windows operating system versions. Case sensitivity issues were also discovered in Registry entries and were again solved by path normalisation. Overall, the path normalisation technique can be considered successful, ultimately increasing digital artifact detection rates.

7.4.2 System Efficiency

Efficiency indicates the performance of the implemented system to execute digital artifact detection in a timely manner to achieve rapid triage requirements. Overall, the efficiency of the implemented system proved to maintain acceptable performance standards based on the requirements of digital forensic triage as specified in the high-level research objective. Initial efficiency results were deemed unacceptable to achieve this goal and various performance enhancements were thus made. A technique of selective file hashing was proposed and implemented with the aim of increasing system efficiency. Results obtained from testing the refined system demonstrated that Vestigium was able to analyse the entire M57-Patents scenario at a relative speed of 1.15; meaning that it took less time to analyse all evidence compared to solely read this actual data. Take the example of a 20 GB disk image which takes 10 minutes to sequentially read all data. Vestigium can perform a complete analysis in less than 10 minutes; that is, less time than it would take to only read the data. This is possible by just processing logical data of interest. Overall, Vestigium completely analysed the entire M57-Patent scenario in a little under 3 hours. It included processing the target data set, extracting file system entries, Registry hives and associated Registry entries, generating
metadata for all evidence sources, performing matching using three application profiles and reporting the findings to the investigator.

A vital output of digital forensic triage is the production of processed evidence that can be reused during later in-depth forensic analysis, if required. Not only have triage efficiency requirements been met by Vestigium, but output from initial examination is produced which is of use for later analysis. Target data set processing resulted in machine-readable metadata reports that can easily be reprocessed using a variety of reliable forensic analysis tools.

### 7.4.3 System Design Modifications

The experimental testing of the M57-Patents scenario provided a total of 79 disk images containing over 5 million file system entries, over 30 million Registry entries at a total uncompressed data set size of approximately 1.5 terabytes. Testing on such a large data set provided insight regarding design issues in terms of system effectiveness and efficiency, as well as identification of a variety of software bugs. The following list highlights each of the identified problems discovered during testing that, as a result, have been solved for each implemented software tool.

- **Vestigium**: A variety of bugs were discovered due to a lack of error checking, all of which have been remedied before proceeding to the next stage of testing. Updated target data set processing was implemented to only parse file allocation table metadata and file extraction implemented using the `icat` tool coupled with the Python `BytesIO` module. Selective file hashing was also designed and implemented.

- **fiwalk**: The version used for testing was available on Windows but was out of date and had operation bugs found during testing. Modifications were made to `fiwalk` source code and the revised version cross-compiled for Microsoft Windows.

- **CellXML-Registry**: Slow RegXML generation was observed during testing and a revised output method implemented and tested. A collection of XML parser errors were discovered when ingesting XML output with Vestigium including incorrectly escaped special characters and Unicode control characters. All issues were rectified by updating the XML output checking functions.

- **LiveDiff**: Six false positive `CCleaner` application profile entries were found during testing and the static blacklist updated to exclude these discovered erroneous results.

The listed software limitations, issues and bugs were all satisfactorily resolved before proceeding to the next stage of experimental testing; that is, system evaluation using a real-world data set.

### 7.5 Real-World Data Set Overview: Second Hand Hard Drives

Performing experimental testing to evaluate a system design to aid digital forensic analysis using a real-world data set is advantageous due to data diversity and unpredictability, thereby,
providing more robust research findings \cite{Garfinkel2009}. Therefore, the third and final data set selected for system evaluation is a range of second-hand used hard drives sourced from Internet auction websites (see Section 4.3.3.3). The specific purpose of real-world data set testing is to aid further development and validation of the system and associated forensic tools developed in this research and to discover bugs from processing diverse data. This section outlines an overview of the real world data set employed in this research including hard drive sources, hard drive selection criteria, a high-level overview of data set content and a brief summary of testing method.

7.5.1 Real-World Data Set Sources

The real word data set was compiled from two sources: 1) Previous research conducted at the same university in the same department as the author; and 2) Additional used digital devices purchased by the author specifically for this research. A total of 100 second-hand hard drives were sourced from previous research conducted by Roberts (2013) at the Department of Information Science, University of Otago. Roberts (2013) originally collected the hard drives from three sources: 1) Company A who supplied 50 drives; 2) Company B who supplied 19 drives; and 3) The New Zealand based online auction website TradeMe\footnote{See: http://www.trademe.co.nz/} where 31 extra drives were purchased. In 2013, additional second-hand hard drives were purchased by the author. All items were purchased from the TradeMe website. The criteria for purchasing the additional second-hand hard drives was as follows:

1) Must be a digital storage device that is likely to have a desktop operating system. This results in targeting hard drives with the following specifications: 3.5” and 2.5” internal hard drives with an IDE, SATA or SCSI interface

2) The hard drives must be listed as second-hand (used)

A total of 95 second-hand HDDs were purchased by the author using the specified requirements. This resulted in a combined total of 195 hard drives to be included in the real-world data set testing portion of this research. A complete list of all sourced hard drives is available in Appendix D.5.

7.5.2 Selection Criteria for Real-World Data Sets

The criteria for selecting hard drives to form a real-world data set is to identify applicable and usable devices for this research project; for example, not all hard drives sourced were readable and, therefore, had to be excluded from experimental testing. The following list documents the criteria to select suitable disk images for experimental testing:

1) The hard drive must be readable

2) The hard drive must have a valid file system

3) The hard drive must be installed with a Microsoft Windows operating system

\footnote{See: http://www.trademe.co.nz/}
Adherence to the specified selection criteria is considered by using a linear approach; the order in which they are specified. If a hard drive cannot be forensically imaged using the dc3dd tool\textsuperscript{23}, the hard drive is deemed unreadable and not suited for testing. Table 7.24 displays a summary of the hard drives in terms of readability status for the hard drives sourced from \cite{Roberts:2013} and the hard drives purchased by the author.

<table>
<thead>
<tr>
<th>Source</th>
<th>Total</th>
<th>Unreadable</th>
<th>Readable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roberts (2013)</td>
<td>100</td>
<td>22</td>
<td>78</td>
</tr>
<tr>
<td>Purchases drives</td>
<td>95</td>
<td>17</td>
<td>78</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>195</td>
<td>39</td>
<td>156</td>
</tr>
</tbody>
</table>

The first selection criterion resulted in 156 hard drives that were classified as readable; a total of 78 from \cite{Roberts:2013} and 78 purchased for this research. Two more criteria are required for selection for testing. The system design was implemented to perform forensic analysis using file system (and Windows Registry) metadata. The system design was also implemented to target a Microsoft Windows operating system. Therefore, the target hard drive must have both a valid file system and a Microsoft Windows operating system installed. Technically, the Vestigium tool has the functionality to process any readable hard drive or forensic disk image with, or without, a file system. However, if no file system metadata is present the tool would return no results.

The determined file system must be either: 1) File Allocation Table (FAT) or a member of the FAT file system family (FAT16, FAT32 etc.); or 2) New Technology File System (NTFS). Both file systems are specified as supported by Microsoft Windows for operating system installation\textsuperscript{24}. A final check was performed to determine if the hard drive had a Microsoft Windows operating system installed. A rudimentary test was performed to scan the DFXML report output from \texttt{fiwalk} and search for a Windows directory in the root directory. Table 7.25 displays an overview of the 156 readable HDDs as shown in Table 7.24.

<table>
<thead>
<tr>
<th>Source</th>
<th>NTFS</th>
<th>FAT</th>
<th>Other</th>
<th>None</th>
<th>Has Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roberts (2013)</td>
<td>21</td>
<td>17</td>
<td>3</td>
<td>37</td>
<td>23</td>
</tr>
<tr>
<td>Purchased drives</td>
<td>38</td>
<td>9</td>
<td>1</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>59</td>
<td>26</td>
<td>4</td>
<td>67</td>
<td>55</td>
</tr>
</tbody>
</table>

A file system check was performed using the \texttt{fiwalk} tool which provides support for commonly encountered desktop file systems including FAT, NTFS, EXT and HFS. Only forensic images that could be parsed by \texttt{fiwalk} and produce a valid DFXML report were selected for testing. Furthermore, the determined file system must be either: 1) File Allocation Table (FAT) or a member of the FAT file system family (FAT16, FAT32 etc.); or 2) New Technology File System (NTFS). Both file systems are specified as supported by Microsoft Windows for operating system installation\textsuperscript{24}. A final check was performed to determine if the hard drive had a Microsoft Windows operating system installed. A rudimentary test was performed to scan the DFXML report output from \texttt{fiwalk} and search for a Windows directory in the root directory. Table 7.25 displays an overview of the 156 readable HDDs as shown in Table 7.24.

\textsuperscript{23}dc3dd is a forensic acquisition tool previously used during experimental testing (see Section 6.2.4).

\textsuperscript{24}See: \url{http://windows.microsoft.com/en-nz/windows7/comparing-ntfs-and-fat32-file-systems}
with details of the file system type and a count to indicate how many HDDs potentially have a Microsoft Windows operating system installed.

As displayed in Table 7.25, a total of 59 out of the 156 readable hard drives had a valid NTFS file system, while 26 had a valid FAT file system. A total of 67 hard drives were readable and able to be forensically acquired, but did not have a valid file system. Finally, a total of 4 hard drives had a non-Windows supported file system (including two HFS, one UFS and one EXT file system). From the total of 85 hard drives with a valid file system; that is, 38 from Roberts (2013) and 47 from purchases, not all potentially had a Microsoft Windows operating system. Consequently, a total of 55 second-hand hard drives were found to have a valid FAT or NTFS partition as well as a Windows folder in the root directory. These were selected for experimental testing.

### 7.5.3 Overview of Selected Hard Drives from Real-World Data Set

Prior to performing experimental testing the size and complexity of the data set was investigated and determined. Similar to known data set and public data set testing this was achieved using the same two Python scripts authored to determine data set content (see Section 6.6.1). The file system statistics script (FileSystemStats.py) and Windows Registry statistics script (RegistryStats.py) are available in Appendix C.8 and C.9. Table 7.26 displays an overview of the content of the data set for all 55 disk images and also an average value per disk. The data set size is provided, file system counts in terms of allocated and unallocated (deleted) entries, and pertinent Windows Registry counts including number of hive files and the count of allocated and unallocated entries. Appendix D.6 provides more in-depth

<table>
<thead>
<tr>
<th>Property</th>
<th>Total</th>
<th>Average per disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set size</td>
<td>2,268 GB</td>
<td>41.24 GB</td>
</tr>
<tr>
<td>Directories</td>
<td>715,899</td>
<td>13,016</td>
</tr>
<tr>
<td>Data files</td>
<td>3,629,294</td>
<td>65,987</td>
</tr>
<tr>
<td>Other</td>
<td>204,107</td>
<td>3,711</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>3,181,491</td>
<td>57,845</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>1,367,809</td>
<td>24,869</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4,549,300</strong></td>
<td><strong>82,714</strong></td>
</tr>
<tr>
<td>Hive files</td>
<td>1,247</td>
<td>22.67</td>
</tr>
<tr>
<td>Registry keys</td>
<td>5,572,600</td>
<td>101,320</td>
</tr>
<tr>
<td>Registry values</td>
<td>10,591,395</td>
<td>192,571</td>
</tr>
<tr>
<td>Allocated entries</td>
<td>15,785,832</td>
<td>287,015</td>
</tr>
<tr>
<td>Unallocated entries</td>
<td>378,163</td>
<td>6,876</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>16,163,995</strong></td>
<td><strong>293,891</strong></td>
</tr>
</tbody>
</table>

Table 7.26: Overview of the content of hard drives selected for real-world data set testing
information regarding real-world data set content.

The 55 disk images from the real-world data set equate to approximately 2.2 terabytes of raw uncompressed data (2,268 GB), while the average size for each disk image is approximately 41 GB. This is much larger than the disk image sizes encountered in the M57-Patents scenario, which had an average uncompressed size of approximately 18 GB. The total compressed size for the real-world data set was 975 GB, an average compressed size of approximately 18 GB per disk. Similar to the M57-Patents scenario, the real-world data set was stored in the EWF format using the deflate compression algorithm and best compression method.

The total number of all file system entries is a little over 4.5 million, with approximately 3 million allocated and 1.25 million unallocated file system entries. A total of 1,247 Registry hive files were extracted from the 55 disk images, an average of approximately 22 per disk. From the extracted hive files, a total of over 16 million Registry entries were present, with just under 16 million allocated Registry entries and almost 400 thousand unallocated Registry entries recovered.

### 7.5.4 Real-World Data Set Experimental Testing Method

The experimental testing method used for real-world data set testing follows the same procedure implemented for M57-Patents scenario testing (see Section 7.1.2). The Vestigium tool is used to correlate digital artifacts between the application profiles (APXML documents) and the target data set (each of the 55 selected forensic disk image from the real-world data set). The initial three anti-forensic tool profiles were used for testing, as well as the application profile authored during M57-Patents testing for the XP Keylogger tool. Listing 7.5 displays an example of the command used to invoke Vestigium against one disk image (TM0101ST.E01) from the real-world data set with a user-specified output directory (D:\RWDS\TM0101ST-output) and all three application profiles. The listing specifies new line characters (\^) and comments (REM) using Windows command conventions.

```
$ C:\Python34\python.exe Vestigium.py ^
D:\RWDS\TM0101ST.RAW ^
REM Forensic disk image
D:\RWDS\TM0101ST-output ^
REM Output folder
XPAdvancedKeylogger-2.1-6.1.7601.apxml ^
REM XP Keylogger profile
CCleaner-5.09-6.1.7601.apxml ^
REM CCleaner profile
Eraser-6.2.0.2970-6.1.7601.apxml ^
REM Eraser profile
TrueCrypt-7.1a-6.1.7601.apxml ^
REM TrueCrypt profile
```

Listing 7.5: Command line example to invoke Vestigium against the real-world data set

In contrast to M57-Patents scenario testing, no ground truth information for the real-world data set was established. This is because the primary purpose of this testing is to determine functionality of the implemented system in a diverse and unpredictable data set.
7.6 Real-World Data Set Findings

This section presents the findings from experimental testing using the real-world data set. A real-world investigation scenario is intended to model, as near as possible, the testing of a data set that replicates diverse and random content that would be expected in an actual digital investigation. From the available 156 second-hand hard drives sourced for testing, a total of 55 were selected for testing. This section reports the results from executing the Vestigium tool with all four created application profiles (CCleaner, Eraser, TrueCrypt and XP Keylogger) against the selected hard drives. Effectiveness results in terms of detecting known digital artifacts, followed by results of testing system efficiency. A further investigation regarding efficiency is performed to ascertain differences when Vestigium is executed on a field laptop, this being a common practice in digital forensic triage. Finally, a discussion of results is put forward.

7.6.1 Detecting Anti-forensic Tools using Vestigium

The Vestigium tool was invoked using the prescribed experimental testing method against the compiled real-world data set. As used in all effectiveness testing evaluations, the same four digital artifact classifiers \((tp, tn, fp, fn)\) and four effectiveness metrics \((P, R, F_1, A)\) were calculated. Table 7.27 displays the results for file system matching for target disk images from the real-world data set where one or more digital artifacts were detected.

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>(tp)</th>
<th>(tn)</th>
<th>(fp)</th>
<th>(fn)</th>
<th>(P)</th>
<th>(R)</th>
<th>(F_1)</th>
<th>(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE_91</td>
<td>CCleaner-5.09</td>
<td>2</td>
<td>160,356</td>
<td>0</td>
<td>63</td>
<td>1.00</td>
<td>0.03</td>
<td>0.06</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Vestigium reported one disk image from the real-world data set that contained at least one digital artifact match. The hard drive, named IDE_91, was found to have two file system entries; that is, two true positives that were matched against the CCleaner application profile. The IDE_91 hard drive was originally sourced from the Roberts [2013] data set. Using the created CCleaner application as ground truth, effectiveness metrics were calculated. Overall, the recall score achieved was 0.03 and can be considered low compared to other effectiveness results presented in M57-Patents scenarios testing. There were no false positives detected on any of the 55 hard drives, again, demonstrating the functionality of the implemented matching methods. The following listing displays the full file system path for both detected digital artifacts:

```
%PROGRAMFILES%/CCleaner
%PROGRAMFILES%/CCleaner/uninst.exe
```

Further analysis was performed on IDE_91 to determine the CCleaner version number. However, neither the file system nor Registry revealed any details, but file timestamp information indicates the tool was installed on 2005-08-06, and last accessed (executed) on 2006-
12-08. This confirms that digital artifacts were still detectable on the target data set when using an application profile authored more than ten years later. Considering the uniqueness of the application profile entries it can be concluded that, while only two file system entries were detected, Vestigium was still able to detect the presence of the anti-forensic tool on the hard drive. In the remaining 54 hard drives from the real-world data set there we no other true positive matches detected.

The results from Windows Registry matching against the real-world data set are displayed in Table 7.28.

Table 7.28: Effectiveness results for Registry matching for the real-world data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Profile</th>
<th>tp</th>
<th>tn</th>
<th>fp</th>
<th>fn</th>
<th>P</th>
<th>R</th>
<th>F₁</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE_91</td>
<td>CCleaner-5.09</td>
<td>6</td>
<td>685,393</td>
<td>0</td>
<td>109</td>
<td>1.00</td>
<td>0.05</td>
<td>0.10</td>
<td>1.00</td>
</tr>
</tbody>
</table>

The IDE_91 hard drive, as with file system matching, was the only hard drive that contained detected artifacts in the results reported by Vestigium. Again, effectiveness scores were low for recall and F₁-measure, due to detection of a low proportion of application profile contents. Precision and accuracy had perfect scores (1.00) as there were no false positives detected while also maintaining correct classification of all processed digital artifacts. A total of six Registry entries were detected, all from the SOFTWARE Registry hive file. The listing below specifies the detected full Registry paths:

```
software\microsoft\windows\currentversion\app paths\ccleaner.exe
software\microsoft\windows\currentversion\uninstall\ccleaner
software\microsoft\windows\currentversion\app paths\ccleaner.exe\(default)
software\microsoft\windows\currentversion\app paths\ccleaner.exe\path
software\microsoft\windows\currentversion\uninstall\ccleaner\displayname
software\microsoft\windows\currentversion\uninstall\ccleaner\uninstallstring
```

Overall, the effectiveness results from real-world data set testing were low, with only one hard drive containing the CCleaner tool. However, it was never expected that a high number of digital artifacts would be detected. Real-world data set testing was primarily included in system evaluation to provide a data set that would more closely resemble a real-world investigation scenario. With that said, efficiency testing presents a more crucial aspect of the real-world data set testing process and is now covered in the following subsection.

### 7.6.2 System Efficiency Results

Current digital forensic challenges include the need to analyse a potentially large number of targets for each case which are also increasing in size and complexity. This ultimately means that forensic analysis solutions must be computationally efficient. Furthermore, forensic triage has become an essential part of investigations to identify targets with potential evidential value and prioritise further in-depth analysis of specific targets. Real-world data set testing provides an excellent testing scenario as the contained data closely replicates investigation targets that would be encountered in the field.
The *Vestigium* tool was invoked against all 55 selected second-hand hard drives in the real-world data set. The same experimental testing system, as utilised in M57-Patents scenario testing, was implemented (see Section 7.3.1) and the same efficiency metrics were calculated. To reiterate, absolute speed reports the elapsed clock time and relative speed reports the rate of data processing compared to sequentially reading the entire disk image. Table 7.29 displays the calculated efficiency metrics achieved during real-world data set processing. The absolute speed and relative speed are provided, as well as the processing time taken for *Vestigium* to complete the two stages of operation including file system and Registry processing and matching.

Table 7.29: Overview of system efficiency for real-world data set processing (all results are displayed in minutes apart from relative speed which is the fraction of time taken compared to sequential hard drive read speed)

<table>
<thead>
<tr>
<th>Vestigium Processing</th>
<th>Efficiency Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>File System</td>
</tr>
<tr>
<td>Total</td>
<td>102.89</td>
</tr>
<tr>
<td>Average</td>
<td>1.87</td>
</tr>
</tbody>
</table>

As previously explained, the selected 55 hard drives that form the real-world data set were forensically captured and stored in the EWF format using the *deflate* compression algorithm and best compression option. The total uncompressed size of the data set is 2,268 GB (approximately 2.2 TB), while the total compressed size is 975 GB. This equates to an average uncompressed size of just over 41 GB per disk. The overall size is therefore greater than the M57-Patents scenario disk images, even though there are fewer disks in the real-world data set (55 versus 79), thus presenting a more realistic data set to determine system efficiency (in terms of current trends in data set size).

**Absolute speed:** Real-world data set processing took approximately 176 minutes, a little under 3 hours with an average of 3.2 minutes for each hard drive processed. Complete file system processing and matching by *Vestigium* took 102 minutes, while Registry processing took 73 minutes.

**Relative speed:** The calculated relative speed for real-world data set testing was 1.80, meaning *Vestigium* analysed all input data at almost twice the speed at which the target data set could be sequentially read. Compared to M57-Patents data set efficiency metrics, this is a better result, where the revised system produced a relative speed of 1.15. The average processing speed also increased, from 137.24 MB/s in M57-Patents scenario testing to 215.01 MB/s in real-world data set testing.

The results from real-world data set efficiency testing, again, highlight the performance of the *Vestigium* tool. Results show that a 2 TB data set was able to be analysed in less than 3 hours, almost twice as fast than the data could be forensically captured. However, additional investigation is required to further determine the efficiency of the implemented system under
constrained computing resources such as that experienced in a typical digital forensic triage investigation.

7.6.3 System Efficiency Results: Disk Access Method and Analysis System

The system efficiency results presented for the M57-Patents scenario (see Section 7.3) and the real-world data set (see Section 7.6.2) utilised a normal desktop Personal Computer (PC) to replicate an investigator’s analysis system. However, the objective of this research is for a system design and implementation to fulfil the requirements of digital forensic triage. Therefore, further testing to determine the efficiency when using a computer system with limited resources suitable for field triage needs to be undertaken.

A laptop computer would be a conventional platform on which to execute on-site triage examination before further in-depth analysis is later performed on a more computationally efficient system (e.g., a server with a fast multi-core processor and excessive RAM). When using a triage field laptop there is no option for internal SATA connection of the target hard drive. Therefore, the access method for the target disk also needs to be changed. This is usually achieved by attaching the target hard drive via an external device. And so, a final efficiency testing scenario was executed to determine the performance of the system by: 1) Changing the disk access method to an external hard drive caddy (without changing the analysis system); and then 2) Using a consumer grade laptop with limited processing capabilities with the same external hard drive caddy.

The first step is to change the disk access method used to perform analysis. This test was performed before also changing the analysis system, as changing two configurations on the analysis system at once makes it difficult to determine the effect of each system change. No other modifications to the original testing system were performed. The following external HDD caddy was used:

- Welland Turbo Leopard 2.5”/3.5” SATA to USB 3.0 Dual Dock Enclosure

A test was performed to determine the difference in drive read speed when using the external HDD caddy on the testing system compared to an internal SATA connection. The same hard drive that contained the forensic disk images from the real-world data set was again used (Western Digital Green 2.0 TB SATA 3). The same drive read speed tests were performed using the winsat utility and a total of ten hard drive benchmark tests were performed and an average value determined. The complete hard drive read speed test results are provided in Appendix D.5. The results showed that the average disk read speed after changing the HDD access method was 31.9 Megabytes per second (MB/s). The original testing system using an internal SATA 3 connection achieved an average disk read speed of 119.6 Megabytes per second (MB/s). The much lower speed can be explained by the USB 3.0 connection which is much slower than an internal SATA 3 connection. The result from this test will be used to compare to the drive read speed of the field triage laptop system. The second step of additional efficiency testing was to perform more in-depth testing on a field triage laptop. The specifications of the selected field triage laptop testing system are as follows:

256
The experimental testing method to determine HDD read speed was the same as used during M57-Patents scenario testing, specifically, the use of the `winsat` utility (see Section 7.3.1). The results achieved determined that the sequential hard drive read speed for the field laptop was 32.8 MB/s. Compared to the original desktop testing system (an average read speed of 119.6 MB/s) this is significantly lower. Compared to the original desktop testing system using the same external HDD caddy, the results are very similar (an average read speed of 31.9 MB/s). Thus, the major difference in performance results by changing the analysis systems is due to the reduced hard drive read speed when using an external HDD caddy.

Once again, the same tests were performed by invoking `Vestigium` against the selected disk images from the real-world data set. Table 7.30 displays a comparison of system efficiency for the real-world data set using the original desktop analysis system versus the field triage laptop using the external HDD caddy. The same efficiency metrics are displayed including `Vestigium` processing times for each evidence source, absolute speed and relative speed.

Table 7.30: Overview of system efficiency for real-world data set processing on field triage laptop (all results are displayed in minutes apart from relative speed which is the fraction of time taken compared to sequential hard drive read speed)

<table>
<thead>
<tr>
<th></th>
<th>Vestigium Processing</th>
<th>Efficiency Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>File System</td>
<td>Registry</td>
</tr>
<tr>
<td><strong>Desktop</strong></td>
<td>102.89</td>
<td>73.05</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>1.87</td>
<td>1.33</td>
</tr>
<tr>
<td><strong>Laptop</strong></td>
<td>105.24</td>
<td>75.22</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>1.91</td>
<td>1.37</td>
</tr>
</tbody>
</table>

The results for the field laptop testing scenario has provided a set of interesting results to further evaluate system efficiency. The absolute speed to process all real-world data set input was very close to the desktop system. The desktop system took approximately 176 minutes compared to the field laptop system which only took approximately 4 extra minutes. However, the achieved relative speed varied greatly between the two analysis systems. This is due to the low drive read speed of the laptop system and HDD caddy. The desktop system had a relative speed of 1.80, while the field laptop had a much higher relative speed of 6.39 (higher relative speed is better). This means that the field laptop had better overall efficiency than the
desktop system. In contrast, both analysis systems had a very similar average data analysis speed of approximately 210 MB/s in both scenarios.

The comparison of computer systems used to execute Vestigium has revealed that the tool does not require fast disk access. This is because of the metadata parsing approach presented in this research which requires minimal processing power compared to other forensic analysis solutions. This is both an advantage and a potential disadvantage in some scenarios. Using a more powerful analysis system will not enhance efficiency of Vestigium, as the solution does not lend itself to increased disk read speed or parallel processing. File system and Registry parsing is not parallelised and the tools implemented in this research run on a single CPU core. Informal testing using a server with high-specifications support this premise. This may be problematic as faster performance is not achievable with more computing power, but alternatively is a bonus as the solution can be efficiently run on a portable analysis system. Nevertheless, the system in this research was designed specifically to be capable of running with very low resources, proving a high suitability for forensic triage.

7.6.4 Discussion and Observations

The real-world data set provided unpredictable and diverse data on which to further evaluate the system implemented in this research. The 55 selected second-hand hard drives encompassed approximately 2.2 TB of uncompressed data and included 4.5 million file system entries and over 16 million Registry entries.

In terms of effectiveness, Vestigium detected one hard drive with digital artifacts related to the CCleaner tool. The recall score for file system matching and Registry matching were relatively low compared to both known data set testing and M57-Patents scenario testing. Only two file system entries and six Registry entries were detected. However, it was anticipated that very few digital artifacts would be found due to the inclusion of only four anti-forensic tool profiles. The likelihood of these four tools being present on a small selection of second-hand hard drives would be considered low. Nevertheless, no false positive matches were reported for the entire data set, a good result given the diversity of digital artifacts encountered.

In terms of efficiency, the finalised system implementation is considered to meet the prescribed triage requirements. It took under 3 hours to process a total of 2,268 GB of uncompressed data. An average speed of over 200 MB/s was achieved and the calculated relative speed was 1.80, meaning that Vestigium processed all data almost twice as fast than the time required to solely read disk contents sequentially. This result was faster than that achieved during M57-Patents scenario testing (1.15). The differences between relative speed can be explained by a smaller number of logically grouped data (e.g., allocated and unallocated data files, including Registry hive files). A final efficiency test was conducted using a laptop computer in place of the specified desktop system used in all other efficiency testing. The test was conducted to illustrate the speed and flexibility of the implemented system, being able to be executed as a field-based response triage tool. It was shown that there were minimal differences between efficiency results in the two systems, even when the laptop solution has a disk read speed of only 32.8 MB/s, compared to 119.6 MB/s on the desktop system. However,
there was a dramatic difference in relative speed, the field laptop reaching a score of over 6.00. Ultimately, the metadata analysis approach implemented in this research has proven to be efficient even when operating on a computer with minimal resources.

Real-world data set testing was the final step in the evaluation of the system design and the final element of the Design Science Research Methodology (DSRM) process model. The specified research objectives first raised in Chapter 4 can now be addressed.

7.7 Overview of Achieved Research Objectives

The high-level and various lower-level objectives previously set out have been methodologically worked through to a resolution. System demonstration earlier solved lower-level research objectives one and two (see Section 6.7). The evaluation and associated experimental testing results presented in this chapter has aided in providing results to address lower-level research objectives three, four, five and six. This section outlines the progress made regarding each stated research objective.

**Lower-level Research Objective Three:** To establish an effective and efficient automated technique to generate a metadata representation of the target data set

Functionality of the method used to parse, extract and generate a metadata representation for the file system and Windows Registry evidence sources was proven during demonstration. A thorough evaluation of target data set processing was then performed in this chapter including determining effectiveness of the underlying tools used in the system implementation and efficiency of the implementation with comparison to other similar solutions. Overall, the XML-based approach, using `fiwalk` and `CellXML-Registry` tools, can be considered successful at generating a representation of both file system and Registry entries. File system effectiveness was achieved using `fiwalk`, a robust and thoroughly tested file system parsing tool. However, system efficiency was initially problematic due to the computational overhead of hashing all entries. This was resolved by implementing selective file hashing which reduced the time required to process file system entries while maintaining detection effectiveness. In terms of Registry processing, `CellXML-Registry` proved effective at parsing and generating the revised RegXML syntax for later matching. Specifically, `CellXML-Registry` proved effective at recovering and detecting deleted Registry entries (see Section 7.2.3.3) from the known uninstalled XP Keylogger tool from the M57-Patents scenario. Furthermore, comparison to previous approaches indicated that `CellXML-Registry` extracted more correct Registry entries compared to the `hivexml` tool (see Table 7.18). Although some errors were encountered in both tools, all were rectified during the course of the testing process. Overall, the evaluation of target data set processing encompassed two large and diverse data sets, thousands of Registry hive files and approximately 50 million Registry entries. Testing on such a sizeable and varied data set proved that the solution presented in this research was justifiably robust to carry out target data set processing.
Lower-level Research Objective Four: To determine effective and efficient automated methods to correctly detect relevant digital artifacts from a target data set using the known content from a reference set

The system design included a total of eight matching methods (five for file system entries and three for Registry entries) to perform detection of digital artifacts on the target data set using the application profile of known content as a reference set (see Section 5.5.3). Each matching method prescribed different metadata properties to achieve matching depending on the artifact type. Functionality of the matching methods was observed during system demonstration and results showed that only relevant digital artifacts were detected by Vestigium. Similar results were achieved during system evaluation. A high proportion of correctly detected digital artifacts was observed (true positives), while a low number of incorrectly detected digital artifact was observed (false negatives). The only false positive results were due to erroneous application profile entries for the CCleaner tool (see Section 7.2.2). This was rectified by updating the static blacklist used in LiveDiff to exclude the known irrelevant entries. Overall, the use of rich metadata correlation for detecting digital artifacts was a technique previously not implemented but which has proven to be successful. The metadata matching methods proved effective at detecting a high proportion of relevant artifacts, while still providing enough flexibility to perform correlation in complex matching scenarios.

Lower-level research objectives five and six are related objectives. The objectives specified that digital artifact matching should be achievable in two complex matching scenarios: 1) When detecting digital artifacts from different application software versions (e.g., TrueCrypt version 6.3a versus TrueCrypt version 7.1a); and 2) When detecting the same, or different, application software version on different operating system versions (e.g., detecting TrueCrypt on Windows XP and Windows Vista when the application profile was authored on Windows 7). The primary goal of lower-level objectives five and six was to reduce the requirement of creating an application profile for every application version on every operating system version, a time-consuming process. Both objectives are discussed below.

Lower-level Research Objective Five: To determine effective and efficient techniques to perform detection of digital artifacts created by different application software versions

The results from M57-Patents scenario reported that Vestigium was able to detect a different anti-forensic tool version on a different operating system for all tested anti-forensic tools. However, CCleaner and TrueCrypt had much higher recall rates, while Eraser reported low recall scores. Investigation revealed that this was caused by a complete tool rewrite between the tested Eraser versions. Nevertheless, Vestigium was still capable of detecting digital artifacts from Eraser on the M57-Patents scenario, albeit a low number, even after the tool was completely re-authored. Since the detected digital artifacts are considered unique to the tool, even a low detection rate provides sufficient evidence of anti-forensic tool presence. Real-world data set testing detected CCleaner on one disk image with a different
version than that used to author the application profile. Although a low recall score was again reported, the uniqueness of the detected digital artifacts still determined anti-forensic tool presence, the primary goal of the system created in this research. Compared to other solutions, especially the de facto technique of hash set analysis, Vestigium proved much more effective at performing digital artifact detection between application versions (see Section 7.2.1.2). Overall, based on the findings achieved, lower-level research objective five is considered solved.

The findings also provided interesting information regarding application software behaviour between released versions. Some software versions may introduce only slight changes (e.g., TrueCrypt version 6.3a and 7.1a), while other software differ greatly between newly revised versions (e.g., Eraser 5.8.7 and 6.2.0). Unfortunately, such variations are difficult to gauge without in-depth knowledge of the application or thorough research. However, based on the outcomes from this research, the author suggests that an application profile should be created every 1–2 years for a specific anti-forensic tool (or other application software). This time-frame should provide an application profile which continues to be effective in a range of investigation scenarios.

Lower-level Research Objective Six: To determine effective and efficient techniques to perform detection of application software artifacts on different Windows operating system versions

Matching digital artifacts between Windows operating system versions is difficult due to changes in where and how information is logically stored within the file system and Registry. Lower-level research objective six called for the system to detect application software (anti-forensic tools) between different Windows operating system versions. This was primarily achieved using path normalisation to transform known variable content in the absolute path for all application profile and target data set entries. Results showed that a much higher proportion of both file system and Registry entries were detectable using the full path property when path normalisation was used (see Figure 7.2). Given these results and effectiveness of path normalisation, lower-level research objective six can be considered solved.

7.8 Conclusion

System evaluation has resulted in thorough experimental testing to determine the effectiveness and efficiency of the solution proposed in this research. First, a public data set was used to provide research reproducibility and the ability for other researchers to build on the material presented during this thesis. Second, a real-world data set comprised of second-hand hard drives was used to provide more diverse and unpredictable data, leading to a more robust research output and rigorously tested forensic tools. Throughout the evaluation, comparisons were made to other similar forensic analysis approaches highlighting the strengths and weaknesses of the system design. The results presented in this chapter were assembled to establish if the research objectives had been satisfied. Based on all the testing and evaluation, the
findings of the system design presented in this research can be considered successful; that is, automating detection of digital artifacts from anti-forensic tools, leading to reporting the presence of specific applications of a target data set. In addition, digital forensic triage requirements have been met, maintaining acceptable levels of effectiveness and computational efficiency, while also producing initial examination output.

The next chapter finalises the research conducted by reviewing the high-level research objective and assessing the success of the system design. Future development of the authored tools herein is an important extension and by-product of the research, while at the same time the limitations and scope are also explored. This leads to identification of potential future research areas.
Digital forensics is a branch of forensic science that involves the investigation of any digital or electronic source. Digital forensic triage is an investigation technique that involves the identification of case targets of evidential value and their prioritisation for further analysis, most often performed under significant time and resource constraints. This research has contributed to advancement of the fundamental technology used to perform rapid and automated digital forensic triage to detect the presence, or absence, of anti-forensic tools on a target system. It enables an investigator to quickly identify systems which contain anti-forensic tools, hacking tools or any other software that may be potentially misused. Once identified, further in-depth analysis can be performed on the identified and prioritised targets of forensic interest.

This research project was undertaken using the Design Science Research Methodology (DSRM) process model [Peffers et al., 2007]. The model specifies for research to be conducted in a sequence of stages based on process elements. Firstly, the research area was introduced including a high-level overview of research motivation, methodology, contribution and document structure (see Chapter 1). Pertinent background information regarding digital forensics and anti-forensics then followed (see Chapter 2) augmented by further literature that related specifically to the selected topic of interest including previous anti-forensic detection research, system-level reverse engineering and automated forensic analysis techniques (see Chapter 3). A research methodology was subsequently documented, identifying research problems from the literature review, formulating research objectives and informing the development of an experimental testing method (see Chapter 4).

The objective of this research was to design and implement a solution to perform automated detection of digital artifacts from application software (specifically, anti-forensic tools).
on a target data set, which, to reiterate, led to the high-level research objective as follows:

**High-level Research Objective:** To enable effective, automated detection of relevant digital artifacts to identify application software presence on a target data set. The solution must adhere to digital forensic triage requirements including high system efficiency and initial forensic examination output.

In order to achieve the high-level research objective, a system architecture was proposed, thereafter designed and implemented. The system is comprised of two main stages: 1) Automated identification of relevant digital artifacts from anti-forensic tools leading to the creation of an application profile of known, relevant and unique digital artifacts; and 2) Automated correlation between the application profile(s) and a target data set leading to detection of digital artifacts and evidence of anti-forensic presence, or absence (see Chapter 5). The system design was then developed in the form of software, specifically, a collection of forensic tools.

The demonstration phase of the process took place in a laboratory controlled environment to confirm that the functionality did indeed address the research objective (see Chapter 6). Application profiles were created for three anti-forensic tools and a collection of data reduction techniques were trialled to filter irrelevant operating system noise. From the results of testing and lessons learned a revised method was proposed which led to a final set of profiles being created for experimental testing. A known data set, authored in a laboratory environment to demonstrate the digital artifact matching stage of the system implementation, led to the achievement of the low-level research objectives relating to effectiveness of application profile creation methods (see Section 6.7).

According to Peffers et al. (2007), the evaluation phase of design science includes observing and measuring how well the designed artifact, in this case, the computer software, supports a solution by comparing the research objectives to the observed results using relevant metrics and analysis techniques. The system was thus evaluated to determine effectiveness and efficiency in a real-world investigation scenario (see Chapter 7). Two data sets were used for evaluation: 1) A public data set to provide research reproducibility; and 2) A real-world data set comprised of second-hand hard drives with unknown content providing unpredictable and diverse data.

A selection of system requirements were specified on which to aid the subsequent evaluation of the research objective (see Section 4.2.2.1). In addition, criteria to determine system effectiveness and efficiency were also specified and then applied (see Section 4.2.5). Throughout the evaluation phase, effectiveness and efficiency metrics were calculated to determine the capability of the implemented system, and the following outcomes were accomplished:

- **Effectiveness:** The system was able to detect a high-number of relevant digital artifacts from target data sets in a variety of investigation scenarios. Ultimately, the results dictated that the system was capable of determining the presence (or absence) of anti-forensic tools with a significant level of effectiveness.
- **Efficiency:** The system was able to operate at a satisfactory level of efficiency when processing the target data sets during system evaluation. The results showed that
an overall relative speed of 0.53 was obtained, meaning the system is able to analyse evidence at a rate faster than that required to only perform a forensic acquisition of the target device.

- **Automation:** The system design was focused on a high-level of automation for digital artifact identification and detection. The only stage of the system design which requires manual intervention was during application profile creation, where the application life cycle must be manually created. This was earlier anticipated to be the case, and overall, the system provides a highly automated analysis solution.

- **Triage:** The requirements of performing a partial forensic examination output was achieved by the system design, producing metadata reports for file system and Registry evidence sources for later post-processing and in-depth analysis. Furthermore, the system is capable of the fundamental triage requirement of identification of targets of interest by detecting anti-forensic tool presence.

Based on the system requirements, evaluation criteria and observed evaluation metrics the high-level research objective can be deemed achieved given the results and outcomes attained in this research.

### 8.1 Tool Development Contributions

The implemented system has required the creation of a number of tools to accomplish the desired functionality, specified based on the system design, as well as advancement of pre-existing tools. This section provides a summary of all software tools produced throughout the course of the research and then specifies future development opportunities for each tool.

1) **Vestigium:** an automated forensic analysis framework designed to ingest application profile(s) and a target data set (forensic disk image). The tool generates metadata reports to represent the original evidence sources and performs digital artifact matching on each input

2) **LiveDiff:** a portable live file system-level differencing tool for the Microsoft Windows platform, specifically designed to automate reverse engineering of application software for forensic analysis purposes

3) **apxml.py:** an Application Programming Interface (API) written in Python to provide read and write functionality to the AXML data abstraction, namely application profiles. Additionally, a collection of scripts are included with the project to automate processing of AXML documents including tools to perform application profile intersection and post-mortem dynamic blacklisting testing

4) **CellXML-Registry:** a portable Windows Registry parsing tool for the Microsoft Windows platform, designed to extract allocated and unallocated (deleted) Registry entries and populate in the standardised RegXML data abstraction

The Vestigium tool was designed as a proof-of-concept tool to provide the functionality to automate various independent components of the system design. At the conclusion of
this research it is thought that future development by the author for the Vestigium tool is unlikely. Instead, the author envisions implementing the techniques used in Vestigium, especially file system matching, into other tools to provide run-time detection as well as multi-platform analysis; for example, fiwalk could be extended to ingest APXML documents and search file system metadata while extracting entries. This could also provide a multi-platform solution capable of OS X and Linux analysis. Nevertheless, Vestigium will remain publicly available for other researchers to build on its advanced metadata matching methods.

The overall effectiveness of the LiveDiff tool in cooperation with the APXML data abstraction was evident in various testing scenarios presented in this research. Compared to previous approaches, LiveDiff provides an exceptionally simple and rapid approach to reverse engineering application software using differential analysis to determine file system-level and Registry-level changes. The automated data collection proved a great deal more efficient compared to currently available post-mortem techniques, while still retaining effective digital artifact identification. Furthermore, the APXML data abstraction provides extensive metadata, especially when compared to other output from similar reverse engineering tools, for example Regshot. Given the advantages of LiveDiff, the author fully expects continued development of the tool will take place. Supplementary to the functionality discussed in this research, the author has already further advanced the utility of the tool. Support has been included for block-based hashing of data files\(^1\) and inclusion of entropy calculation for each block segment\(^2\). These additions provide the ability to create an application profile with 512 or 4096 byte size block hashing using the MD5 algorithm. Informal testing has already been performed and the resultant APXML document can be directly read into the hashdb tool and subsequently processed using bulk_extractor. These scenarios demonstrate the performance prospects and potential future usefulness of the LiveDiff tool. In addition, any modifications to LiveDiff would likely result in needed updates to the apxml.py modules, another project the author is keen to promote.

The CellXML-Registry tool proved effective and efficient at extracting and documenting Windows Registry entries in the RegXML data abstraction. The revised RegXML CellObject format implemented in CellXML-Registry provided more functionality than the previous nested structure and proved easy to parse and search. Continued development would be an advantage to future research, especially to implement changes when the underlying Registry parser library is updated. However, the author does not envisage any major functionality updates as the software itself provides the required Registry parsing capability. The fiwalk tool was used in this research to perform file system processing, specifically to parse the file allocation table and generate metadata for each entry. While the tool was not developed in this research, bug fixes and modifications were performed and a cross-compilation method for Microsoft Windows was documented. According to the tool creator, Simson Garfinkel, development of fiwalk has ceased and requires continued development to fully integrate into The Sleuth Kit set of forensic analysis tools. For this reason, the author

\(^1\)See: [https://github.com/thomaslaurenson/LiveDiff/commit/3ce5e40](https://github.com/thomaslaurenson/LiveDiff/commit/3ce5e40)

\(^2\)See: [https://github.com/thomaslaurenson/LiveDiff/commit/03a1878](https://github.com/thomaslaurenson/LiveDiff/commit/03a1878)
has decided on, and begun, further development of the tool including the following goals: 1) Development of a Visual Studio project for native Windows compilation; 2) Support for 64-bit binaries; 3) Inclusion of a more efficient file hashing implementation; and 4) Potential multi-threading support to improve computational efficiency.

8.2 Research Limitations

As with any research project, there were limitations on what and how much was possible to accomplish due to time and resource constraints. During the course of the work, a number of limitations were identified by the author in terms of system design and implementation. Additionally, due to the research scope, various experimental testing limitations were also apparent during demonstration and evaluation of the implemented system. The following subsections provide an overview of the limitations encountered.

8.2.1 System Design and Implementation Limitations

The primary system design limitation identified by the author is that the solution was designed specifically for the Microsoft Windows XP operating system and newer versions. This is true for both stages of the system design; that is, digital artifact identification using LiveDiff and digital artifact matching using Vestigium. Inclusion of the Windows Registry as an evidence source is only feasible on Windows-based operating systems (as it is a Windows-only component) and the solution cannot be easily transferred to other operating system types; for example, Apple OS X uses property lists (plists) to store system, user and application configuration information. The underlying concepts to perform the file system portion of differencing and matching are transferable to other operating system types, but would require modification to the authored tools.

Another limitation of the presented system design is the inclusion of only two evidence sources; that is, the file system and Windows Registry. Other researchers have stated that additional sources of evidence would potentially augment detection of application software [Garfinkel 2010]. Two examples are volatile memory (e.g., RAM dumps) and network traffic (e.g., network packet captures). Another consideration is that suitable forensic data abstractions (similar to DFXML) have yet to be authored that are able to reliably represent each evidence source. The same problem exists for property lists, where no forensic data abstractions are available to suitably represent and process the contained evidence. Nevertheless, this research did amalgamate four digital artifact types (directories, files, Registry keys and values) into a reference set and provided a highly automated processing method. Similar solutions in digital forensics perform matching by solely using files and hash values.

In terms of data file matching, more advanced hashing functions have recently been developed and slowly integrated into existing known and reliable forensic analysis techniques (see Section 3.3.2). It is conceivable that the use of block-based hashing or approximate matching (using similarity digests) may have the potential to aid data file detection, especially in complex matching scenarios, such as with data files that differ between versions. However,
present research using either solution has not been addressed as a means of application software detection. Instead, researchers have focussed on the use of block-based hashing to detect residual data files (Jones et al., 2015), and similarity digests to detect similar versions of common document formats and images (Garfinkel & McCarrin, 2015). The primary reason for excluding advanced hashing algorithms was that the objective in this research specified for a solution that would be suitable for rapid triage, a goal that would be difficult to achieve with modern hashing solutions that have low efficiency (compared to traditional hash functions).

The detection of portable application software was also not included in the system design. Portable applications do not require installation and are invoked directly from a transferrable executable file. The LiveDiff and CellXML-Registry tools authored in this research are examples of portable applications, requiring no installation and a single executable to operate both tools. Since portable applications can be stored in any file system location, detection using full path values would require modification. However, a new matching method could be added to Vestigium which would be similar to the File Deleted and File Hash matching methods. Detection of similar files would be difficult as the new method would require an exact hash match, however, a potential solution would be inclusion of block-based or approximate matching algorithms in this scenario. A related limitation is varying file system paths that a user may install software to; that is, not using the default application installer path. If a user installed software in a non-default location this may cause detection issues. An additional method could solve the detection problem but would require further research.

Neither the LiveDiff or Vestigium tool provide complete multi-platform support. LiveDiff is a Windows only tool. Vestigium has potential for multi-platform support as the primary language is written in Python and available on most modern desktop computer systems. However, the underlying CellXML-Registry tool used for parsing Registry hive files is Windows only, due to reliance on the .NET framework. Providing support for investigators to operate forensic analysis tools on any major platform (e.g., Windows, Linux and OS X) is definitely an advantage, but due to scope and time constraints, multi-platform support was not feasible for this research project.

The resultant software artifacts produced in this research were authored as proof-of-concept tools and although all were tested on three data sets, millions of digital artifacts and on different Windows versions, the tools are still experimental and potentially not considered to be fully reliable for real-world digital investigations. Further testing and community feedback would greatly add to future development, tool refinement and undiscovered bugs. Finally, design and implementation of a Graphical User Interface (GUI) would likely ease technical requirements of operating all of the developed tools. However, the purpose of this research was development and testing of a solution for the identified problem area, not development of a solution applicable for investigators with low technical knowledge.

8.2.2 System Demonstration, Evaluation and Testing Limitations

In addition to the limitations outlined for the designed and implemented system, a variety of limitations were also identified in the experimental testing portion of this research. Again,
the limitations were due to scope of the research conducted, as well as what was achievable in the available time frame.

A range of data sets were used for experimental testing including: 1) A laboratory controlled testing environment using specifically authored known content; 2) A publicly available data set to provide reproducible results; and 3) A real-world data set with diverse and unpredictable data to provide more robust research output. Even though the data sets used for experimental testing were varied and contained a large number of investigation scenarios, testing will always benefit from the inclusion of additional data sets and the different testing scenarios that they would present (though to what extent this would confirm or shift the research conclusions is unknown).

A more important experimental testing limitation centers around the fact that a limit of three anti-forensic tools were selected and used for testing (four in later testing when xp-keylogger was included). The author decided to perform a thorough in-depth analysis of a small number of tools rather than opting to perform a lesser analysis of an increased number of tools. Selecting three primary tools allowed the author to perform a very rigorous review of each tool, as well as the inclusion of testing multiple phases in the application life cycle. Detailed analysis was also performed on the detection of deleted and residual digital artifacts, a scenario rarely covered in similar research, which again was only possible given that only three tools were included. Nevertheless, the research conducted would benefit by widening the scope and challenge with experimental testing of more anti-forensic tools, as well as other types of application software. This is true for both the digital artifact identification and the digital artifact matching stages. Further testing of a wider range of software may yield interesting findings and ultimately inform overall improvements to the implemented system in terms of both effectiveness and efficiency.

Another experimental testing limitation was not including analysis of Solid State Drives (SSDs), especially in terms of the ability to automate detection of residual (deleted) data. The prevalence of SSDs in digital investigations introduces challenges during forensic examination and analysis, especially in terms of the recovery of deleted data files, and also deleted Registry entries. General digital forensics data recovery techniques also suffer from this issue, and it was decided to not include this topic primarily to the scope of the research performed. In addition to forensic analysis of SSDs as target devices, it is likely in the future that SSDs will become common-place in investigator’s analysis systems – providing dramatically increased hard drive read speed capabilities and combating sequential read speed issues observed in traditional hard drives. Again, this was not investigated due to the research scope.

### 8.3 Future Research

The preceding discussion regarding research limitations has provided insight regarding interesting aspects of the research performed and has highlighted areas that would benefit from future research. This section now focusses on prospective research areas for continued development and extension of the solution presented in this research. Furthermore, other practical
uses of the research output are put forward regarding the integration of the techniques designed in this research for forensic analysis projects.

Adopting the LiveDiff tool designed in this research may benefit the NIST Diskprint project as, according to Jones et al. (2015), enhanced computation is specified as a future research goal. Compared to the post-mortem data collection method implemented in the Diskprint project, automated live data collection using LiveDiff could accelerate Diskprint creation time. It was also stated that another future goal related to operating system noise reduction during data collection. The techniques of application profile intersection and dynamic blacklisting designed in this research have demonstrated proven effectiveness and efficiency and would easily be transferable to the Diskprint project.

Grier and Richard III (2015) proposed a new approach to digital evidence acquisition using a technique called sifting collectors. Regions of a disk with expected forensic value are selectively collected using profiles to specify digital artifacts of potential forensic value; for example, a profile for collecting email information may contain: 1) Known file extensions (e.g., .pst, .ost and .pab); 2) Known directories (e.g., Thunderbird/profiles); and 3) Registry hives (e.g., NTUSER.DAT). It is envisioned that contributions made in this research could be leveraged to aid sifting collectors. Application profiles could be readily used to generate a sifting collector profile for anti-forensic tools which would extract only potentially relevant artifacts during data acquisition. Also, sifting collectors may benefit from the path normalisation technique designed herein, allowing a sifting collector profile to be executed against different Windows versions where file locations differ.

Performing forensic analysis using metadata is a common investigation technique due to performance efficiency. Furthermore, it is an advantage to process metadata and extract logically grouped data (e.g., documents and emails) as it represents how people view electronic data. While this research used metadata to represent logically grouped data to provide identification of application software presence, further research is required to prove the reliability of metadata generation tools. Nelson, Steggall, and Long (2014) performed a comparative analysis of metadata generation tools for the XBOX 360 file system. They discovered discrepancies in how tools processed file system entries and reported the data, finding inconsistencies in reported file locations. Furthermore, a co-operative mode was specified where output of multiple metadata generation tools could verify correct evidence representation using a consensus approach. The solutions presented in this research (CellXML-Registry), and other digital forensic metadata generations tools, would benefit from additional testing and comparison of resultant extracted metadata. Specifically, the author noted discrepancies in Windows Registry parsing tools which would benefit from a similar metadata comparison study.

Finally, the use of advanced forensic hashing techniques (block-based hashing and approximate matching) requires investigation for applicability to detect similar files from different versions of application software. This could include executables (e.g., .exe), installers (e.g., .exe), libraries (e.g., .dll) and shortcut files (e.g., .lnk). Thus far, none of the mentioned file types have been analysed to determine suitability of block-based hashing or approximate matching to perform detection. Future research could analyse data files created by different application
versions and determine if either solution could aid detection.

8.4 Conclusion

This research has contributed to advancement of the fundamental technology used to perform automated forensic triage of investigation targets to identify and prioritise digital evidence sources of potential value. Specifically, this research aimed to perform automated effective and efficient identification of anti-forensic tools on a target data set. It provides the capability to execute the implemented system against a potentially large number of investigation targets and be returned results of possible latent anti-forensic tool activity. It also enables identification of potentially interesting targets and the prioritisation of further in-depth analysis on forensically interesting devices.

A new method for application profile creation was implemented and tested that provides a high level of automation for both profile creation and the removal of irrelevant operating system noise. Based on results achieved, the solution is both more effective and more efficient when compared to previous and current solutions used the field of digital forensics. A new method for automated digital artifact correlation was implemented and tested that performs detection of anti-forensic tools to determine the presence, or otherwise, on a target data set. In contrast to similar solutions, the system uses rich metadata to perform digital artifact matching, thus enabling more effective detection than de facto hash set analysis techniques as well as inclusion of more evidence sources such as the evidence-rich Windows Registry.

A collection of novel theoretical contributions have been made throughout this project with experimental testing results to prove capability. Contributions include: 1) An automated system-level reverse engineering method to rapidly create application profile reference sets; 2) Dynamic blacklisting to filter irrelevant operating system noise when performing system-level reverse engineering to identify digital artifacts from application software; 3) Advanced digital artifact matching methods to correlate file system and Registry artifacts and perform matching in complex scenarios. Furthermore, all system design elements have been implemented in the form of computer software which are available under open source licenses to promote further advancement for improvements, research and development.

The 21st century has seen a huge gain in the momentum of the information age, a time in human history that is greatly influenced by computerisation that affects most aspects of our society. In the 1980s personal computers had become widespread. Soon after, the Internet was born and has now expanded into being part of everyday life. It is now commonplace to carry a portable Internet-enabled personal computer in one’s pocket, a smart phone, or have instant connection for a myriad of digital devices such as tablet and laptop computers. Due to this rapidly changing technological ecosystem, computer related crime has become pervasive. Although a relatively new field, digital forensic investigation is now the accepted norm in civil and criminal cases. Continued research and development is vital to combat cybercrime and the criminals that seek to maliciously use our electronic data against us.


2011 International Conference on Intelligent Semantic Web-Services and Applications 
(p. 1-7). New York, NY, USA: ACM.
Concealment Programs Using Passive File System Analysis. In M. S. Olivier & S. Shenoi 
(Eds.), Advances in Digital Forensics II (Vol. 222, p. 171-183). Springer New York.
USB devices in Windows 8. In 2015 International Symposium on Advanced Computing and 
Communication (p. 364-371).
2009-m57-patents/docs/
Dingledine, R., Mathewson, N., & Syverson, P. (2004). Tor: The second-generation onion 
.heidi.ie/
survey of researchers & practitioners attitudes and opinions. In Information Security 
for South Africa (p. 1-8).
Wiley & Sons Inc.
April, 2016, from http://pcsupport.about.com/od/toolsofthetrade/tp/ 
free-data-destruction-software.htm
Forensic Wiki. (2012). Determining OS version from an evidence image - Forensic 
Determining_OS_version_from_an_evidence_image#Windows_NT4
2009(2), 18-20.
Foster, K. (2012). Using Distinct Sectors in Media Sampling and Full Media Analysis to Detect


285
Appendix A provides supplementary material concerning the research methodology specified in this project. Figure A.1 documents the original ethical approval granted by the University of Otago for the analysis of second-hand hard drives. Figure A.2 documents the addition of the author to the approved project.
Dear Dr Wolfe,

I am writing to let you know that, at its recent meeting, the Ethics Committee considered your proposal entitled "Analysing the contents of Second Hand Hard Drives".

As a result of that consideration, the current status of your proposal is: **Approved**

For your future reference, the Ethics Committee's reference code for this project is: **10/239**.

The comments and views expressed by the Ethics Committee concerning your proposal are as follows:

The Committee appreciates the Category A application being made. The additional information provided has allayed initial concerns around personal information that resulted from the receipt of the Category B Reporting Sheet.

Approval is for up to three years. If this project has not been completed within three years from the date of this letter, re-approval must be requested. If the nature, consent, location, procedures or personnel of your approved application change, please advise me in writing.

Yours sincerely,

Mr Gary Witte
Manager, Academic Committees
Tel: 479 9256
Email: gary.witte@otago.ac.nz

c.e. Assoc. Prof M Winkoff Head Department of Information Science

---

Figure A.1: Ethical approval granted for research involving second-hand hard drives
ETHICAL APPROVAL AT DEPARTMENTAL LEVEL OF A
PROPOSAL INVOLVING HUMAN PARTICIPANTS (CATEGORY B)

PLEASE read the important notes appended to this form before completing the sections below

NAME OF DEPARTMENT: Information Science

TITLE OF PROJECT: Analysing Second Hand Hard Drive for the Existence of Confidential or Personal Information #10/239

PROJECTED START DATE OF PROJECT: 5 January 2011 – extended to 16 Jan 2017

STAFF MEMBER RESPONSIBLE FOR PROJECT: Dr. Henry B. Wolfe

NAMES OF OTHER INVESTIGATORS OR INSTRUCTORS: (Please specify whether staff or student. If student, please give the name of the qualification for which the student is enrolled)

Dax Roberts – added (Tom Laurensen, Henry Gee)

BRIEF DESCRIPTION OF THE AIMS: Please give a brief summary (approx. 200 words) of the nature of the proposal:-

We will accept the invitation to join an international consortium headed by British Telecom. That consortium forensically investigates second hand hard drives to determine if sensitive data is left on hard drives when they are sold. We will be investigating a number of second hand hard drives sold within New Zealand to form a baseline classification of data recovered on these drives. The results of this research will be published in the “New Zealand Computer Crime and Security Survey”, and also in international journals with the consortium. The research will also form part of a document for other stakeholders in New Zealand security such as the Government Communications Security Bureau and its Centre for Critical Infrastructure Protection.

Figure A.2: Ethical approval granted to the author for this research
Appendix B provides additional relevant documentation regarding the system design presented in Chapter 5. The following list specifies all included material:

1. Section B.1 documents the complete Registry differential analysis algorithm used in LiveDiff
2. Section B.2 documents a comparison between Regshot and LiveDiff source code
3. Section B.3 documents additional APXML data abstraction information including examples of the creator and metadata elements and a working example of a populated APXML document for the TrueCrypt tool
4. Section B.4 provides additional information regarding the apxml.py API for processing APXML documents including an example script to print statistics from an input APXML document and an example script to convert an APXML document to the CSV file format
5. Section B.5 provides the full source code for the HiveExtractor.py module to extract Registry hive files from a target data set
6. Section B.6 documents source code and programming functions added to the Registry parser library used in the CellXML-Registry tool
7. Section B.7 provides the complete source code for the path normalisation modules designed and implemented for use in the Vestigium tool
B.1 Windows Registry Differencing Algorithm

**Algorithm 2** Differential analysis strategy for Windows Registry entries

```plaintext
1: procedure COMPARE_REGISTRY
2:   for each KEYCONTENT in Snapshot1 (KC1) do
3:     for each KEYCONTENT in Snapshot2 (KC2) do
4:       if (KC2 has previously been matched) then skip KC2
5:     else KC2 is a matching key
6:      end if
7:     for each VALUECONTENT in KC1->FisrtVC (VC1) do
8:       for each VALUECONTENT in KC2->FisrtVC (VC2) do
9:         if (VC2 has not been matched) then continue
10:        end if
11:         if (VC2 type does not equal VC1 type) then continue
12:        end if
13:         if (VC2 name does not equal VC1 name) then continue
14:        end if
15:         if (VC2 data equals VC1 data) then VC2 is matched
16:        else VC2 is modified
17:        end if
18:         if (VC2 is null then VC1 is deleted
19:        end if
20:       end for
21:     end for
22:     for each VALUECONTENT in KC2->FisrtVC (VC2) do
23:       if (VC2 is not matched) then VC2 is new value
24:      end if
25:     end for
26:     if (KC1->FisrtSubKC OR KC2->FisrtSubKC then COMPARE_REGISTRY()
27:    end if
28:   end if
29: if (KC2 is null) then KC1 is deleted
30: for each VALUECONTENT in KC1->FisrtVC (VC1) do
31:   if (VC1 is deleted
32: for each KEYCONTENT in Snapshot2 (KC2) do
33:   if (KC2 has previously been matched) then skip KC2
34:   end if
35:   for each VALUECONTENT in KC2->FisrtVC (VC2) do
36:     if (VC2 then VC1 is deleted
37:   end if
38: end for
39: end for
40: end procedure
```

292
B.2 Comparison of LiveDiff versus Regshot

LiveDiff is authored in the C programming language. The source code from the Regshot project was used as a foundation on which to build LiveDiff. Three source code files from the Regshot project were identified and modified to create LiveDiff: 1) fileshot.c to collect a snapshot of the local file system and compare two snapshots; 2) regshot.c to collect a snapshot of the HKLM and HKU Registry hive files and compare two snapshots; and 3) global.h contains global methods, structures and variables. A total of four source code files were authored specifically for LiveDiff: 1) blacklist.c; 2) dfxml.c; 3) livediff.c; and 4) output.c.

It is important to ascertain the differences between the amount of copied source code and re-implemented (or modified) source code taken from Regshot to build LiveDiff. The following listing provides a count of the number of lines for each source code file (either a .c or .h file) from the LiveDiff tool:

```
$ find . -name '*.c' -o -name '*.h' | xargs wc -l
298 ./blacklist.c
744 ./dfxml.c
1111 ./fileshot.c
623 ./global.h
662 ./livediff.c
353 ./output.c
2113 ./regshot.c
5904 total
```

The following listing provides a count of the number of lines for each source code file (either a .c or .h file) from the Regshot tool:

```
$ find . -name '*.c' -o -name '*.h' | xargs wc -l
235 ./setup.c
1010 ./fileshot.c
648 ./global.h
249 ./version.rc.h
280 ./ui.c
81 ./resource.h
277 ./language.c
149 ./misc.c
265 ./output.c
98 ./version.h
2934 ./regshot.c
550 ./winmain.c
6776 total
```

The first Regshot source code file used was global.h. The following listing displays the source code line counts for each global.c file from LiveDiff and then Regshot. In total, global.h has 347 lines added and 322 lines removed from the source code file. This is approximately half of the entire source code. The remaining unmodified code is primarily
the structures used for data storage during application operation (e.g., FILECONTENT, KEYCONTENT, VALUECONTENT struct definitions).

```
$ diff -b -u livediff/global.h regshot/global.h | grep -E '^\+\' | wc -l
347
$ diff -b -u regshot/global.h livediff/global.h | grep -E '^\+\' | wc -l
322
```

The modifications made to fileshot.c and regshot.c were much more detailed. Table B.1 displays an overview of the code modifications for each programming function in the fileshot.c source code file. The table includes the programming function name, if it is included in Regshot and/or LiveDiff, lines added from modification and lines removed after modification.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>RegShot</th>
<th>LiveDiff</th>
<th>Added Lines</th>
<th>Removed Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculateSHA1()</td>
<td>✓</td>
<td>✓</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>ClearFileMatchFlags()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ClearHeadFileMatchFlags()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CompareFiles()</td>
<td>✓</td>
<td>✓</td>
<td>35</td>
<td>20</td>
</tr>
<tr>
<td>CompareHeadFiles()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DirChainMatch()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FileShot()</td>
<td>✓</td>
<td>✓</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>FindDirChain()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FreeAllFileContents()</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>FreeAllHeadFiles()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GetFilesSnap()</td>
<td>✓</td>
<td>✓</td>
<td>67</td>
<td>65</td>
</tr>
<tr>
<td>GetWholeFileName()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LoadFiles()</td>
<td>✓</td>
<td>✓</td>
<td>39</td>
<td>20</td>
</tr>
<tr>
<td>LoadHeadFiles()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SaveFiles()</td>
<td>✓</td>
<td>✓</td>
<td>33</td>
<td>29</td>
</tr>
<tr>
<td>SaveHeadFiles()</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SearchDirChain()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Total**  

<table>
<thead>
<tr>
<th>Added Lines</th>
<th>Removed Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>261</td>
<td>155</td>
</tr>
</tbody>
</table>

Table B.2 displays an overview of the code modifications for each programming function in the regshot.c source code file. The table includes the programming function name, if it is included in Regshot and/or LiveDiff, lines added from modification and lines removed after modification.
Table B.2: Overview of `regshot.c` functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>RegShot</th>
<th>LiveDiff</th>
<th>Added Lines</th>
<th>Removed Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdjustBuffer()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CheckShotsChronology()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>65</td>
</tr>
<tr>
<td>ClearRegKeyMatchFlags()</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CompareRegKeys()</td>
<td>✓</td>
<td>✓</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>CompareShots()</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>CreateNewResult()</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>DisplayResultInfo()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>DisplayShotInfo()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>EmptyFileBuffer()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>FreeAllCompResults()</td>
<td>✓</td>
<td>✓</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>FreeAllKeyContents()</td>
<td>✓</td>
<td>✓</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>FreeAllValueContents()</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>FreeCompareResult()</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FreeShot()</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>GetRegistrySnap()</td>
<td>✓</td>
<td>✓</td>
<td>76</td>
<td>56</td>
</tr>
<tr>
<td>GetValueDataType()</td>
<td>✗</td>
<td>✓</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>GetWholeKeyName()</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>GetWholeValueData()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>GetWholeValueName()</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>LoadRegKeys()</td>
<td>✓</td>
<td>✓</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>LoadShot()</td>
<td>✓</td>
<td>✓</td>
<td>128</td>
<td>44</td>
</tr>
<tr>
<td>OutputComparisonResult()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>187</td>
</tr>
<tr>
<td>ParseValueData()</td>
<td>✗</td>
<td>✓</td>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>RegShot()</td>
<td>✗</td>
<td>✓</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>ResultToString()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>115</td>
</tr>
<tr>
<td>SaveRegKeys()</td>
<td>✓</td>
<td>✓</td>
<td>48</td>
<td>33</td>
</tr>
<tr>
<td>SaveShot()</td>
<td>✓</td>
<td>✓</td>
<td>69</td>
<td>24</td>
</tr>
<tr>
<td>Shot()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>71</td>
</tr>
<tr>
<td>SwapShots()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>TransData()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>161</td>
</tr>
<tr>
<td>TransformValueData</td>
<td>✗</td>
<td>✓</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>WriteFileBuffer()</td>
<td>✓</td>
<td>✗</td>
<td>0</td>
<td>82</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>28</td>
<td>21</td>
<td>743</td>
<td>1,116</td>
</tr>
</tbody>
</table>
B.3 APXML Data Abstraction

This section provides supplementary information regarding the Application Profile XML (APXML) data abstraction. Additionally, a short working example of a partially populated APXML document for the TrueCrypt tool is also provided. Listing B.1 displays an unpopulated creator element with all available XML tags.

```xml
<creator version='1.0'>
  <program>LiveDiff.exe</program>
  <version>1.0.0</version>
  <build_environment>
    <compiler></compiler>
    <compilation_date></compilation_date>
  </build_environment>
  <execution_environment>
    <os_sysname></os_sysname>
    <os_release></os_release>
    <os_version></os_version>
    <host></host>
    <arch></arch>
    <command_line></command_line>
    <start_date></start_date>
  </execution_environment>
</creator>
```

Listing B.1: Example of the Application Profile XML creator element

Listing B.2 displays an example of a populated metadata element with all available XML tags. The listing provides the default XML namespace attributes that are required to correctly parse an APXML document.

```xml
<metadata
  xmlns:dfxml='http://www.forensicswiki.org/wiki/Category:Digital_Forensics_XML'
  xmlns:regxml='http://www.forensicswiki.org/wiki/RegXML'
  xmlns:delta='http://www.forensicswiki.org/wiki/Forensic_Disk_Differencing'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:dc='http://purl.org/dc/elements/1.1/'>
  <dc:type>Application Profile</dc:type>
  <dc:publisher>thomaslaurenson.com</dc:publisher>
  <dc:date>2015-07-31T10:10-0800</dc:date>
  <application name="TrueCrypt" version="7.1a"/>
</metadata>
```

Listing B.2: Example of the Application Profile XML metadata element

Listing B.3 displays a partially populated APXML document for the TrueCrypt anti-forensic tool. Two FileObjects and two CellObjects are provided.
<?xml version="1.0"?>
  <fileobject delta:new_file="1">
    <filename>C:\Program Files\TrueCrypt</filename>
    <filename_norm>%PROGRAMFILES%/TrueCrypt</filename_norm>
    <basename>TrueCrypt</basename>
    <basename_norm>TrueCrypt</basename_norm>
    <alloc_inode>1</alloc_inode>
    <meta_type>2</meta_type>
    <app_name>TrueCrypt</app_name>
    <app_state>install</app_state>
  </fileobject>

  <cellobject delta:new_cell="1">
    <cellpath>HKLM\SOFTWARE\Classes\AppID\TrueCrypt Format.exe</cellpath>
    <cellpath_norm>software\classes\appid\truecrypt format.exe</cellpath_norm>
    <basename>AppId</basename>
    <name_type>v</name_type>
    <alloc>1</alloc>
    <data_type>REG_SZ</data_type>
    <data>{777DCDFD-C330-480B-B582-B02B57580CC9}</data>
    <app_name>TrueCrypt</app_name>
    <app_state>install</app_state>
  </cellobject>

  <rusage>
    <end_date>2016-01-29T08:38:03Z</end_date>
  </rusage>
</apxml>

Listing B.3: Working example of TrueCrypt APXML document
B.4 APXML API: apxml.py

The APXML document has an associated API, named apxml.py. This supplementary appendix provides two working examples of apxml.py usage: 1) APXMLPrintStats.py parses an APXML document and prints high-level information regarding the count for each digital artifacts type; and 2) APXML2CSV.py parses and APXML document and converts the contents to the CSV file format for viewing in Microsoft Excel, or similar spreadsheet applications. Listing B.4 provides the full source code for APXMLPrintStats.py.

```python
#!/usr/bin/env python3

import os

try:
    import dfxml
except ImportError:
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
sysexit(1)

try:
    import Objects
except ImportError:
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
sysexit(1)

try:
    import apxml
except ImportError:
    print("The apxml.py module is required to run this script")
    print("Now Exiting...")
sysexit(1)

SUFFIXES = {1000: [ 'KB', 'MB', 'GB', 'TB', 'PB', 'ZB', 'YB'],
            1024: [ 'KB', 'MB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    """ Convert a file size to human-readable form """
    if size < 0:
        raise ValueError( 'Input must be non-negative'
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
    if size < multiple:
        return '{0:.1f} {1}'.format(size, suffix)
    raise ValueError('Input too large')
```

298
```python
if __name__=='__main__':
    import argparse
    parser = argparse.ArgumentParser(description='''APXMLPrintStats.py''',
                                     formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument('profile', help='APXML document')
    args = parser.parse_args()

    fi = args.profile
    apxml_obj = apxml.iterparse(fi)
    apxml.generate_stats(apxml_obj)

    if apxml_obj.rusage.end_date is not None:
        start = apxml_obj.creator.execution_environment.start_date
        end = apxml_obj.rusage.end_date
        processing_time = end - start
        time = int(processing_time.total_seconds())

    filesize = os.stat(fi)
    ap_filesize = approximate_size(filesize.st_size)
    name = os.path.basename(fi)

    print("%s,%s,%s,%s,%s,%s,%s" % (name,
                                      "{: ,}".format(apxml_obj.stats.dirs),
                                      "{: ,}".format(apxml_obj.stats.files),
                                      "{: ,}".format(apxml_obj.stats.keys),
                                      "{: ,}".format(apxml_obj.stats.values),
                                      "{: ,}".format(apxml_obj.stats.all),
                                      time,
                                      ap_filesize))
```

Listing B.4: Full source code listing for APXMLPrintStats.py

Listing B.4 provides the full source code for APXML2CSV.py, used to convert an APXML document to the CSV file format.

```python
#!/usr/bin/env python3
import os

try:
    import dfxml
except ImportError:
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
syexit(1)

try:
    import Objects
```

299
except ImportError:
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

try:
    import apxml
except ImportError:
    print("The apxml.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

def make_csv_files(files):
    # Create CSV for file system entries
    files_csv = "files.csv"

    with open(files_csv, 'w') as f:
        f.write("
"app_name",
"app_state",
"annos",
"filename",
"filename_norm",
"basename",
"basename_norm",
"filesize",
"meta_type",
"alloc_name",
"alloc_inode",
"sha1")
        for fi in files:
            f.write("
"app_name",
fi.app_name,
"app_state",
fi.app_state,
"annos",
fi.annos,
"filename",
fi.filename,
"filename_norm",
fi.filename_norm,
"basename",
fi.basename,
"basename_norm",
fi.basename_norm,
"filesize",
fi.filesize,
"meta_type",
fi.meta_type,
"alloc_name",
fi.alloc_name,
"alloc_inode",
fi.alloc_inode,
"sha1")

def make_csv_cells(cells):
    # Create CSV for Registry entries
    cells_csv = "cells.csv"
with open(cells_csv, 'w') as f:
    f.write("%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s
" % (
        "app_name",
        "app_state",
        "annos",
        "cellpath",
        "cellpath_norm",
        "basename",
        "basename_norm",
        "alloc",
        "data_type",
        "data",
        "data_raw")
    )
    for co in cells:
        f.write("%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s
" % (
            co.app_name,
            co.app_state,
            co.annos,
            co.cellpath,
            co.cellpath_norm,
            co.basename,
            co.basename_norm,
            co.alloc,
            co.data_type,
            co.data,
            co.data_raw))

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='''APXML2CSV.py '''
                                    , formatter_class = argparse.RawTextHelpFormatter)
    parser.add_argument('profile', help = 'APXML document')
    args = parser.parse_args()

    files = list()
    cells = list()

    apxml_obj = apxml.iterparse(args.profile)

    for obj in apxml_obj:
        if isinstance(obj, Objects.FileObject):
            files.append(obj)
        if isinstance(obj, Objects.CellObject):
            cells.append(obj)

    make_csv_files(files)
    make_csv_cells(cells)

Listing B.5: Full source code listing for APXML2CSV.py
B.5 Registry Hive Extraction: HiveExtractor.py

Listing B.6 displays the complete source code listing for the HiveExtractor.py script used to parse a target file system, identify Registry hive files and export the detected hive files to a user-specified directory.

```python
#!/usr/bin/env python3

# >>>>>>> DISCLAIMER <<<<<<
# This software is heavily derived from the regxml_extractor project
# originally researched and written by Alex Nelson. The project website
# is available at: https://github.com/ajnelson/regxml_extractor
# The following files from the project were used for reference:
# 1) rx_extract_hives.py
# 2) regxml_extractor.sh
#
# The original project requests the following copyright be retained:
# Copyright (c) 2012, Regents of the University of California
# All rights reserved.

***

Author: Thomas Laurenson
Email: thomas@thomaslaurenson.com
Website: thomaslaurenson.com
Date: 2016/04/21

Description:
HiveExtractor.py is a script to extract Windows Registry hive files from
a target forensic image (evidence file) using a DFXML report generated
by the fiwalk program. Two outputs are produced:
1) Directory of extracted hive files
2) DFXML report of file system and file metadata for extracted hive files

Copyright (c) 2016, Thomas Laurenson

******************************************************************************
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
******************************************************************************
***
```

302
try:
    import Objects
except ImportError:
    print('Error: DFXML Objects.py module is required')
    print('You can download from: https://github.com/simsong/dfxml')
    print('Now Exiting...')
sys.exit(1)

class HiveExtractor:
    def __init__(self, image=None, xml=None, outdir=None, alloc=False):
        self.imagefile = image
        self.xmlfile = xml
        self.outputdir = outdir
        self.allocated = alloc
        self.hives = list()
        self.target_fi_count = 0

    def process_target(self):
        *** Process the target image ***
        print('
>>> Processing target image for hive files ...')
        for (event, obj) in Objects.iterparse(self.xmlfile):
            if isinstance(obj, Objects.FileObject):
                if isinstance(obj, Objects.FileObject):
                    self.extract_hives(obj)
                return

    def extract_hives(self, fi):
        *** Match hives based on file names from DFXML. Hive file names are
        taken from: Windows Registry Forensics by Carvey (2011, p.18),
        which is referenced in the regxml_extractor project. ***
        if fi.filename is None:
            return
        fn = fi.filename.lower()
        self.target_fi_count += 1
        if self.target_fi_count % 5000 == 0:
            print("Processed %d files from target DFXML file");

__version__ = "0.1.0"
% self.target_fi_count)

# List of known hive file names
hive_names = ['ntuser.dat',
              'repair/sam',
              'repair/security',
              'repair/software',
              'repair/system',
              'system32/config/sam',
              'system32/config/security',
              'system32/config/software',
              'system32/config/system',
              'system32/config/components',
              'local settings/application data/microsoft/windows/usrclass.dat']

# Find hive files using file name matching from fiwalk DFXML output
for hive_name in hive_names:
    if (fn.endswith(hive_name) and (self.allocated and fi.is_allocated())):
        self.extract(fi)
    elif not allocated and fn.endswith(hive_name):
        self.extract(fi)

def extract(self, fi):
    out_fn = fi.filename
    out_fn = out_fn.replace('/', '-').replace(' ', '-')
    out_fpath = os.path.join(self.outputdir, out_fn)
    # Open output file and write file contents
    with open(out_fpath, 'wb') as f:
        contents = fi.byte_runs.iter_contents(self.imagefile)
        contents = b''.join(contents)
        f.write(contents)
    f.close()
    # Check the SHA-1 of fileobject VS extracted hive
    if fi.sha1 is not None:
        shal = self.sha1_file(out_fpath)
        if shal != fi.sha1:
            print("Warning: SHA-1 hash mismatch for: %s"
                  % os.path.basename(out_fpath))
        self.hives.append(fi)

def sha1_file(self, fi):
    """Helper method to SHA-1 hash extracted hive file """
    hasher = hashlib.sha1()
    with open(fi, 'rb') as f:
        buf = f.read()
        hasher.update(buf)
    return hasher.hexdigest()

def dfxml_report(self):
"**Generate a DFXML report**

dc = {"name": os.path.basename(__file__),
    "type": "Hash List",
    "date": datetime.datetime.now().isoformat(),
    "os_sysname": platform.system(),
    "os_version": platform.version(),
    "os_host": platform.node(),
    "os_arch": platform.machine()}

dfxml = Objects.DFXMLObject(command_line = " ".join(sys.argv),
    sources = [self.imagefile],
    dc = dc,
    files = self.hives)

# Write a temp DFXML file, format it, then write to logfile
temp_fi = io.StringIO(dfxml.to_dfxml())
xml_fi = xml.dom.minidom.parse(temp_fi)

report_fn = os.path.basename(self.imagefile)
report_fn = os.path.splitext(report_fn)[0] + ".xml"
report_fn = os.path.join(self.outputdir, report_fn)

print("\n>>> DFXML Report : %s\n" % report_fn)

with open(report_fn, 'w') as f:
    f.write(xml_fi.toprettyxml(indent=" "))

if __name__=='__main__':
    parser = argparse.ArgumentParser(description="HiveExtractor.py is a script to extract Windows Registry hive files from a target forensic image (evidence file) using a DFXML report generated by the fswalk program. Two outputs are produced:
1) Directory of extracted hive files
2) DFXML report of file system and file metadata for extracted hive files ", formatter_class = argparse.RawTextHelpFormatter)
    parser.add_argument("-a",
                        help = "Only extract allocated hive files ",
                        action = "store_true",
                        default = False)
    parser.add_argument("-z",
                        help = "Zap (delete) the output dir if it exists ",
                        action = "store_true",
                        default = False)
args = parser.parse_args()
imagefile = args.imagefile
outputdir = args.outputdir
xmlfile = args.dfxml
allocated = args.a
zapdir = args.z

# Make output directory
if os.path.exists(outputdir):
    if zapdir:
        shutil.rmtree(outputdir)
        os.makedirs(outputdir)
    elif not os.path.exists(outputdir):
        os.makedirs(outputdir)

# Check for DFXML input, generate if not supplied
if xmlfile == None:
    print("\n>>> No fiwalk DFXML report provided")
    print(" Running fiwalk now...")
    print(" This may take a long time depending on target size...")
    xmlfile = os.path.splitext(imagefile)[0] + ".xml"
    command = "fiwalk -X " + xmlfile + " " + imagefile
    sysrc = os.system(command)
    if sysrc:
        print("\nAn error occured when running fiwalk.")

# Extract Registry hive files
registry_fis = list()
print(" 
>>> EXTRACTING REGISTRY HIVES")
print(" 
>>> EXTRACTING REGISTRY HIVES")
he = HiveExtractor(image = imagefile,
                   xml = xmlfile,
                   outdir = outputdir,
                   alloc = allocated)
he.process_target()
he.dfxml_report()
B.6

CellXML-Registry Source Code

The CellXML-Registry tool utilised the underlying functionality provided by the Registry
parser project created by Eric Zimmerman.1 However, source code additions were required to the Registry parser library to provide functionality to export to the prescribed
RegXML syntax. This section documents the changes made to the Registry parser library, specifically modifications to the original RegistryHive.cs source code file2 . Due
to the difficulty in determining the changes made to the library by the author, this appendix
section documents all additional programming functions added. Listing B.7 displays the two
added functions for checking XML output strings for special XML characters and Unicode
control characters.
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

p r i v a t e s t r i n g SpecialXMLCharacterCheck ( s t r i n g XMLstring ) {
i f ( XMLstring . ToLowerInvariant ( ) . IndexOf ( ’ & ’ ) != −1)
{
XMLstring = XMLstring . R e p l a c e ( "&" , "&amp ; " ) ;
}
i f ( XMLstring . ToLowerInvariant ( ) . IndexOf ( ’ < ’ ) != −1)
{
XMLstring = XMLstring . R e p l a c e ( "<" , "&l t ; " ) ;
}
i f ( XMLstring . ToLowerInvariant ( ) . IndexOf ( ’ > ’ ) != −1)
{
XMLstring = XMLstring . R e p l a c e ( ">" , "&g t ; " ) ;
}
i f ( XMLstring . ToLowerInvariant ( ) . IndexOf ( ’ " ’ ) != −1)
{
XMLstring = XMLstring . R e p l a c e ( " \ " " , "&quot ; " ) ;
}
i f ( XMLstring . ToLowerInvariant ( ) . IndexOf ( ’ \ ’ ’ ) != −1)
{
XMLstring = XMLstring . R e p l a c e ( " ’ " , "&apos ; " ) ;
}
r e t u r n XMLstring ;
}
p r i v a t e s t r i n g ControlXMLCharacterCheck ( s t r i n g XMLstring ) {
c h a r [ ] arrForm = XMLstring . ToCharArray ( ) ;
S t r i n g B u i l d e r b u f f e r = new S t r i n g B u i l d e r ( XMLstring . Length ) ;
foreach ( c h a r ch i n arrForm )
i f ( ! Char . I s C o n t r o l ( ch ) ) b u f f e r . Append ( ch ) ;
return b u f f e r . ToString ( ) ;
}

Listing B.7: Special character checking functions added to RegistryHive.cs
1
2

See: https://github.com/EricZimmerman/Registry
See: github.com/EricZimmerman/Registry/blob/master/Registry/RegistryHive.cs

307


Listing B.8 displays the ExportDataToXMLFormat() function which was added to provide support for RegXML output.

```csharp
    public void ExportDataToXMLFormat(string outfile, bool deletedOnly)
    {
        var KeyCount = 0;
        var ValueCount = 0;
        var KeyCountDeleted = 0;
        var ValueCountDeleted = 0;

        Console.WriteLine(" > Starting XML generation . . . ");

        var header = new StringBuilder();
        using (var sw = new StreamWriter(outfile, false))
        {
            sw.AutoFlush = true;
            sw.WriteLine(header.ToString());
            sw.WriteLine("<hive>");

            if (!deletedOnly)
            {
                if (Root.LastWriteTime != null)
                {
                    KeyCount = 1;
                    sw.WriteLine("<cellobject root='1'/>");
                    sw.WriteLine("<cellpath>[0]<cellpath>", Root.KeyPath);
                    sw.WriteLine("<name_type>k<name_type>");
                    sw.WriteLine("<mtime>{0}</mtime>", Root.LastWriteTime.UtcDateTime.ToString("o");
                    sw.WriteLine("<alloc >1</alloc>");
                    sw.WriteLine("<byte_runs>");
                    sw.WriteLine("<byte_run file_offset="{0}" len="{1}"/>", Root.NKRecord.AbsoluteOffset, (Root.NKRecord.Size - Root.NKRecord.Padding.Length));
                    sw.WriteLine("</byte_runs>");
                    sw.WriteLine("</cellobject>");
                }

                foreach (var val in Root.Values)
                {
                    ValueCount += 1;
                    sw.WriteLine("<cellobject>");
                    sw.WriteLine("<cellpath>{0}\{1}</cellpath>", val.ValueName);
                    sw.WriteLine("<basename>{0}</basename>", val.ValueName);
                    sw.WriteLine("<mtime>{0}</mtime>", val.LastWriteTime.UtcDateTime.ToString("o");
                    sw.WriteLine("<alloc >1</alloc>");
                    sw.WriteLine("<byte_runs>");
                    sw.WriteLine("<byte_run file_offset="{0}" len="{1}"/>", val.NKRecord.AbsoluteOffset, (val.NKRecord.Size - val.NKRecord.Padding.Length));
                    sw.WriteLine("</byte_runs>");
                    sw.WriteLine("</cellobject>");
                }

                Console.WriteLine(" > Ending XML generation . . . ");
            }
        }
    }
```
sw.WriteLine(" <name_type>v</name_type>");
if (val.VKRecord.IsFree)
{
    sw.WriteLine(" <alloc>0</alloc>");
    ValueCountDeleted += 1;
}
else
{
    sw.WriteLine(" <alloc>1</alloc>");
    ValueCount += 1;
}
sw.WriteLine(" <data_type>{0}</data_type>", val.VKRecord.DataType);
sw.WriteLine(" <data>{0}</data>", BitConverter.ToString(val.VKRecord.ValueDataRaw).Replace("−", " "));
sw.WriteLine(" <byte_runs>");
if (val.VKRecord.DataType != VKCellRecord.DataTypeEnum.RegNone)
{
    // Two byte_run elements are written because:
    // 1st: Points to the absolute offset of the VK record
    // 2nd: Points to the actual offset where the data is
    sw.WriteLine(" <byte_run file_offset="{0}" len="{1}"/>", val.VKRecord.AbsoluteOffset, ((val.VKRecord.Size) * (-1) - val.VKRecord.Padding.Length));
    sw.WriteLine(" <byte_run file_offset="{0}" len="{1}"/>", (val.VKRecord.OffsetToData + 4096), val.VKRecord.ValueDataRaw.Length);
}
sw.WriteLine(" </byte_runs>");
sw.WriteLine(" </cellobject>");
}
// Now start recursively dumping subkeys from the ROOT key
DumpKeyXMLFormat(Root, sw, ref KeyCount, ref ValueCount, ref KeyCountDeleted, ref ValueCountDeleted);
}

var theRest = CellRecords.Where(a => a.Value.IsReferenced == false);
//may not need to if we do not care about orphaned values
foreach (var keyValuePair in theRest)
{
    //Console.WriteLine("{0}\n\n", keyValuePair.);
    try
    {
        if (keyValuePair.Value.Signature == "vk")
        {
            ValueCountDeleted += 1;
            var val = keyValuePair.Value as VKCellRecord;
string data = BitConverter.ToString(val.ValueDataRaw).Replace("-", " ");

// Check the cell path (key + value name) for:
// 1) Special characters
// 2) Unicode control characters
string KeyCellpath = SpecialXMLCharacterCheck(val.ValueName);
KeyCellpath = ControlXMLCharacterCheck(KeyCellpath);

// Write CellObject
sw.WriteLine(@"<cellobject>
<cellpath>{0}</cellpath>
<basename>{1}</basename>
<name_type>v</name_type>
<alloc>{2}</alloc>
<data_type>{3}</data_type>
<data>{4}</data>
<byte_runs>
<byte_run file_offset="{5}" len="{6}"/>
<byte_run file_offset="{7}" len="{8}"/>
</byte_runs>
</cellobject>" ,
KeyCellpath ,
Convert.ToInt32(val.IsFree) ,
val.DataType ,
data ,
val.AbsoluteOffset ,
(val.Size * (-1) - val.Padding.Length) ,
(val.OffsetToData + 4096) ,
val.DataLength);

if (keyValuePair.Value.Signature == "nk")
{
    // this should never be once we re-enable deleted key rebuilding
    KeyCountDeleted += 1;
    var nk = keyValuePair.Value as NKCellRecord;
    var key = new RegistryKey(nk, null);
    string KeyCellpath = SpecialXMLCharacterCheck(key.KeyPath);
    KeyCellpath = ControlXMLCharacterCheck(KeyCellpath);
    sw.WriteLine(@"<cellpath>{0}</cellpath>
    <basename>{1}</basename>
    <name_type>k</name_type>
    <mtime>{2}</mtime>" ,
    KeyCellpath ,
    nk.Size ,
    nk.Type ,
    nk.OffsetToData ,
    nk.DataLength);
Listing B.8: RegXML output function added to RegistryHive.cs

Listing B.9 displays the DumpKeyXMLFormat() function which was added to provide support for RegXML output. The function recursively exports a Registry key, its sub-key and associated values to the RegXML syntax.

```csharp
private void DumpKeyXMLFormat(RegistryKey key, StreamWriter sw, ref int keyCount, ref int valueCount, ref int keyCountDeleted, ref int valueCountDeleted)
{
    // Iterate through each subkey
    foreach (var subkey in key.SubKeys)
    {
        // Write XML for Registry subkey entry
        sw.WriteLine("<cellobject>");

        // Perform a special character check
        string KeyCellpath = SpecialXMLCharacterCheck(subkey.KeyPath);
        KeyCellpath = ControlXMLCharacterCheck(KeyCellpath);
        sw.WriteLine(" <cellpath >{0}</cellpath>", KeyCellpath);

        // Write XML for Registry subkey entry
        sw.WriteLine("<cellobject>" ,
            KeyCellpath ,
            key.LastWriteTime.Value.UtcDateTime.ToString("o") ,
            key.NKRecord.AbsoluteOffset ,
            (key.NKRecord.Size - key.NKRecord.Padding.Length));

        DumpKeyXMLFormat(key, sw, ref keyCount, ref valueCount, ref keyCountDeleted, ref valueCountDeleted);
    }
    catch (Exception ex)
    {
        _logger.Warn("There was an error exporting free record at offset 0x{0:X}. Error: {1}", keyValuePair.Value.AbsoluteOffset, ex.Message);
    }
    sw.WriteLine("</hive>");
    _logger.Info(">>> total_keys: {0}", keyCount);
    _logger.Info(">>> total_values: {0}", valueCount);
    _logger.Info(">>> total_deleted_keys: {0}", keyCountDeleted);
    _logger.Info(">>> total_deleted_values: {0}", valueCountDeleted);
}
```
14    sw.WriteLine(" <name_type>k</name_type>");
15    sw.WriteLine(" <mtime>{0}</mtime>", subkey.LastWriteTime.Value.
16        UtcDateTime.ToString("o");)
17    if (subkey.NKRecord.IsDeleted)
18    {
19        sw.WriteLine(" <alloc>0</alloc>");
20        keyCountDeleted += 1;
21    }
22    else
23    {
24        sw.WriteLine("<alloc>1</alloc>");
25        keyCount += 1;
26    }
27    sw.WriteLine("<byte_runs>");
28    sw.WriteLine("<byte_run file_offset="{0}" len="{1}"/>",
29        subkey.NKRecord.AbsoluteOffset , (subkey.NKRecord.Size − subkey.
30        NKRecord.Padding.Length));
31    sw.WriteLine("</byte_runs>");
32    sw.WriteLine("</cellobject>");
33
34    // Iterate through each value
35    foreach (var val in subkey.Values)
36    {
37        // Write XML for Registry value entry
38        sw.WriteLine("<cellobject>");
39
40        // Perform a special character check
41        string ValueKeypath = SpecialXMLCharacterCheck(subkey.KeyPath);
42        ValueKeypath = ControlXMLCharacterCheck(ValueKeypath);
43        string ValueBasename = SpecialXMLCharacterCheck(val.ValueName);
44        ValueBasename = ControlXMLCharacterCheck(ValueBasename);
45        string ValueCellpath = String.Join("\", new String[]
46            {ValueKeypath, ValueBasename});
47        sw.WriteLine("<cellpath>{0}</cellpath>", ValueCellpath);
48        sw.WriteLine("<basename>{0}</basename>", ValueBasename);
49        sw.WriteLine(" <name_type>v</name_type>");
50        if (val.VKRecord.IsFree)
51        {
52            sw.WriteLine(" <alloc>0</alloc>");
53            valueCountDeleted += 1;
54        }
55        else
56        {
57            sw.WriteLine(" <alloc>1</alloc>");
58            valueCount += 1;
59        }
60        sw.WriteLine("<data_type>{0}</data_type>", val.VKRecord.
61            DataType);
62        sw.WriteLine("<data>{0}</data>", BitConverter.ToString(val.
if (val.VKRecord.DataType != VKCellRecord.DataTypeEnum.RegNone)
{
    // Two byte_run elements are written because:
    // 1st: Points to the absolute offset of the VK record
    // 2nd: Points to the actual offset where the data is
    sw.WriteLine("<byte_run file_offset="+ val.VKRecord.AbsoluteOffset +" len="+ val.VKRecord.Size * (-1) - val.VKRecord.Padding.Length +"/>
    , val.VKRecord.OffsetToData + 4096, val.VKRecord.ValueDataRaw.Length);
}
sw.WriteLine("</byte_runs>");
sw.WriteLine("</cellobject>");

// Finished with this key, process the next subkey
DumpKeyXMLFormat(subkey, sw, ref keyCount, ref valueCount, ref keyCountDeleted, ref valueCountDeleted);

Listing B.9: RegXML generation function added to RegistryHive.cs
B.7 Path Normalisation Modules

Two Python modules (libraries that can be imported and the functions utilised) were authored to perform path normalisation. Listing B.10 displays the complete source code listing for the FilePathNormaliser.py module used to perform file system path normalisation in the Vestigium tool.

```python
#!/usr/bin/python

__version__ = "1.0.0"

import collections

class FilePathNormaliser:
    def __init__(self):
        """Initialise FilePathNormaliser object. """
        self.variable_paths = collections.OrderedDict()
        self.variable_paths["programfiles"] = [
            "Program Files",
            "Program Files (x86)"
        ]
        self.variable_paths["allusersprofile"] = [
            "Documents and Settings/All Users",
            "ProgramData",
            "Users/Public"
        ]
        self.variable_paths["userprofile"] = [
            "Users",
            "Documents and Settings"
        ]
        self.variable_paths["localappdata"] = [
            "%USERPROFILE%/Local Settings/Application Data",
            "%USERPROFILE%/AppData/Local"
        ]
        self.variable_paths["appdata"] = [
            "%USERPROFILE%/Application Data",
            "%USERPROFILE%/AppData/Roaming"
        ]
        self.variable_paths["startmenu"] = [
            "%ALLUSERSPROFILE%/Start Menu",
            "%ALLUSERSPROFILE%/Microsoft/Windows/Start Menu",
            "%APPDATA%/Microsoft/Windows/Start Menu",
            "%USERPROFILE%/Start Menu",
            "%USERPROFILE%\%APPDATA%\Microsoft/Windows/Start Menu"
        ]
        self.variable_paths["windir"] = [
            "Windows",
            "WINDOWS"
        ]
```
```python
self.variable_paths["systemroot"] = [
    "%WINDIR%/system32",
    "%WINDIR%/System32"
]

self.variable_paths["prefetch"] = [
    "%WINDIR%/prefetch",
    "%WINDIR%/Prefetch"
]

def normalize(self, fullpath):
    """ Normalize a logical file system path value of a target file """
    # Check root directory
    if fullpath.startswith("C:\\"):
        fullpath = fullpath[3:]

    # Check/replace backslash characters in path
    temp = fullpath.split("\\")
    fullpath = "/" . join(temp)

    # Now, normalize full path
    for key in self.variable_paths:
        for name in self.variable_paths[key]:
            # Normalize Program Files path
            if key == "programfiles":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%PROGRAMFILES%")
            # Normalize All Users path
            elif key == "allusersprofile":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%ALLUSERSPROFILE%")
            # Normalize Local App Data path
            elif key == "localappdata":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%LOCALAPPDATA%")
            # Normalize Windows directory path
            if key == "windir":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%WINDIR%")
            # Normalize Windows System Root path
            if key == "systemroot":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%SYSTEMROOT%")
            # Normalize Windows Prefetch path (and filename)
            if key == "prefetch":
                if fullpath.startswith(name):
                    fullpath = fullpath.replace(name, "%PREFETCH%")
                # Also normalize prefetch name (remove random number string suffix to allow path matching)
                if fullpath.endswith(".pf"):
                    index = fullpath.index(".pf") - 9
```
Listing B.10: Full source code listing for FilePathNormaliser.py

Listing B.11 displays the full source code for the CellPathNormaliser.py module used to transform the full cell_path of Registry entries to a normalised representation.
normpath = normpath[5:]
if normpath.startswith("HKU\"):
    normpath = normpath[4:]
    normpath = normpath.split("\\")
    del normpath[0]
    normpath = "\\".join(normpath)
    normpath = "NTUSER.DAT\\" + normpath
return normpath

def normalize_rootkey(self, cellpath, rootkey):
    """Normalize the cellpath rootkey of the Target CellObject (TCO)"
    Before: CMI-CreateHive{F10156BE....5DA29D131144}\ControlSet001
    After: SYSTEM\ControlSet001"
    # Normalise hive rootkey, or return if None
    if cellpath:
        normpath = cellpath.split("\\")
        normpath[0] = rootkey
        normpath = "\\".join(normpath)
        return normpath
    else:
        return cellpath

def normalize_cellpath(self, cellpath, rootkey):
    """Normalize the cellpath of the Target CellObject (TCO)""
    # Split a cellpath on backslashes, or return if None
    if cellpath:
        normpath = cellpath.split("\\")
    else:
        return cellpath
    # System hive normalisation
    if rootkey == "system":
        control_names = ["\controlset001",
                        "\controlset002",
                        "\controlset003",
                        "\currentcontrolset",
                        "\clone"]
        # If path has "\controlset" name, normalise target path
        # See: http://support.microsoft.com/kb/100010
        #for name in control_names:
        if len(normpath) >= 2:
            if normpath[1] in control_names:
                normpath[1] = "\%controlset%"
        # Join the split normalised path and return
        normpath = "\\".join(normpath)
    return normpath
def normalise_basename(self, basename):
    # If the basename is a path, normalize using the normalize
    # function from the FilePathNormaliser module
    basename_norm = None

    # Decrypt a UserAssist entry ('P:' equates to 'C:' using rot13)
    if basename.startswith("P:"):
        basename_norm = codecs.decode(obj.basename, "rot_13")

    # Strip UserAssist prefix in older windows versions
    elif basename.startswith("HRZR_EHACNGU"):
        basename_norm = basename[12:]  # Strip UserAssist prefix

    else:
        basename_norm = basename

    # If basename_norm starts with C:
    if basename_norm.startswith("C:"):
        basename_norm = basename_norm[3:]  # Remove leading C:

    # Replace backslash with forward slash
    basename_norm = basename_norm.replace("\", "/")

    # Now normalise using file path normaliser modules
    basename_norm = self.file_path_normaliser.normalize(basename_norm)

    return basename_norm

Listing B.11: Full source code listing for CellPathNormaliser.py
Appendix C provides additional relevant documentation regarding the system demonstration presented in Chapter 6. The following list specifies all included material:

1. Section C.1 provides the Python script used to detect the Windows operating system version of the target data set
2. Section C.2 provides an overview of all 75 created application profiles for system demonstration
3. Section C.3 provides the Python script used to perform application profile intersection
4. Section C.4 documents full results from application profile intersection
5. Section C.5 provides the Python script used to perform post-mortem blacklist, as used in experimental testing
6. Section C.7 documents the contents of the final application profiles created during system demonstration and evaluation
7. Section C.8 and C.9 provides the Python script used to determine the file system and Registry content of all data sets used in this research
8. Section C.10 documents the digital artifacts counts observed in the known data set
Listing C.1 displays a simple Python script authored to automatically determine if the Microsoft Windows operating system is present on a target data set with unknown content.

```python
#!/usr/bin/python
__version__ = "1.0.0"

try:
    import dfxml
except ImportError:
    print('Error: DFXML dfxml.py module is required')
    print('You can download from: https://github.com/simsong/dfxml')
    print('Now Exiting...
    sys.exit(1)

try:
    import Objects
except ImportError:
    print('Error: DFXML Objects.py module is required')
    print('You can download from: https://github.com/simsong/dfxml')
    print('Now Exiting...
    sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
def check_filesystem(fi):
    # Check filename exists, then extract root folder
    if fi.filename:
        root = fi.filename.split('/')[0]
    else:
        return
    if root.lower() == "windows" or root.lower() == "winnt":
        return True
    else:
        return False

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
def extract_software_hive(fi):
    # Check filename exists, then extract root folder
    if fi.filename:
        if fi.filename.lower().endswith("system32/config/system")
            out_fpath = fi.filename.replace('/','−').replace('-','−')
            with open(out_fpath, 'wb') as f:
                contents = fi.byte_runs.iter_contents(self.imagefile)
                contents = b"".join(contents)
                f.write(contents)
            f.close()
        return out_fpath
```

```python
def extract_windows_version(co):
    if co.cellpath:
        if "Microsoft\CurrentVersion\ProductName" in co.cellpath:
            print(" > ProductName: %s" % co.data)
        if "Microsoft\CurrentVersion\CSDVersion" in co.cellpath:
            print(" > CSDVersion: %s" % co.data)
        if "Microsoft\CurrentVersion\ProductId" in co.cellpath:
            print(" > ProductId: %s" % co.data)

if __name__=="__main__":
    import argparse
    parser = argparse.ArgumentParser(description="DetectWindows.py")
    parser.add_argument("target", help="Target disk image or DFXML report")
    args = parser.parse_args()

    # Check file system for root "windows" folder
    for (event, obj) in Objects.iterparse(target):
        if isinstance(obj, Objects.FileObject):
            has_windows = check_filesystem(obj)
            if has_windows:
                print(">>> Windows root folder detected...")
                break

    # Extract software hive file
    for (event, obj) in Objects.iterparse(target):
        if isinstance(obj, Objects.FileObject):
            software_hive = extract_software_hive(obj)
            if software_hive:
                print(">>> SOFTWARE hive file detected and extracted...")
                break

    # Locate Registry key, and extract required values
    for (event, obj) in Objects.iterparse_CellObjects(software_hive):
        if isinstance(obj, Objects.CellObject):
            extract_windows_version(obj)
```

Listing C.1: Full source code listing for `DetectWindows.py`
C.2 Created Application Profiles

The first phase of system demonstration involved creating application profiles using the implemented system design, namely, the LiveDiff tool. Three anti-forensic tools were selected to create application profiles: 1) CCleaner; 2) Eraser; and 3) TrueCrypt. Five separate application life cycles were recreated and data collection performed, including: 1) Install; 2) Open; 3) Close; 4) Uninstall; and 5) Reboot.

Three tables are included in this subsection. Each table presents a summary of application profile contents and metadata for each of the three selected anti-forensic tools. For each anti-forensic tool the following information is provided:

1) A count for each digital artifact type: directory, file, Registry key and Registry value
2) A total count of all digital artifact types
3) The time taken to create the application profile (presented in seconds)
4) The file size of the resultant application profile (APXML document)
5) The average of all properties
6) The total of all properties

The following tables are included:

1) Table C.1 presents a summary of application profile contents for CCleaner
2) Table C.2 presents a summary of application profile contents for Eraser
3) Table C.3 presents a summary of application profile contents for TrueCrypt
Table C.1: Summary of application profile contents for CCleaner version 5.09

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>Time (sec)</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC01</td>
<td>19</td>
<td>415</td>
<td>96</td>
<td>447</td>
<td>977</td>
<td>241</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC02</td>
<td>19</td>
<td>403</td>
<td>96</td>
<td>462</td>
<td>980</td>
<td>246</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC03</td>
<td>19</td>
<td>405</td>
<td>96</td>
<td>435</td>
<td>955</td>
<td>245</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC04</td>
<td>19</td>
<td>406</td>
<td>96</td>
<td>490</td>
<td>1,011</td>
<td>250</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>CC05</td>
<td>19</td>
<td>403</td>
<td>96</td>
<td>462</td>
<td>980</td>
<td>249</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC06</td>
<td>19</td>
<td>411</td>
<td>97</td>
<td>455</td>
<td>982</td>
<td>335</td>
<td>2.3 MB</td>
</tr>
<tr>
<td>CC07</td>
<td>19</td>
<td>386</td>
<td>97</td>
<td>441</td>
<td>943</td>
<td>250</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC08</td>
<td>19</td>
<td>395</td>
<td>95</td>
<td>444</td>
<td>953</td>
<td>245</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC09</td>
<td>19</td>
<td>382</td>
<td>96</td>
<td>455</td>
<td>952</td>
<td>240</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC10</td>
<td>19</td>
<td>376</td>
<td>96</td>
<td>440</td>
<td>931</td>
<td>240</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>CC11</td>
<td>19</td>
<td>383</td>
<td>96</td>
<td>439</td>
<td>937</td>
<td>244</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC12</td>
<td>19</td>
<td>385</td>
<td>96</td>
<td>464</td>
<td>964</td>
<td>243</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>CC13</td>
<td>19</td>
<td>370</td>
<td>95</td>
<td>430</td>
<td>914</td>
<td>231</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>CC14</td>
<td>19</td>
<td>405</td>
<td>96</td>
<td>467</td>
<td>987</td>
<td>230</td>
<td>3.5 MB</td>
</tr>
<tr>
<td>CC15</td>
<td>19</td>
<td>385</td>
<td>96</td>
<td>455</td>
<td>955</td>
<td>234</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC16</td>
<td>19</td>
<td>404</td>
<td>96</td>
<td>471</td>
<td>990</td>
<td>238</td>
<td>3.5 MB</td>
</tr>
<tr>
<td>CC17</td>
<td>19</td>
<td>401</td>
<td>96</td>
<td>433</td>
<td>949</td>
<td>230</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC18</td>
<td>19</td>
<td>396</td>
<td>96</td>
<td>463</td>
<td>974</td>
<td>230</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC19</td>
<td>19</td>
<td>383</td>
<td>96</td>
<td>458</td>
<td>956</td>
<td>238</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC20</td>
<td>19</td>
<td>369</td>
<td>97</td>
<td>482</td>
<td>967</td>
<td>231</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC21</td>
<td>19</td>
<td>359</td>
<td>97</td>
<td>451</td>
<td>920</td>
<td>237</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>CC22</td>
<td>19</td>
<td>390</td>
<td>96</td>
<td>456</td>
<td>961</td>
<td>230</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC23</td>
<td>19</td>
<td>382</td>
<td>95</td>
<td>430</td>
<td>926</td>
<td>236</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>CC24</td>
<td>19</td>
<td>386</td>
<td>95</td>
<td>442</td>
<td>942</td>
<td>234</td>
<td>2.2 MB</td>
</tr>
<tr>
<td>CC25</td>
<td>19</td>
<td>407</td>
<td>96</td>
<td>460</td>
<td>982</td>
<td>243</td>
<td>2.2 MB</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>19</td>
<td>391</td>
<td>96</td>
<td>453</td>
<td>960</td>
<td>242.8</td>
<td>2.3 MB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>469</td>
<td>9,787</td>
<td>11,332</td>
<td>23,988</td>
<td>6,070</td>
<td>57.7 MB</td>
<td></td>
</tr>
</tbody>
</table>
Table C.2: Summary of application profile contents for **Eraser** version 6.2.0.2970

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>Time (sec)</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER01</td>
<td>25</td>
<td>301</td>
<td>201</td>
<td>2,112</td>
<td>2,639</td>
<td>257</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER02</td>
<td>25</td>
<td>380</td>
<td>201</td>
<td>2,102</td>
<td>2,708</td>
<td>272</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER03</td>
<td>25</td>
<td>292</td>
<td>194</td>
<td>2,108</td>
<td>2,619</td>
<td>255</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>ER04</td>
<td>25</td>
<td>309</td>
<td>203</td>
<td>2,104</td>
<td>2,641</td>
<td>260</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER05</td>
<td>25</td>
<td>350</td>
<td>203</td>
<td>2,114</td>
<td>2,692</td>
<td>261</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER06</td>
<td>25</td>
<td>350</td>
<td>201</td>
<td>2,119</td>
<td>2,695</td>
<td>269</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER07</td>
<td>25</td>
<td>312</td>
<td>203</td>
<td>2,108</td>
<td>2,648</td>
<td>262</td>
<td>3.3 MB</td>
</tr>
<tr>
<td>ER08</td>
<td>25</td>
<td>384</td>
<td>203</td>
<td>2,111</td>
<td>2,723</td>
<td>263</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER09</td>
<td>25</td>
<td>299</td>
<td>201</td>
<td>2,110</td>
<td>2,635</td>
<td>259</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER10</td>
<td>25</td>
<td>418</td>
<td>201</td>
<td>2,124</td>
<td>2,768</td>
<td>265</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER11</td>
<td>25</td>
<td>316</td>
<td>203</td>
<td>2,132</td>
<td>2,766</td>
<td>265</td>
<td>3.4 MB</td>
</tr>
<tr>
<td>ER12</td>
<td>25</td>
<td>340</td>
<td>200</td>
<td>2,111</td>
<td>2,676</td>
<td>257</td>
<td>3.3 MB</td>
</tr>
<tr>
<td>ER13</td>
<td>25</td>
<td>297</td>
<td>200</td>
<td>2,106</td>
<td>2,628</td>
<td>257</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER14</td>
<td>25</td>
<td>355</td>
<td>203</td>
<td>2,130</td>
<td>2,713</td>
<td>254</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER15</td>
<td>25</td>
<td>389</td>
<td>201</td>
<td>2,116</td>
<td>2,731</td>
<td>266</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER16</td>
<td>25</td>
<td>355</td>
<td>203</td>
<td>2,131</td>
<td>2,714</td>
<td>259</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER17</td>
<td>25</td>
<td>437</td>
<td>203</td>
<td>2,117</td>
<td>2,782</td>
<td>268</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER18</td>
<td>25</td>
<td>299</td>
<td>200</td>
<td>2,110</td>
<td>2,634</td>
<td>259</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER19</td>
<td>25</td>
<td>301</td>
<td>201</td>
<td>2,113</td>
<td>2,640</td>
<td>263</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER20</td>
<td>25</td>
<td>291</td>
<td>202</td>
<td>2,117</td>
<td>2,635</td>
<td>256</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER21</td>
<td>25</td>
<td>310</td>
<td>201</td>
<td>2,117</td>
<td>2,653</td>
<td>263</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER22</td>
<td>25</td>
<td>309</td>
<td>203</td>
<td>2,115</td>
<td>2,652</td>
<td>256</td>
<td>3.2 MB</td>
</tr>
<tr>
<td>ER23</td>
<td>25</td>
<td>294</td>
<td>201</td>
<td>2,145</td>
<td>2,665</td>
<td>252</td>
<td>3.4 MB</td>
</tr>
<tr>
<td>ER24</td>
<td>25</td>
<td>317</td>
<td>201</td>
<td>2,126</td>
<td>2,669</td>
<td>255</td>
<td>4.5 MB</td>
</tr>
<tr>
<td>ER25</td>
<td>25</td>
<td>312</td>
<td>203</td>
<td>2,115</td>
<td>2,655</td>
<td>258</td>
<td>3.2 MB</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>25</strong></td>
<td><strong>333</strong></td>
<td><strong>201</strong></td>
<td><strong>2,117</strong></td>
<td><strong>2,676</strong></td>
<td><strong>260</strong></td>
<td><strong>3.3 MB</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>625</strong></td>
<td><strong>8,317</strong></td>
<td><strong>5,036</strong></td>
<td><strong>52,913</strong></td>
<td><strong>66,891</strong></td>
<td><strong>6,511</strong></td>
<td><strong>81.8 MB</strong></td>
</tr>
</tbody>
</table>
Table C.3: Summary of application profile contents for TrueCrypt version 7.1a

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
<th>Time (sec)</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC01</td>
<td>15</td>
<td>232</td>
<td>153</td>
<td>526</td>
<td>926</td>
<td>245</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>TC02</td>
<td>15</td>
<td>226</td>
<td>153</td>
<td>511</td>
<td>905</td>
<td>238</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC03</td>
<td>15</td>
<td>233</td>
<td>153</td>
<td>511</td>
<td>912</td>
<td>239</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC04</td>
<td>15</td>
<td>215</td>
<td>153</td>
<td>507</td>
<td>890</td>
<td>242</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC05</td>
<td>15</td>
<td>236</td>
<td>154</td>
<td>529</td>
<td>934</td>
<td>240</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC06</td>
<td>15</td>
<td>230</td>
<td>154</td>
<td>531</td>
<td>930</td>
<td>235</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC07</td>
<td>15</td>
<td>237</td>
<td>154</td>
<td>532</td>
<td>938</td>
<td>240</td>
<td>2.3 MB</td>
</tr>
<tr>
<td>TC08</td>
<td>15</td>
<td>237</td>
<td>154</td>
<td>536</td>
<td>942</td>
<td>240</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC09</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>524</td>
<td>913</td>
<td>241</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC10</td>
<td>15</td>
<td>238</td>
<td>154</td>
<td>530</td>
<td>937</td>
<td>237</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC11</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>526</td>
<td>915</td>
<td>239</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC12</td>
<td>15</td>
<td>219</td>
<td>154</td>
<td>507</td>
<td>895</td>
<td>233</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC13</td>
<td>15</td>
<td>210</td>
<td>153</td>
<td>507</td>
<td>885</td>
<td>231</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC14</td>
<td>15</td>
<td>235</td>
<td>154</td>
<td>527</td>
<td>931</td>
<td>234</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC15</td>
<td>15</td>
<td>236</td>
<td>154</td>
<td>512</td>
<td>917</td>
<td>236</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC16</td>
<td>15</td>
<td>218</td>
<td>154</td>
<td>527</td>
<td>914</td>
<td>243</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC17</td>
<td>15</td>
<td>228</td>
<td>153</td>
<td>533</td>
<td>929</td>
<td>247</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>TC18</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>527</td>
<td>916</td>
<td>242</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC19</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>528</td>
<td>917</td>
<td>240</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC20</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>525</td>
<td>914</td>
<td>238</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC21</td>
<td>15</td>
<td>221</td>
<td>155</td>
<td>523</td>
<td>914</td>
<td>244</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC22</td>
<td>15</td>
<td>217</td>
<td>153</td>
<td>530</td>
<td>915</td>
<td>252</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>TC23</td>
<td>15</td>
<td>229</td>
<td>154</td>
<td>516</td>
<td>914</td>
<td>236</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC24</td>
<td>15</td>
<td>220</td>
<td>157</td>
<td>521</td>
<td>913</td>
<td>246</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>TC25</td>
<td>15</td>
<td>220</td>
<td>154</td>
<td>523</td>
<td>912</td>
<td>241</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>Average</td>
<td>15</td>
<td>225</td>
<td>154</td>
<td>523</td>
<td>917</td>
<td>240</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>Total</td>
<td>375</td>
<td>5,637</td>
<td>3,847</td>
<td>13,069</td>
<td>22,928</td>
<td>5,999</td>
<td>53.6 MB</td>
</tr>
</tbody>
</table>

325
C.3 Application Profile Intersection: **APXMLIntersection.py**

Listing C.2 displays the full source code for the APXMLIntersection.py script.

```
#!/usr/bin/env python3

###
Author: Thomas Laurenson
Email: thomas@thomaslaurenson.com
Website: thomaslaurenson.com
Date: 2016/01/04

Description:
The APXMLIntersection.py Python module takes an AXML document as input and normalises FileObjects and CellObjects properties.

Copyright (c) 2016, Thomas Laurenson

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

>>> CHANGETLOG:
  0.1.0 Base functionality ()

***

import os
import sys
import io
import xml.dom.minidom

try:
    import dfxml
except ImportError:
    print("Error: APXMLIntersection.py")
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)
```
try:
    import Objects
except ImportError:
    print("Error: APXMLIntersection.py")
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

try:
    import apxml
except ImportError:
    print("Error: APXMLIntersection.py")
    print("The apxml.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

# Intersect object
class Intersection(object):
    def __init__(self, profilesList):
        self.profileList = list()
        self.order = list()

        # DFXML and RegXML Objects to store FileObjects
        # and CellObjects found in the APXML documents
        self.dfxml_obj = Objects.DFXMLObject()
        self.regxml_obj = Objects.RegXMLObject()

        # Keep record of profile name for output
        self.out_fn = os.path.basename(profilesList[0])
        self.out_fn = os.path.splitext(self.out_fn)[0]

        # Parse each APXML file to a OrderedDict
        for i, profile in enumerate(profilesList):
            print(" > %s" % profile)
            apxml_obj = apxml.iterparse(profile)
            apxml.generate_stats(apxml_obj)
            # Split the file system path for the application profile
            name = profile.split('/')[-1]
            name = name[len(name) - 1]
            #name = name.split('*-*')[0]
            apxml_obj.name = name
            self.profileList.append(apxml_obj)

    def sort_profiles(self, method):
        *** Sort profiles by selected method. ***
        if method == None:
            pass
elif method == "highest":
    self.profileList.sort(key=lambda x: x.stats.all, reverse=True)
elif method == "lowest":
    self.profileList.sort(key=lambda x: x.stats.all, reverse=False)
elif method == "stacked":
    self.profileList.sort(key=lambda x: x.stats.all, reverse=False)
    low = list(self.profileList)
    high = list(self.profileList)
    low.sort(key=lambda x: x.stats.all, reverse=False)
    high.sort(key=lambda x: x.stats.all, reverse=True)
    low = low[0:10]
    high = high[0:10]

del self.profileList[:]
for (h, l) in zip(high, low):
    self.profileList.append(h)
    self.profileList.append(l)

def first_pass(self):
    """Process the first profile. """
    self.order.append(self.profileList[0])
    for obj in self.profileList[0]:
        if isinstance(obj, Objects.FileObject):
            obj.count = 1
            self.dfxml_obj.append(obj)
        if isinstance(obj, Objects.CellObject):
            obj.count = 1
            self.regxml_obj.append(obj)

def next_pass(self, count):
    """Process each subsequent profile. """
    self.order.append(self.profileList[count])
    for obj in self.profileList[count]:
        match = False
        if isinstance(obj, Objects.FileObject):
            for fi in self.dfxml_obj:
                if self.compare_files(fi, obj):
                    match = True
                    if fi.count is None:
                        obj.count = 1
                        self.dfxml_obj.append(obj)
                    else:
                        fi.count += 1
                    if match == False:
                        obj.count = 1
                        self.dfxml_obj.append(obj)
            elif isinstance(obj, Objects.CellObject):
                for cell in self.regxml_obj:
                    if self.compare_cells(cell, obj):
                        match = True
if cell.count is None:
    obj.count = 1
    self.regxml_obj.append(obj)
else:
    cell.count += 1
    if match == False:
        obj.count = 1
        self.regxml_obj.append(obj)

def compare_files(self, fi1, fi2):
    """ Compare all metadata properties between two FileObjects """
    if fi1.filename.endswith(".lnk"):
        # Compare ShortCut (lnk) files
        # Do not compare SHA-1 hash value
        return (fi1.filename == fi2.filename and
                fi1.meta_type == fi2.meta_type and
                fi1.alloc_inode == fi2.alloc_inode and
                fi1.alloc_name == fi2.alloc_name and
                fi1.annos == fi2.annos and
                fi1.app_state == fi2.app_state)
    elif fi1.filename.endswith(".pf"):
        # Normalize Prefetch file for comparison, e.g.,
        # Before: C:\Windows\Prefetch\TRUECRYPT.EXE-009A2E5A.pf
        # After: C:\Windows\Prefetch\TRUECRYPT.EXE
        path1 = os.path.splitext(fi1.filename)[0]
        path1 = path1.split("-") [0]
        path2 = os.path.splitext(fi2.filename)[0]
        path2 = path2.split("-") [0]
        return (path1 == path2 and
                fi1.meta_type == fi2.meta_type and
                fi1.alloc_inode == fi2.alloc_inode and
                fi1.alloc_name == fi2.alloc_name and
                fi1.annos == fi2.annos and
                fi1.app_state == fi2.app_state)
    else:
        return (fi1.filename == fi2.filename and
                fi1.meta_type == fi2.meta_type and
                fi1.sha1 == fi2.sha1 and
                fi1.alloc_inode == fi2.alloc_inode and
                fi1.alloc_name == fi2.alloc_name and
                fi1.annos == fi2.annos and
                fi1.app_state == fi2.app_state)

def compare_cells(self, co1, co2):
    """ Compare all metadata properties between two CellObjects """
    if "UserAssist" in co1.cellpath:
        return (co1.cellpath == co2.cellpath and
return (co1.cellpath == co2.cellpath and
        co1.name_type == co2.name_type and
        co1.alloc == co2.alloc and
        co1.data_type == co2.data_type and
        co1.annos == co2.annos and
        co1.app_state == co2.app_state)

def stats(self, count):
    fis = list(self.dfxml_obj)
    cos = list(self.regxml_obj)
    intersect_fis = [fi for fi in fis if fi.count == count + 1]
    intersect_cos = [co for co in cos if co.count == count + 1]

    cCOUNT = count + 1
    cALL = sum(1 for x in intersect_fis) + sum(1 for x in intersect_cos)
    cDIRS = len([fi for fi in intersect_fis if fi.meta_type == 2])
    cFILES = len([fi for fi in intersect_fis if fi.meta_type == 1])
    cKEYS = len([co for co in intersect_cos if co.name_type == "k"])
    cVALUES = len([co for co in intersect_cos if co.name_type == "v"])

    print("%s,%d,%d,%d,%s,%s \n" % (self.order[count].name,
        cDIRS,
        cFILES,
        cKEYS,
        '{: ,}'.format(cVALUES),
        '{: ,}'.format(cALL)))

def csv_output(self, count):
    """ Create CSV output for intersected entries. """
    count += 1
    files_csv = "n" + str(count) + "_FILES.csv"
    cells_csv = "n" + str(count) + "_CELLS.csv"
    # CSV for FILES
    with open(files_csv, 'w') as f:
        f.write("count , state , filename , delta , meta_type , alloc_name ,
                alloc_inode , filesize , sha1 \n")
        for fi in self.dfxml_obj:
            f.write("%s,%s,%s,%s,%s,%s,%s,%s \n" % (fi.count,
                fi.app_state,
                fi.filename,
                "".join(fi.annos),
                fi.meta_type,
                fi.alloc_name,
                fi.alloc_inode,
                fi.meta_type,
                fi.alloc_name,
                fi.alloc_inode,
                fi.meta_type,
                fi.alloc_name,
                fi.alloc_inode)
def apxml_output(self, count):
    """ Create APXML output for intersected entries. """

    apxml_out = self.profileList[0]

    # Remove all files and cells from APXMLObject
    del apxml_out._files[:]
    del apxml_out._cells[:]

    # Append files and cells to new APXML
    for fi in self.dfxml_obj:
        if fi.count == count:
            apxml_out.append(fi)
    for cell in self.regxml_obj:
        if cell.count == count:
            apxml_out.append(cell)

    # Write a temp APXML document
    temp_fi = io.StringIO(apxml_out.to_apxml())
    # Format APXML using minidom
    xml_fi = xml.dom.minidom.parse(temp_fi)
    apxml_report = xml_fi.toprettyxml(indent="  ")
    # Set the file output name
    fn = self.out_fn + "−n" + str(count) + "−INTERSECTION.apxml"
    # Write out APXML document
    with open(fn, "w", encoding="utf-16-le") as f:
        #f.write("<?xml version='1.0' encoding='UTF-16' ?>")
        f.write(apxml_report)
Listing C.2: Full source code listing for APXMLIntersection.py

C.4 Application Profile Intersection Results

Application profile intersection was tested to perform automated removal of irrelevant digital artifacts from the created application profiles. A total of 25 application profiles were created for each anti-forensic tool selected for testing. For each anti-forensic tool, profile intersection was conducted by performing set intersection of all 25 application profiles. The following tables display the count for each digital artifact type (directories, files, Registry keys and Registry values) and the total count of all digital artifacts. Table C.4, C.5 and C.6 display the resultant digital artifact counts from application profile intersection.
Table C.4: Application Profile Intersection Results for CCleaner version 5.09

<table>
<thead>
<tr>
<th>Profile</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC13</td>
<td>19</td>
<td>370</td>
<td>95</td>
<td>430</td>
<td>914</td>
</tr>
<tr>
<td>CC21</td>
<td>12</td>
<td>332</td>
<td>81</td>
<td>234</td>
<td>659</td>
</tr>
<tr>
<td>CC23</td>
<td>12</td>
<td>284</td>
<td>80</td>
<td>233</td>
<td>609</td>
</tr>
<tr>
<td>CC10</td>
<td>11</td>
<td>282</td>
<td>79</td>
<td>225</td>
<td>597</td>
</tr>
<tr>
<td>CC11</td>
<td>11</td>
<td>278</td>
<td>79</td>
<td>225</td>
<td>593</td>
</tr>
<tr>
<td>CC24</td>
<td>11</td>
<td>278</td>
<td>79</td>
<td>225</td>
<td>593</td>
</tr>
<tr>
<td>CC07</td>
<td>11</td>
<td>276</td>
<td>79</td>
<td>211</td>
<td>577</td>
</tr>
<tr>
<td>CC17</td>
<td>11</td>
<td>276</td>
<td>79</td>
<td>211</td>
<td>577</td>
</tr>
<tr>
<td>CC09</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>210</td>
<td>575</td>
</tr>
<tr>
<td>CC08</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>209</td>
<td>574</td>
</tr>
<tr>
<td>CC03</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>208</td>
<td>573</td>
</tr>
<tr>
<td>CC15</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>208</td>
<td>573</td>
</tr>
<tr>
<td>CC19</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>208</td>
<td>573</td>
</tr>
<tr>
<td>CC22</td>
<td>11</td>
<td>275</td>
<td>79</td>
<td>208</td>
<td>573</td>
</tr>
<tr>
<td>CC12</td>
<td>11</td>
<td>275</td>
<td>78</td>
<td>205</td>
<td>569</td>
</tr>
<tr>
<td>CC20</td>
<td>11</td>
<td>275</td>
<td>78</td>
<td>205</td>
<td>569</td>
</tr>
<tr>
<td>CC18</td>
<td>11</td>
<td>275</td>
<td>78</td>
<td>205</td>
<td>569</td>
</tr>
<tr>
<td>CC01</td>
<td>11</td>
<td>274</td>
<td>78</td>
<td>205</td>
<td>568</td>
</tr>
<tr>
<td>CC02</td>
<td>11</td>
<td>274</td>
<td>78</td>
<td>205</td>
<td>568</td>
</tr>
<tr>
<td>CC05</td>
<td>11</td>
<td>274</td>
<td>78</td>
<td>205</td>
<td>568</td>
</tr>
<tr>
<td>CC06</td>
<td>11</td>
<td>264</td>
<td>78</td>
<td>203</td>
<td>556</td>
</tr>
<tr>
<td>CC25</td>
<td>11</td>
<td>264</td>
<td>78</td>
<td>203</td>
<td>556</td>
</tr>
<tr>
<td>CC14</td>
<td>11</td>
<td>264</td>
<td>78</td>
<td>203</td>
<td>556</td>
</tr>
<tr>
<td>CC16</td>
<td>11</td>
<td>264</td>
<td>78</td>
<td>203</td>
<td>556</td>
</tr>
<tr>
<td>CC04</td>
<td>11</td>
<td>263</td>
<td>78</td>
<td>203</td>
<td>555</td>
</tr>
</tbody>
</table>
Table C.5: Application Profile Intersection Results for Eraser version 6.2.0.2970

<table>
<thead>
<tr>
<th>Profile</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER03</td>
<td>25</td>
<td>292</td>
<td>194</td>
<td>2,108</td>
<td>2,619</td>
</tr>
<tr>
<td>ER13</td>
<td>15</td>
<td>243</td>
<td>181</td>
<td>1,915</td>
<td>2,354</td>
</tr>
<tr>
<td>ER18</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,909</td>
<td>2,347</td>
</tr>
<tr>
<td>ER09</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,909</td>
<td>2,347</td>
</tr>
<tr>
<td>ER20</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,890</td>
<td>2,328</td>
</tr>
<tr>
<td>ER01</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,889</td>
<td>2,327</td>
</tr>
<tr>
<td>ER19</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,889</td>
<td>2,327</td>
</tr>
<tr>
<td>ER04</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,888</td>
<td>2,326</td>
</tr>
<tr>
<td>ER07</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,888</td>
<td>2,326</td>
</tr>
<tr>
<td>ER22</td>
<td>15</td>
<td>242</td>
<td>181</td>
<td>1,888</td>
<td>2,326</td>
</tr>
<tr>
<td>ER21</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER25</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER23</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER24</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER11</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER12</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER05</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER06</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER02</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER14</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER16</td>
<td>15</td>
<td>240</td>
<td>181</td>
<td>1,888</td>
<td>2,324</td>
</tr>
<tr>
<td>ER08</td>
<td>15</td>
<td>239</td>
<td>181</td>
<td>1,888</td>
<td>2,323</td>
</tr>
<tr>
<td>ER15</td>
<td>15</td>
<td>239</td>
<td>181</td>
<td>1,888</td>
<td>2,323</td>
</tr>
<tr>
<td>ER10</td>
<td>15</td>
<td>239</td>
<td>181</td>
<td>1,888</td>
<td>2,323</td>
</tr>
<tr>
<td>ER17</td>
<td>15</td>
<td>239</td>
<td>181</td>
<td>1,876</td>
<td>2,311</td>
</tr>
</tbody>
</table>
Table C.6: Application Profile Intersection Results for TrueCrypt version 7.1a

<table>
<thead>
<tr>
<th>Profile</th>
<th>Dirs</th>
<th>Files</th>
<th>Keys</th>
<th>Values</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC13</td>
<td>15</td>
<td>210</td>
<td>153</td>
<td>507</td>
<td>885</td>
</tr>
<tr>
<td>TC04</td>
<td>14</td>
<td>202</td>
<td>143</td>
<td>349</td>
<td>708</td>
</tr>
<tr>
<td>TC12</td>
<td>14</td>
<td>201</td>
<td>143</td>
<td>330</td>
<td>688</td>
</tr>
<tr>
<td>TC02</td>
<td>14</td>
<td>196</td>
<td>143</td>
<td>326</td>
<td>679</td>
</tr>
<tr>
<td>TC03</td>
<td>14</td>
<td>196</td>
<td>143</td>
<td>326</td>
<td>679</td>
</tr>
<tr>
<td>TC25</td>
<td>14</td>
<td>196</td>
<td>143</td>
<td>322</td>
<td>675</td>
</tr>
<tr>
<td>TC09</td>
<td>14</td>
<td>196</td>
<td>143</td>
<td>322</td>
<td>675</td>
</tr>
<tr>
<td>TC24</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC16</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC20</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC21</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC23</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC11</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC22</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC18</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC15</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC19</td>
<td>14</td>
<td>195</td>
<td>143</td>
<td>319</td>
<td>671</td>
</tr>
<tr>
<td>TC01</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC17</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC06</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC14</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC05</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC10</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC07</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
<tr>
<td>TC08</td>
<td>14</td>
<td>187</td>
<td>140</td>
<td>318</td>
<td>659</td>
</tr>
</tbody>
</table>
C.5 Application Profile Blacklisting: APXMLBlacklist.py

Listing C.3 displays the full source code for the APXMLBlacklist.py script.

```python
import os
import io
import glob
import copy
import collections
import xml.dom.minidom

try:
    import dfxml
except ImportError:
    print("Error: APXMLBlacklist.py")
    print(" The dfxml.py module is required to run this script")
    print(" Now Exiting...")
    sys.exit(1)

try:
    import Objects
except ImportError:
    print("Error: APXMLBlacklist.py")
    print(" The Objects.py module is required to run this script")
    print(" Now Exiting...")
    sys.exit(1)

try:
    import apxml
except ImportError:
    print("Error: APXMLBlacklist.py")
    print(" The apxml.py module is required to run this script")
    print(" Now Exiting...")
    sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# Blacklist object

class Blacklist(object):
    def __init__(self, profile, bFILES, bHKLM, bHKU, working_dir, output_dir):
        self.profile = profile
        self.basename = os.path.splitext(self.profile)[0]
        self.bFILES = bFILES
        self.bHKLM = bHKLM
        self.bHKU = bHKU
        self.working_dir = working_dir
        self.output_dir = output_dir

        self.filesBlacklist = list()
        self.cellsBlacklist = list()
```
def parse_blacklist(self):
    # Open and read each LiveDiff blacklist file.
    with open(self.bFILES, encoding="utf-16-le") as f:
        for line in f:
            line = line.strip()
            self.filesBlacklist.append(line)
    with open(self.bHKLM, encoding="utf-16-le") as f:
        for line in f:
            line = line.strip()
            self.cellsBlacklist.append(line)
    with open(self.bHKU, encoding="utf-16-le") as f:
        for line in f:
            line = line.strip()
            self.cellsBlacklist.append(line)

def parse_apxml(self):
    # Parse APXML report using apxml.py module.
    self.apxml_obj = apxml.iterparse(self.profile)
    apxml.generate_stats(self.apxml_obj)

def blacklist(self):
    # Perform blacklisting.
    for obj in self.apxml_obj:
        if isinstance(obj, Objects.FileObject):
            if obj.filename in self.filesBlacklist:
                obj.blacklisted = True
            else:
                obj.blacklisted = False
        if isinstance(obj, Objects.CellObject):
            if obj.cellpath in self.cellsBlacklist:
                obj.blacklisted = True
            else:
                obj.blacklisted = False

def apxml_output(self):
    # Reconstruct blacklisted APXML document.
    new_apxml_obj = copy.deepcopy(self.apxml_obj)
    del new_apxml_obj._files[:]
    del new_apxml_obj._cells[:]

    # Append files and cells to new APXML
    for obj in self.apxml_obj:
        if not obj.blacklisted:
            new_apxml_obj.append(obj)

    # Write a temp APXML document
    temp_fi = io.StringIO(new_apxml_obj.to_apxml())
# Format APXML using minidom
xml_fi = xml.dom.minidom.parse(temp_fi)
apxml_report = xml_fi.toprettyxml(indent=" ")

# Set the file output name
fn = self.output_dir + "/" + os.path.basename(self.basename)
fn = fn + "--blacklisted.apxml"

# Write out APXML document
with open(fn, "w", encoding="utf-16-le") as f:
    f.write(apxml_report)
f.close()

def csv_output(self):
    """ Create CSV for blacklisted entries. """
fn = self.output_dir + "/" + os.path.basename(self.basename)
files_csv = fn + "_FILES.csv"
cells_csv = fn + "_CELLS.csv"

# CSV for FILES
with open(files_csv, 'w') as f:
    f.write("blacklisted, filename, delta, meta_type, alloc_name, alloc_inode, filesize, sha1
")
    for fi in self.apxml_obj:
        if isinstance(fi, Objects.FileObject):
            if not fi.blacklisted:
                f.write("%s,%s,%s,%s,%s,%s,%s
" % ("NO", fi.filename,
                " ".join(fi.annos),
                fi.meta_type,
                fi.alloc_name,
                fi.alloc_inode,
                fi.filesize,
                fi.sha1))
            elif fi.blacklisted:
                f.write("%s,%s,%s,%s,%s,%s,%s
" % ("YES", fi.filename,
                " ".join(fi.annos),
                fi.meta_type,
                fi.alloc_name,
                fi.alloc_inode,
                fi.filesize,
                fi.sha1))

# CSV for CELLS
with open(cells_csv, 'w') as f:
    f.write("blacklisted, cellpath, delta, name_type, alloc, data_type, data
")
    for co in self.apxml_obj:
        if isinstance(co, Objects.CellObject):
            if not co.blacklisted:
                f.write("%s,%s,%s,%s,%s,%s,%s
" % ("NO", co.cellpath,
                " ".join(co.annos),
                co.name_type,
                co.alloc_name,
                co.alloc_inode,
                co.data_type,
                co.data))
            elif co.blacklisted:
                f.write("%s,%s,%s,%s,%s,%s,%s
" % ("YES", co.cellpath,
                " ".join(co.annos),
                co.name_type,
                co.alloc_name,
                co.alloc_inode,
                co.data_type,
                co.data))
elif co.blacklisted:
    f.write("%s,%s,%s,%s,%s,%s\n" % ("YES", co.cellpath, 
        "\n" + "\n".join (fi.annos), co.name_type, 
        co.alloc, 
        co.data_type, 
        co.data))

if __name__=='__main__':
    import argparse
    parser = argparse.ArgumentParser(description='''BlacklistSingle.py.''
                                    , formatter_class = argparse.RawTextHelpFormatter)
    parser.add_argument('livediff_output', help = 'LiveDiff output directory')
    parser.add_argument('output_dir', help = 'Output directory')
    args = parser.parse_args()
    working_dir = args.livediff_output
    output_dir = args.output_dir
    os.makedirs(output_dir)
    profile = glob.glob(working_dir + "/*.apxml") [0]
    bFILES = working_dir + "/blacklistFILES.txt"
    bHKLM = working_dir + "/blacklistHKLM.txt"
    bHKU = working_dir + "/blacklistHKU.txt"
    blacklisting = Blacklist(profile , bFILES, bHKLM, bHKU, working_dir, 
                              output_dir)
    blacklisting.parse_blacklist()
    blacklisting.parse_apxml()
    blacklisting.blacklist()
    blacklisting.apxml_output()
    blacklisting.csv_output()

Listing C.3: Full source code listing for APXMLBlacklist.py
C.6 LiveDiff Static Blacklist

Listing C.4 displays the full static blacklist implemented in the LiveDiff tool.

```plaintext
# LIVEDIFF STATIC BLACKLIST
# Any line starting with "#" is a comment
# The following prefixes are defined:
# DIR= Specify a directory and contents to be excluded
# FILE= Specify a file to be excluded
# KEY= Specify a Registry key, sub-key and values to be excluded.
# VALUE= Specify a Registry value to be excluded.

########## FILE SYSTEM ENTRIES
DIR=C:\ProgramData\Microsoft\Windows Defender
DIR=C:\ProgramData\Microsoft\RAC
DIR=C:\ProgramData\Microsoft\Search
DIR=C:\Users\All Users\Microsoft\RAC
DIR=C:\Users\All Users\Microsoft\Search
DIR=C:\Users\All Users\Microsoft\Windows Defender
DIR=C:\Users\forensic\AppData\Local\Microsoft\Internet Explorer
DIR=C:\Users\forensic\AppData\Local\Microsoft\Windows\History\History .IE5
DIR=C:\Users\forensic\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content .IE5
DIR=C:\Users\forensic\AppData\LocalLow\Microsoft\CryptnetUrlCache
DIR=C:\Users\forensic\AppData\Roaming\Microsoft\Crypto
DIR=C:\Users\forensic\AppData\Roaming\Microsoft\Windows\Recent\CustomDestinations
DIR=C:\Windows\SoftwareDistribution
DIR=C:\Windows\System32\wdi
DIR=C:\Windows\System32\winevt\Logs
DIR=C:\Windows\Logs
DIR=C:\Windows\System32\LogFiles
DIR=C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache

########## WINDOWS REGISTRY ENTRIES
KEY=HKLM\SOFTWARE\Microsoft\Security Center
KEY=HKLM\SOFTWARE\Microsoft\SystemCertificates\AuthRoot\Certificates
KEY=HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData
KEY=HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy
KEY=HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate
KEY=HKLM\SOFTWARE\Microsoft\Windows\Search\Gather\Windows\SystemIndex
KEY=HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers
KEY=HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule
KEY=HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SystemRestore\Volatile
KEY=HKLM\SYSTEM\ControlSet001
KEY=HKLM\SYSTEM\ControlSet002
KEY=HKLM\SYSTEM\CurrentControlSet\Control\DeviceClasses
KEY=HKLM\SYSTEM\CurrentControlSet\Enum\STORAGE
KEY=HKLM\SYSTEM\CurrentControlSet\Control\Class
```
Listing C.4: Full listing of the LiveDiff static blacklist

44  KEY=HKLM\SYSTEM\CurrentControlSet\Control\Session Manager
45  KEY=HKLM\SYSTEM\CurrentControlSet\services\VSS
46  KEY=HKU\.DEFAULT\Software\Classes\Local Settings\MuiCache
47  KEY=HKU\S-1-5-18\Software\Classes\Local Settings\MuiCache
48  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Classes\Local Settings\MuiCache
49  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Classes\Local Settings\Software\Microsoft\Windows\Shell
50  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExt
51  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\Explorer\Main\WindowsSearch
52  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\StartPage2
53  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Microsoft\Windows\CurrentVersion\Internet Settings
54  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Cached
55  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Microsoft\Windows\CurrentVersion\Shell Extensions\Cached
56  KEY=HKU\S-1-5-21-884162780-2507285075-3001024006-1000\Software\Microsoft\Windows\CurrentVersion\Action Center

341
C.7 Final Application Profiles

This section outlines and presents additional information and data regarding the final versions of the application profiles that were created for testing against the known data set, public data set (M57-Patents scenario) and the real-world data set.

Listing C.5 displays the full source code used to extract statistics from the finalised application profiles.

```python
#!/usr/bin/env python3

import os

try:
    import dfxml
except ImportError:
    print("Error : APXMLPrintProfile.py")
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
    sys.exit(1)

try:
    import Objects
except ImportError:
    print("Error : APXMLPrintProfile.py")
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
    sys.exit(1)

try:
    import apxml
except ImportError:
    print("Error : APXMLPrintProfile.py")
    print("The apxml.py module is required to run this script")
    print("Now Exiting...")
    sys.exit(1)

if __name__=='__main__':
    import argparse
    parser = argparse.ArgumentParser(description="'APXMLPrintProfile.py'.", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument('profile', help='profile')
    args = parser.parse_args()
    fi = args.profile
    apxml_obj = apxml.iterparse(fi)
    apxml.generate_stats(apxml_obj)
```
print("State,Type,New,Modified,Deleted,Total")
deltas = ["new", "modified", "deleted"]
states = ["install", "open", "close", "uninstall"]
for state in states:
    print("%s,%s,%d,%d,%d,%d" % (state,
        "Dirs",
        apxml_obj.stats._dirs[state].count("new"),
        apxml_obj.stats._dirs[state].count("modified"),
        apxml_obj.stats._dirs[state].count("deleted"),
        len(apxml_obj.stats._dirs[state])))
    print("%s,%s,%d,%d,%d,%d" % (state,
        "Files",
        apxml_obj.stats._files[state].count("new"),
        apxml_obj.stats._files[state].count("modified"),
        apxml_obj.stats._files[state].count("deleted"),
        len(apxml_obj.stats._files[state])))
    print("%s,%s,%d,%d,%d,%d" % (state,
        "Keys",
        apxml_obj.stats._keys[state].count("new"),
        apxml_obj.stats._keys[state].count("modified"),
        apxml_obj.stats._keys[state].count("deleted"),
        len(apxml_obj.stats._keys[state])))
    print("%s,%s,%d,%d,%d,%d" % (state,
        "Values",
        apxml_obj.stats._values[state].count("new"),
        apxml_obj.stats._values[state].count("modified"),
        apxml_obj.stats._values[state].count("deleted"),
        len(apxml_obj.stats._values[state])))

Listing C.5: Full listing of the APXMLPrintProfile.py script

Tables C.7, C.8, C.9 and C.10 provides a count of all application profile contents based on digital artifact type (e.g., directory), application state (e.g., install), delta properties (e.g., new) and a total count.
Table C.7: Summary of the final application profile for **CCleaner** version 5.09

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>New</th>
<th>Modified</th>
<th>Deleted</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>Dirs</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>install</td>
<td>Files</td>
<td>62</td>
<td>0</td>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td>install</td>
<td>Keys</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>install</td>
<td>Values</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>open</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>open</td>
<td>Keys</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>open</td>
<td>Values</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>close</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Values</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>uninstall</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>uninstall</td>
<td>Files</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>uninstall</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>uninstall</td>
<td>Values</td>
<td>1</td>
<td>0</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>139</td>
<td>0</td>
<td>116</td>
<td>257</td>
</tr>
</tbody>
</table>

Table C.8: Summary of the final application profile for **Eraser** version 6.2.0

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>New</th>
<th>Modified</th>
<th>Deleted</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>Dirs</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>install</td>
<td>Files</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>install</td>
<td>Keys</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>install</td>
<td>Values</td>
<td>93</td>
<td>0</td>
<td>0</td>
<td>93</td>
</tr>
<tr>
<td>open</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>open</td>
<td>Keys</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>open</td>
<td>Values</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>close</td>
<td>Dirs</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>close</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>close</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Values</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uninstall</td>
<td>Dirs</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>uninstall</td>
<td>Files</td>
<td>0</td>
<td>0</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>uninstall</td>
<td>Keys</td>
<td>2</td>
<td>0</td>
<td>34</td>
<td>36</td>
</tr>
<tr>
<td>uninstall</td>
<td>Values</td>
<td>0</td>
<td>0</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>186</td>
<td>0</td>
<td>164</td>
<td>350</td>
</tr>
</tbody>
</table>

344
Table C.9: Summary of the final application profile for TrueCrypt version 7.1a

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>New</th>
<th>Modified</th>
<th>Deleted</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>Dirs</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>install</td>
<td>Files</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>install</td>
<td>Keys</td>
<td>58</td>
<td>0</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>install</td>
<td>Values</td>
<td>84</td>
<td>0</td>
<td>0</td>
<td>84</td>
</tr>
<tr>
<td>open</td>
<td>Dirs</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>open</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>open</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Values</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>close</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>close</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Values</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>uninstall</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>uninstall</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>uninstall</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>uninstall</td>
<td>Values</td>
<td>1</td>
<td>0</td>
<td>81</td>
<td>82</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>165</td>
<td>0</td>
<td>155</td>
<td>321</td>
</tr>
</tbody>
</table>

Table C.10: Summary of the final application profile for XP Keylogger version 2.1

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>New</th>
<th>Modified</th>
<th>Deleted</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>Dirs</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>install</td>
<td>Files</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>install</td>
<td>Keys</td>
<td>295</td>
<td>0</td>
<td>0</td>
<td>295</td>
</tr>
<tr>
<td>install</td>
<td>Values</td>
<td>350</td>
<td>0</td>
<td>0</td>
<td>350</td>
</tr>
<tr>
<td>open</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>Values</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>close</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>close</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>Values</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uninstall</td>
<td>Dirs</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>uninstall</td>
<td>Files</td>
<td>1</td>
<td>0</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>uninstall</td>
<td>Keys</td>
<td>0</td>
<td>0</td>
<td>279</td>
<td>279</td>
</tr>
<tr>
<td>uninstall</td>
<td>Values</td>
<td>1</td>
<td>0</td>
<td>334</td>
<td>335</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>681</td>
<td>0</td>
<td>640</td>
<td>1,321</td>
</tr>
</tbody>
</table>
C.8 Determining Data Set Statistics: FileSystemStats.py

Listing C.6 displays the full source code for the FileSystemStats.py script.

```python
import os
import sys

try:
    import dfxml
except ImportError:
    print("Error: FileSystemStats.py")
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
    sys.exit(1)

try:
    import Objects
except ImportError:
    print("Error: FileSystemStats.py")
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
    sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# FSStats object
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class FSStats(object):
    def __init__(self):
        self.all = 0
        self.dirs = 0
        self.files = 0
        self.other = 0
        self.alloc = 0
        self.unalloc = 0
        self.target_dir_count = 0
        self.target_file_count = 0

    def generate_stats(self, fi):
        if fi.meta_type == 2:
            self.target_dir_count += 1
        elif fi.meta_type == 1:
            self.target_file_count += 1
        self.all += 1

        if fi.meta_type == 1:
            self.files += 1
        elif fi.meta_type == 2:
            self.dirs += 1
        else:
            self.other += 1
```

346
if fi.is_allocated():
    self.alloc += 1
else:
    self.unalloc += 1

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
if __name__=='__main__':
    import argparse
    parser = argparse.ArgumentParser(description= '''FileSystemStats.py''', formatter_class = argparse.RawTextHelpFormatter)
    parser.add_argument('target', help = 'An imagefile or DFXML report')
    args = parser.parse_args()
    stats = FSStats()
    for (event, obj) in Objects.iterparse(args.xmlfile):
        if isinstance(obj, Objects.FileObject):
            stats.generate_stats(obj)

    print("%s\t%d\t%d\t%d\t%d\t%d\t%d" % (os.path.basename(args.xmlfile),
                    stats.all,
                    stats.dirs,
                    stats.files,
                    stats.other,
                    stats.alloc,
                    stats.unalloc))

Listing C.6: Full source code listing for FileSystemStats.py
C.9 Determining Data Set Statistics: RegistryStats.py

Listing C.7 displays the full source code for the RegistryStats.py script.

```python
import os
import sys
import glob

try:
    import dfxml
except ImportError:
    print("Error: RegistryStats.py")
    print("The dfxml.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

try:
    import Objects
except ImportError:
    print("Error: RegistryStats.py")
    print("The Objects.py module is required to run this script")
    print("Now Exiting...")
sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# REGStats object

class REGStats(object):
    def __init__(self):
        self.all = 0
        self.keys = 0
        self.values = 0
        self.other = 0
        self.alloc = 0
        self.unalloc = 0
        self.target_key_count = 0
        self.target_value_count = 0

    def generate_stats(self, co):
        if co.name_type == "k":
            self.target_key_count += 1
        elif co.name_type == "v":
            self.target_value_count += 1

        self.all += 1

        if co.name_type == "k":
            self.keys += 1
        elif co.name_type == "v":
            self.values += 1
else:
```

348
```python
    self.other += 1

    if co.alloc == 1:
        self.alloc += 1
    else:
        self.unalloc += 1

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

if __name__=='__main__':
    import argparse
    parser = argparse.ArgumentParser(description='''RegistryStats.py''',
                                     formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument('xmlfile_dir',
                        help='Directory of RegXML reports')
    args = parser.parse_args()
    stats = REGStats()
    hive_count = 0

    hives = glob.glob(args.xmlfile_dir + '/*.xml')
    #print(hives)

    for hive in hives:
        hive_count += 1
        for (event, obj) in Objects.iterparse_CellObjects(hive):
            if isinstance(obj, Objects.CellObject):
                stats.generate_stats(obj)

    print('%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d' % (args.xmlfile_dir,
                                             hive_count,
                                             stats.all,
                                             stats.keys,
                                             stats.values,
                                             stats.other,
                                             stats.alloc,
                                             stats.unalloc))
```

Listing C.7: Full source code listing for RegistryStats.py
C.10 Known Data Set Contents

Table C.11 displays the complete file system counts for the known data set including the number of directories (dirs), files, other file system data types, allocated entries, unallocated entries and a total count.

Table C.11: Overview of file system counts for the known data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Dirs</th>
<th>Files</th>
<th>Other</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT-01</td>
<td>28,713</td>
<td>48,747</td>
<td>3,091</td>
<td>77,062</td>
<td>3,489</td>
<td>80,551</td>
</tr>
<tr>
<td>CC-01</td>
<td>28,829</td>
<td>48,947</td>
<td>2,918</td>
<td>77,513</td>
<td>3,181</td>
<td>80,694</td>
</tr>
<tr>
<td>CC-02</td>
<td>28,827</td>
<td>48,951</td>
<td>2,917</td>
<td>77,530</td>
<td>3,165</td>
<td>80,695</td>
</tr>
<tr>
<td>CC-03</td>
<td>28,827</td>
<td>48,958</td>
<td>2,917</td>
<td>77,546</td>
<td>3,156</td>
<td>80,702</td>
</tr>
<tr>
<td>CC-04</td>
<td>28,827</td>
<td>48,923</td>
<td>2,918</td>
<td>77,444</td>
<td>3,224</td>
<td>80,668</td>
</tr>
<tr>
<td>ER-01</td>
<td>30,644</td>
<td>49,981</td>
<td>2,910</td>
<td>80,451</td>
<td>3,084</td>
<td>83,535</td>
</tr>
<tr>
<td>ER-02</td>
<td>30,630</td>
<td>49,992</td>
<td>2,909</td>
<td>80,445</td>
<td>3,086</td>
<td>83,531</td>
</tr>
<tr>
<td>ER-03</td>
<td>30,642</td>
<td>50,017</td>
<td>2,909</td>
<td>80,534</td>
<td>3,034</td>
<td>83,568</td>
</tr>
<tr>
<td>ER-04</td>
<td>30,637</td>
<td>50,019</td>
<td>2,916</td>
<td>80,467</td>
<td>3,105</td>
<td>83,572</td>
</tr>
<tr>
<td>TC-01</td>
<td>29,044</td>
<td>49,134</td>
<td>2,930</td>
<td>78,170</td>
<td>2,938</td>
<td>81,108</td>
</tr>
<tr>
<td>TC-02</td>
<td>29,047</td>
<td>49,136</td>
<td>2,929</td>
<td>78,179</td>
<td>2,933</td>
<td>81,112</td>
</tr>
<tr>
<td>TC-03</td>
<td>29,047</td>
<td>49,138</td>
<td>2,930</td>
<td>78,178</td>
<td>2,937</td>
<td>81,115</td>
</tr>
<tr>
<td>TC-04</td>
<td>29,046</td>
<td>49,139</td>
<td>2,934</td>
<td>78,160</td>
<td>2,959</td>
<td>81,119</td>
</tr>
<tr>
<td>Total</td>
<td>382,760</td>
<td>641,082</td>
<td>38,128</td>
<td>1,021,679</td>
<td>40,291</td>
<td>1,061,970</td>
</tr>
<tr>
<td>Average</td>
<td>29,443.08</td>
<td>49,314.00</td>
<td>2,932.92</td>
<td>78,590.69</td>
<td>3,099.31</td>
<td>81,690.00</td>
</tr>
</tbody>
</table>

Table C.12 displays the complete Registry count for the known data set including number of hive files, keys, values, allocated entries, unallocated entries and a total count.

Table C.12: Overview of Registry counts for the known data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT-01</td>
<td>9</td>
<td>114,618</td>
<td>207,452</td>
<td>310,532</td>
<td>11,538</td>
<td>322,070</td>
</tr>
<tr>
<td>CC-01</td>
<td>9</td>
<td>114,796</td>
<td>206,934</td>
<td>311,344</td>
<td>10,386</td>
<td>321,730</td>
</tr>
<tr>
<td>CC-02</td>
<td>9</td>
<td>114,796</td>
<td>206,934</td>
<td>311,344</td>
<td>10,386</td>
<td>321,730</td>
</tr>
<tr>
<td>CC-03</td>
<td>9</td>
<td>114,796</td>
<td>206,934</td>
<td>311,344</td>
<td>10,386</td>
<td>321,730</td>
</tr>
<tr>
<td>CC-04</td>
<td>9</td>
<td>114,782</td>
<td>206,886</td>
<td>311,357</td>
<td>10,311</td>
<td>321,668</td>
</tr>
<tr>
<td>ER-01</td>
<td>9</td>
<td>115,290</td>
<td>204,400</td>
<td>317,725</td>
<td>1,965</td>
<td>319,690</td>
</tr>
<tr>
<td>ER-02</td>
<td>9</td>
<td>115,291</td>
<td>204,403</td>
<td>317,737</td>
<td>1,957</td>
<td>319,694</td>
</tr>
<tr>
<td>ER-03</td>
<td>9</td>
<td>115,292</td>
<td>204,406</td>
<td>317,741</td>
<td>1,957</td>
<td>319,698</td>
</tr>
<tr>
<td>ER-04</td>
<td>9</td>
<td>115,359</td>
<td>204,434</td>
<td>317,551</td>
<td>2,242</td>
<td>319,793</td>
</tr>
<tr>
<td>TC-01</td>
<td>9</td>
<td>127,541</td>
<td>221,375</td>
<td>337,943</td>
<td>10,973</td>
<td>348,916</td>
</tr>
<tr>
<td>TC-02</td>
<td>9</td>
<td>127,542</td>
<td>221,379</td>
<td>337,947</td>
<td>10,974</td>
<td>348,921</td>
</tr>
<tr>
<td>TC-03</td>
<td>9</td>
<td>127,542</td>
<td>221,469</td>
<td>337,965</td>
<td>11,046</td>
<td>349,011</td>
</tr>
<tr>
<td>TC-04</td>
<td>9</td>
<td>127,587</td>
<td>221,562</td>
<td>337,829</td>
<td>11,320</td>
<td>349,149</td>
</tr>
<tr>
<td>Total</td>
<td>117</td>
<td>1,545,232</td>
<td>2,738,568</td>
<td>4,178,359</td>
<td>105,441</td>
<td>4,283,800</td>
</tr>
<tr>
<td>Average</td>
<td>9.00</td>
<td>118,864.00</td>
<td>210,659.08</td>
<td>321,412.23</td>
<td>8,110.85</td>
<td>329,523.08</td>
</tr>
</tbody>
</table>
Appendix D provides additional relevant documentation regarding the system evaluation presented in Chapter 7. The following list specifies all included material:

1. Section D.1 documents the digital artifact counts for the M57-Patents scenario data set
2. Section D.2 provides the Python scripts used to determine the effectiveness of path normalisation for file system and Registry entries
3. Section D.3 documents the method used and results achieved to determine hard drive read speed
4. Section D.4 documents modifications made to the fiwalk tool and cross-compilation notes for compiling Windows binaries
5. Section D.5 documents all second-hand hard drives sources to compile the real-world data set
6. Section D.6 documents the digital artifact counts for the real-world data set

D.1 M57-Patents Scenario Data Set Contents

Table D.1 displays the complete file system counts for the M57-Patents scenario data set including the number of directories (dirs), files, other file system data types, allocated entries, unallocated entries and a total count.
<table>
<thead>
<tr>
<th>Name</th>
<th>Dirs</th>
<th>Files</th>
<th>Other</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>charlie-2009-11-17</td>
<td>7,688</td>
<td>26,699</td>
<td>561</td>
<td>34,373</td>
<td>575</td>
<td>34,948</td>
</tr>
<tr>
<td>charlie-2009-11-18</td>
<td>7,769</td>
<td>27,031</td>
<td>566</td>
<td>34,785</td>
<td>581</td>
<td>35,366</td>
</tr>
<tr>
<td>charlie-2009-11-19</td>
<td>9,656</td>
<td>28,383</td>
<td>596</td>
<td>38,028</td>
<td>607</td>
<td>38,635</td>
</tr>
<tr>
<td>charlie-2009-11-20</td>
<td>9,330</td>
<td>28,467</td>
<td>608</td>
<td>36,380</td>
<td>2,025</td>
<td>38,405</td>
</tr>
<tr>
<td>charlie-2009-11-23</td>
<td>9,265</td>
<td>28,525</td>
<td>617</td>
<td>36,364</td>
<td>2,043</td>
<td>38,407</td>
</tr>
<tr>
<td>charlie-2009-11-24</td>
<td>9,155</td>
<td>28,559</td>
<td>617</td>
<td>36,783</td>
<td>1,548</td>
<td>38,331</td>
</tr>
<tr>
<td>charlie-2009-11-30</td>
<td>9,091</td>
<td>28,642</td>
<td>659</td>
<td>37,580</td>
<td>812</td>
<td>38,392</td>
</tr>
<tr>
<td>charlie-2009-12-01</td>
<td>9,091</td>
<td>28,811</td>
<td>672</td>
<td>37,891</td>
<td>683</td>
<td>38,574</td>
</tr>
<tr>
<td>charlie-2009-12-02</td>
<td>9,125</td>
<td>29,006</td>
<td>665</td>
<td>37,753</td>
<td>1,043</td>
<td>38,796</td>
</tr>
<tr>
<td>charlie-2009-12-03</td>
<td>9,099</td>
<td>29,997</td>
<td>663</td>
<td>37,942</td>
<td>817</td>
<td>38,759</td>
</tr>
<tr>
<td>charlie-2009-12-04</td>
<td>9,103</td>
<td>29,110</td>
<td>672</td>
<td>38,204</td>
<td>681</td>
<td>38,885</td>
</tr>
<tr>
<td>charlie-2009-12-07</td>
<td>9,130</td>
<td>29,352</td>
<td>673</td>
<td>38,459</td>
<td>696</td>
<td>39,155</td>
</tr>
<tr>
<td>charlie-2009-12-08</td>
<td>9,123</td>
<td>29,469</td>
<td>684</td>
<td>38,420</td>
<td>856</td>
<td>39,276</td>
</tr>
<tr>
<td>charlie-2009-12-09</td>
<td>9,285</td>
<td>29,898</td>
<td>690</td>
<td>39,175</td>
<td>698</td>
<td>39,873</td>
</tr>
<tr>
<td>charlie-2009-12-10</td>
<td>9,304</td>
<td>30,016</td>
<td>696</td>
<td>39,314</td>
<td>702</td>
<td>40,016</td>
</tr>
<tr>
<td>charlie-2009-12-11</td>
<td>9,320</td>
<td>30,055</td>
<td>699</td>
<td>39,373</td>
<td>701</td>
<td>40,074</td>
</tr>
<tr>
<td>jo-2009-11-12start</td>
<td>6,808</td>
<td>22,324</td>
<td>410</td>
<td>29,128</td>
<td>414</td>
<td>29,542</td>
</tr>
<tr>
<td>jo-2009-11-12</td>
<td>7,357</td>
<td>26,091</td>
<td>502</td>
<td>33,294</td>
<td>656</td>
<td>33,950</td>
</tr>
<tr>
<td>jo-2009-11-16</td>
<td>7,619</td>
<td>26,848</td>
<td>565</td>
<td>34,433</td>
<td>599</td>
<td>35,032</td>
</tr>
<tr>
<td>jo-2009-11-17</td>
<td>7,649</td>
<td>27,121</td>
<td>556</td>
<td>34,767</td>
<td>559</td>
<td>35,326</td>
</tr>
<tr>
<td>jo-2009-11-18</td>
<td>7,668</td>
<td>27,567</td>
<td>563</td>
<td>35,183</td>
<td>615</td>
<td>35,798</td>
</tr>
<tr>
<td>jo-2009-11-19</td>
<td>7,686</td>
<td>27,706</td>
<td>565</td>
<td>35,386</td>
<td>571</td>
<td>35,957</td>
</tr>
<tr>
<td>jo-2009-11-20-newComputer</td>
<td>6,563</td>
<td>22,012</td>
<td>376</td>
<td>28,570</td>
<td>381</td>
<td>28,951</td>
</tr>
<tr>
<td>jo-2009-11-20-oldComputer</td>
<td>8,926</td>
<td>28,106</td>
<td>617</td>
<td>37,010</td>
<td>639</td>
<td>37,649</td>
</tr>
<tr>
<td>jo-2009-11-23</td>
<td>7,031</td>
<td>25,702</td>
<td>477</td>
<td>32,727</td>
<td>483</td>
<td>33,210</td>
</tr>
<tr>
<td>jo-2009-11-24</td>
<td>8,948</td>
<td>27,131</td>
<td>543</td>
<td>36,073</td>
<td>549</td>
<td>36,622</td>
</tr>
<tr>
<td>jo-2009-11-30</td>
<td>9,504</td>
<td>29,620</td>
<td>598</td>
<td>39,091</td>
<td>631</td>
<td>39,722</td>
</tr>
<tr>
<td>jo-2009-12-01</td>
<td>9,060</td>
<td>29,815</td>
<td>613</td>
<td>37,055</td>
<td>2,433</td>
<td>39,488</td>
</tr>
<tr>
<td>jo-2009-12-02</td>
<td>8,841</td>
<td>29,887</td>
<td>612</td>
<td>37,199</td>
<td>2,141</td>
<td>39,340</td>
</tr>
<tr>
<td>jo-2009-12-03</td>
<td>8,838</td>
<td>29,967</td>
<td>606</td>
<td>37,189</td>
<td>2,222</td>
<td>39,411</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>8,809</td>
<td>29,915</td>
<td>622</td>
<td>37,318</td>
<td>2,028</td>
<td>39,346</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>8,831</td>
<td>29,881</td>
<td>624</td>
<td>37,432</td>
<td>1,904</td>
<td>39,336</td>
</tr>
<tr>
<td>jo-2009-12-08</td>
<td>8,954</td>
<td>29,823</td>
<td>629</td>
<td>37,862</td>
<td>1,544</td>
<td>39,406</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>8,951</td>
<td>29,716</td>
<td>631</td>
<td>38,432</td>
<td>866</td>
<td>39,298</td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>9,017</td>
<td>31,519</td>
<td>797</td>
<td>40,532</td>
<td>801</td>
<td>41,333</td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>9,028</td>
<td>31,524</td>
<td>811</td>
<td>40,537</td>
<td>1,006</td>
<td>41,363</td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>9,041</td>
<td>31,575</td>
<td>814</td>
<td>40,501</td>
<td>929</td>
<td>41,430</td>
</tr>
<tr>
<td>pat-2009-11-12start</td>
<td>6,687</td>
<td>22,007</td>
<td>429</td>
<td>28,604</td>
<td>519</td>
<td>29,123</td>
</tr>
<tr>
<td>pat-2009-11-12</td>
<td>7,369</td>
<td>25,920</td>
<td>509</td>
<td>33,287</td>
<td>511</td>
<td>33,798</td>
</tr>
<tr>
<td>Name</td>
<td>Dirs</td>
<td>Files</td>
<td>Other</td>
<td>Alloc</td>
<td>Unalloc</td>
<td>Total</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>--------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>pat-2009-11-16</td>
<td>7,679</td>
<td>31,036</td>
<td>624</td>
<td>38,671</td>
<td>668</td>
<td>39,339</td>
</tr>
<tr>
<td>pat-2009-11-17</td>
<td>7,713</td>
<td>31,564</td>
<td>618</td>
<td>39,269</td>
<td>626</td>
<td>39,895</td>
</tr>
<tr>
<td>pat-2009-11-18</td>
<td>7,749</td>
<td>32,581</td>
<td>642</td>
<td>40,260</td>
<td>712</td>
<td>40,972</td>
</tr>
<tr>
<td>pat-2009-11-19</td>
<td>8,107</td>
<td>33,597</td>
<td>653</td>
<td>41,684</td>
<td>673</td>
<td>42,357</td>
</tr>
<tr>
<td>pat-2009-11-20</td>
<td>7,668</td>
<td>33,721</td>
<td>670</td>
<td>40,238</td>
<td>1,821</td>
<td>42,059</td>
</tr>
<tr>
<td>pat-2009-11-23</td>
<td>7,335</td>
<td>34,003</td>
<td>703</td>
<td>41,306</td>
<td>735</td>
<td>42,041</td>
</tr>
<tr>
<td>pat-2009-11-24</td>
<td>7,384</td>
<td>34,231</td>
<td>706</td>
<td>41,599</td>
<td>722</td>
<td>42,321</td>
</tr>
<tr>
<td>pat-2009-11-30</td>
<td>7,693</td>
<td>36,381</td>
<td>772</td>
<td>43,936</td>
<td>910</td>
<td>44,846</td>
</tr>
<tr>
<td>pat-2009-12-01</td>
<td>7,683</td>
<td>36,445</td>
<td>773</td>
<td>44,126</td>
<td>775</td>
<td>44,901</td>
</tr>
<tr>
<td>pat-2009-12-02</td>
<td>7,717</td>
<td>36,659</td>
<td>791</td>
<td>44,373</td>
<td>794</td>
<td>45,167</td>
</tr>
<tr>
<td>pat-2009-12-03</td>
<td>7,694</td>
<td>36,936</td>
<td>794</td>
<td>43,499</td>
<td>1,925</td>
<td>45,424</td>
</tr>
<tr>
<td>pat-2009-12-04</td>
<td>7,604</td>
<td>36,941</td>
<td>802</td>
<td>44,542</td>
<td>805</td>
<td>45,347</td>
</tr>
<tr>
<td>pat-2009-12-07</td>
<td>7,838</td>
<td>41,011</td>
<td>745</td>
<td>44,219</td>
<td>5,375</td>
<td>49,594</td>
</tr>
<tr>
<td>pat-2009-12-08</td>
<td>7,869</td>
<td>41,008</td>
<td>744</td>
<td>44,397</td>
<td>5,224</td>
<td>49,621</td>
</tr>
<tr>
<td>pat-2009-12-09</td>
<td>8,065</td>
<td>40,990</td>
<td>758</td>
<td>44,881</td>
<td>4,932</td>
<td>49,813</td>
</tr>
<tr>
<td>pat-2009-12-10</td>
<td>8,029</td>
<td>40,965</td>
<td>754</td>
<td>45,152</td>
<td>4,596</td>
<td>49,748</td>
</tr>
<tr>
<td>pat-2009-12-11</td>
<td>8,046</td>
<td>40,960</td>
<td>756</td>
<td>45,247</td>
<td>4,515</td>
<td>49,762</td>
</tr>
<tr>
<td>terry-2009-11-12start</td>
<td>38,398</td>
<td>78,049</td>
<td>4,492</td>
<td>116,374</td>
<td>4,565</td>
<td>120,939</td>
</tr>
<tr>
<td>terry-2009-11-12</td>
<td>48,838</td>
<td>97,708</td>
<td>5,149</td>
<td>141,215</td>
<td>10,480</td>
<td>151,695</td>
</tr>
<tr>
<td>terry-2009-11-16</td>
<td>49,114</td>
<td>97,747</td>
<td>5,247</td>
<td>142,200</td>
<td>9,908</td>
<td>152,108</td>
</tr>
<tr>
<td>terry-2009-11-17</td>
<td>49,298</td>
<td>97,692</td>
<td>5,262</td>
<td>143,037</td>
<td>9,215</td>
<td>152,252</td>
</tr>
<tr>
<td>terry-2009-11-18</td>
<td>48,897</td>
<td>97,598</td>
<td>5,355</td>
<td>144,035</td>
<td>7,815</td>
<td>151,850</td>
</tr>
<tr>
<td>terry-2009-11-19</td>
<td>38,886</td>
<td>81,460</td>
<td>4,598</td>
<td>120,326</td>
<td>4,618</td>
<td>124,944</td>
</tr>
<tr>
<td>terry-2009-11-20</td>
<td>39,332</td>
<td>82,439</td>
<td>4,608</td>
<td>121,702</td>
<td>4,677</td>
<td>126,379</td>
</tr>
<tr>
<td>terry-2009-11-23</td>
<td>46,301</td>
<td>96,221</td>
<td>4,876</td>
<td>136,437</td>
<td>10,961</td>
<td>147,398</td>
</tr>
<tr>
<td>terry-2009-11-24</td>
<td>49,408</td>
<td>96,291</td>
<td>5,242</td>
<td>143,046</td>
<td>7,895</td>
<td>150,941</td>
</tr>
<tr>
<td>terry-2009-11-30</td>
<td>46,230</td>
<td>99,006</td>
<td>4,004</td>
<td>132,312</td>
<td>16,928</td>
<td>149,240</td>
</tr>
<tr>
<td>terry-2009-12-01</td>
<td>46,201</td>
<td>99,037</td>
<td>4,035</td>
<td>132,301</td>
<td>16,972</td>
<td>149,273</td>
</tr>
<tr>
<td>terry-2009-12-02</td>
<td>46,202</td>
<td>99,037</td>
<td>4,019</td>
<td>132,303</td>
<td>16,955</td>
<td>149,258</td>
</tr>
<tr>
<td>terry-2009-12-03</td>
<td>46,122</td>
<td>99,118</td>
<td>4,081</td>
<td>132,646</td>
<td>16,675</td>
<td>149,321</td>
</tr>
<tr>
<td>terry-2009-12-04</td>
<td>46,137</td>
<td>99,116</td>
<td>4,103</td>
<td>132,692</td>
<td>16,664</td>
<td>149,356</td>
</tr>
<tr>
<td>terry-2009-12-07</td>
<td>45,687</td>
<td>99,228</td>
<td>3,743</td>
<td>129,030</td>
<td>19,628</td>
<td>148,658</td>
</tr>
<tr>
<td>terry-2009-12-08</td>
<td>45,767</td>
<td>99,220</td>
<td>3,735</td>
<td>129,303</td>
<td>19,419</td>
<td>148,722</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>46,063</td>
<td>99,242</td>
<td>3,733</td>
<td>130,135</td>
<td>18,903</td>
<td>149,038</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>46,143</td>
<td>99,066</td>
<td>3,662</td>
<td>130,353</td>
<td>18,518</td>
<td>148,871</td>
</tr>
<tr>
<td>terry-2009-12-11001</td>
<td>46,161</td>
<td>99,034</td>
<td>3,689</td>
<td>130,297</td>
<td>18,587</td>
<td>148,884</td>
</tr>
<tr>
<td>terry-2009-12-11002</td>
<td>46,223</td>
<td>98,517</td>
<td>3,409</td>
<td>125,403</td>
<td>22,746</td>
<td>148,149</td>
</tr>
</tbody>
</table>

Table D.1: Overview of file system counts for the M57-Patents scenario
Table D.2 displays the complete Registry count for the M57-Patents scenario data set including number of hive files, keys, values, allocated entries, unallocated entries and a total count.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>charlie-2009-11-12</td>
<td>18</td>
<td>103,115</td>
<td>179,971</td>
<td>283,066</td>
<td>20</td>
<td>283,086</td>
</tr>
<tr>
<td>charlie-2009-11-12start</td>
<td>18</td>
<td>102,082</td>
<td>178,397</td>
<td>280,458</td>
<td>21</td>
<td>280,479</td>
</tr>
<tr>
<td>charlie-2009-11-16</td>
<td>18</td>
<td>103,221</td>
<td>180,391</td>
<td>283,568</td>
<td>44</td>
<td>283,612</td>
</tr>
<tr>
<td>charlie-2009-11-17</td>
<td>18</td>
<td>103,509</td>
<td>181,118</td>
<td>284,583</td>
<td>44</td>
<td>284,627</td>
</tr>
<tr>
<td>charlie-2009-11-18</td>
<td>18</td>
<td>103,607</td>
<td>181,398</td>
<td>284,961</td>
<td>44</td>
<td>285,005</td>
</tr>
<tr>
<td>charlie-2009-11-19</td>
<td>18</td>
<td>106,196</td>
<td>185,103</td>
<td>291,255</td>
<td>44</td>
<td>291,299</td>
</tr>
<tr>
<td>charlie-2009-11-20</td>
<td>18</td>
<td>106,510</td>
<td>185,836</td>
<td>292,286</td>
<td>60</td>
<td>292,346</td>
</tr>
<tr>
<td>charlie-2009-11-23</td>
<td>18</td>
<td>106,540</td>
<td>185,931</td>
<td>292,400</td>
<td>71</td>
<td>292,471</td>
</tr>
<tr>
<td>charlie-2009-11-24</td>
<td>18</td>
<td>106,617</td>
<td>186,210</td>
<td>292,769</td>
<td>58</td>
<td>292,827</td>
</tr>
<tr>
<td>charlie-2009-11-30</td>
<td>18</td>
<td>107,243</td>
<td>187,302</td>
<td>294,448</td>
<td>97</td>
<td>294,545</td>
</tr>
<tr>
<td>charlie-2009-12-01</td>
<td>18</td>
<td>107,269</td>
<td>187,438</td>
<td>294,609</td>
<td>98</td>
<td>294,707</td>
</tr>
<tr>
<td>charlie-2009-12-02</td>
<td>18</td>
<td>107,273</td>
<td>187,475</td>
<td>294,635</td>
<td>113</td>
<td>294,748</td>
</tr>
<tr>
<td>charlie-2009-12-03</td>
<td>18</td>
<td>107,252</td>
<td>187,525</td>
<td>294,705</td>
<td>72</td>
<td>294,777</td>
</tr>
<tr>
<td>charlie-2009-12-04</td>
<td>18</td>
<td>107,256</td>
<td>187,533</td>
<td>294,725</td>
<td>64</td>
<td>294,789</td>
</tr>
<tr>
<td>charlie-2009-12-07</td>
<td>18</td>
<td>107,272</td>
<td>187,550</td>
<td>294,757</td>
<td>65</td>
<td>294,822</td>
</tr>
<tr>
<td>charlie-2009-12-08</td>
<td>18</td>
<td>107,271</td>
<td>187,555</td>
<td>294,764</td>
<td>62</td>
<td>294,826</td>
</tr>
<tr>
<td>charlie-2009-12-09</td>
<td>18</td>
<td>107,378</td>
<td>188,177</td>
<td>295,505</td>
<td>50</td>
<td>295,555</td>
</tr>
<tr>
<td>charlie-2009-12-10</td>
<td>18</td>
<td>107,395</td>
<td>188,302</td>
<td>295,647</td>
<td>50</td>
<td>295,697</td>
</tr>
<tr>
<td>charlie-2009-12-11</td>
<td>18</td>
<td>107,403</td>
<td>188,349</td>
<td>295,702</td>
<td>50</td>
<td>295,752</td>
</tr>
<tr>
<td>jo-2009-11-12</td>
<td>18</td>
<td>103,008</td>
<td>179,520</td>
<td>282,498</td>
<td>30</td>
<td>282,528</td>
</tr>
<tr>
<td>jo-2009-11-12start</td>
<td>18</td>
<td>101,875</td>
<td>177,641</td>
<td>279,485</td>
<td>31</td>
<td>279,516</td>
</tr>
<tr>
<td>jo-2009-11-16</td>
<td>18</td>
<td>103,116</td>
<td>179,947</td>
<td>283,036</td>
<td>27</td>
<td>283,063</td>
</tr>
<tr>
<td>jo-2009-11-17</td>
<td>18</td>
<td>103,241</td>
<td>180,424</td>
<td>283,636</td>
<td>29</td>
<td>283,665</td>
</tr>
<tr>
<td>jo-2009-11-18</td>
<td>18</td>
<td>103,337</td>
<td>180,615</td>
<td>283,923</td>
<td>29</td>
<td>283,952</td>
</tr>
<tr>
<td>jo-2009-11-19</td>
<td>18</td>
<td>103,372</td>
<td>180,782</td>
<td>284,110</td>
<td>44</td>
<td>284,154</td>
</tr>
<tr>
<td>jo-2009-11-20-newComputer</td>
<td>18</td>
<td>102,455</td>
<td>178,682</td>
<td>281,004</td>
<td>133</td>
<td>281,137</td>
</tr>
<tr>
<td>jo-2009-11-20-oldComputer</td>
<td>18</td>
<td>107,026</td>
<td>185,995</td>
<td>292,982</td>
<td>39</td>
<td>293,021</td>
</tr>
<tr>
<td>jo-2009-11-23</td>
<td>18</td>
<td>102,643</td>
<td>179,449</td>
<td>282,067</td>
<td>25</td>
<td>282,092</td>
</tr>
<tr>
<td>jo-2009-11-24</td>
<td>18</td>
<td>105,268</td>
<td>183,154</td>
<td>288,418</td>
<td>4</td>
<td>288,422</td>
</tr>
<tr>
<td>jo-2009-11-30</td>
<td>18</td>
<td>107,488</td>
<td>186,769</td>
<td>294,232</td>
<td>25</td>
<td>294,257</td>
</tr>
<tr>
<td>jo-2009-12-01</td>
<td>18</td>
<td>107,501</td>
<td>186,886</td>
<td>294,348</td>
<td>39</td>
<td>294,387</td>
</tr>
<tr>
<td>jo-2009-12-02</td>
<td>18</td>
<td>107,505</td>
<td>186,890</td>
<td>294,354</td>
<td>41</td>
<td>294,395</td>
</tr>
<tr>
<td>jo-2009-12-03</td>
<td>18</td>
<td>107,521</td>
<td>186,969</td>
<td>294,467</td>
<td>23</td>
<td>294,490</td>
</tr>
<tr>
<td>jo-2009-12-04</td>
<td>18</td>
<td>107,552</td>
<td>187,210</td>
<td>294,723</td>
<td>39</td>
<td>294,762</td>
</tr>
<tr>
<td>jo-2009-12-07</td>
<td>18</td>
<td>107,579</td>
<td>187,380</td>
<td>294,906</td>
<td>53</td>
<td>294,959</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>jo-2009-12-08</td>
<td>18</td>
<td>107,586</td>
<td>187,387</td>
<td>294,934</td>
<td>39</td>
<td>294,973</td>
</tr>
<tr>
<td>jo-2009-12-09</td>
<td>18</td>
<td>107,708</td>
<td>187,994</td>
<td>295,702</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jo-2009-12-10</td>
<td>18</td>
<td>107,738</td>
<td>188,059</td>
<td>295,797</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jo-2009-12-11-001</td>
<td>18</td>
<td>107,761</td>
<td>188,166</td>
<td>295,927</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jo-2009-12-11-002</td>
<td>18</td>
<td>107,761</td>
<td>188,183</td>
<td>295,944</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-12</td>
<td>18</td>
<td>103,073</td>
<td>179,719</td>
<td>282,792</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-12start</td>
<td>18</td>
<td>103,284</td>
<td>180,553</td>
<td>283,837</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-16</td>
<td>18</td>
<td>103,630</td>
<td>181,724</td>
<td>284,829</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-17</td>
<td>18</td>
<td>104,702</td>
<td>183,274</td>
<td>287,976</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-18</td>
<td>18</td>
<td>105,281</td>
<td>184,718</td>
<td>289,999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-19</td>
<td>18</td>
<td>105,281</td>
<td>184,758</td>
<td>290,035</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-20</td>
<td>18</td>
<td>105,281</td>
<td>184,788</td>
<td>290,057</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-23</td>
<td>18</td>
<td>105,281</td>
<td>184,894</td>
<td>291,785</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-11-30</td>
<td>18</td>
<td>105,281</td>
<td>184,918</td>
<td>292,136</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-01</td>
<td>18</td>
<td>105,281</td>
<td>185,058</td>
<td>292,939</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-02</td>
<td>18</td>
<td>105,281</td>
<td>185,195</td>
<td>293,926</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-03</td>
<td>18</td>
<td>105,281</td>
<td>185,217</td>
<td>294,034</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-04</td>
<td>18</td>
<td>105,281</td>
<td>185,245</td>
<td>294,526</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-05</td>
<td>18</td>
<td>105,281</td>
<td>185,273</td>
<td>294,604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-06</td>
<td>18</td>
<td>105,281</td>
<td>185,297</td>
<td>294,584</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-07</td>
<td>18</td>
<td>105,281</td>
<td>185,324</td>
<td>294,618</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-08</td>
<td>18</td>
<td>105,281</td>
<td>185,350</td>
<td>294,696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-09</td>
<td>18</td>
<td>105,281</td>
<td>185,375</td>
<td>294,760</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-10</td>
<td>18</td>
<td>105,281</td>
<td>185,400</td>
<td>294,835</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pat-2009-12-11</td>
<td>18</td>
<td>105,281</td>
<td>185,425</td>
<td>294,860</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-12</td>
<td>10</td>
<td>207,109</td>
<td>402,147</td>
<td>568,856</td>
<td>40,400</td>
<td>609,256</td>
</tr>
<tr>
<td>terry-2009-11-12start</td>
<td>10</td>
<td>190,840</td>
<td>348,595</td>
<td>539,435</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-16</td>
<td>10</td>
<td>211,415</td>
<td>412,183</td>
<td>623,598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-17</td>
<td>10</td>
<td>207,648</td>
<td>403,293</td>
<td>610,941</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-18</td>
<td>10</td>
<td>197,831</td>
<td>383,218</td>
<td>569,398</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-19</td>
<td>10</td>
<td>158,689</td>
<td>320,434</td>
<td>479,123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-20</td>
<td>10</td>
<td>174,012</td>
<td>355,585</td>
<td>530,097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-23</td>
<td>10</td>
<td>214,207</td>
<td>410,712</td>
<td>624,919</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-24</td>
<td>10</td>
<td>218,888</td>
<td>424,588</td>
<td>643,476</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-11-30</td>
<td>10</td>
<td>211,607</td>
<td>408,218</td>
<td>620,825</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-12-01</td>
<td>10</td>
<td>215,862</td>
<td>417,870</td>
<td>633,732</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-12-02</td>
<td>10</td>
<td>215,799</td>
<td>418,096</td>
<td>633,895</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-12-03</td>
<td>10</td>
<td>215,994</td>
<td>418,146</td>
<td>634,140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-12-04</td>
<td>10</td>
<td>215,984</td>
<td>418,155</td>
<td>634,139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terry-2009-12-07</td>
<td>10</td>
<td>215,936</td>
<td>418,079</td>
<td>634,015</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

355
### Table D.2: Overview of Registry counts for the M57-Patents scenario

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>terry-2009-12-08</td>
<td>10</td>
<td>215,925</td>
<td>417,784</td>
<td>610,142</td>
<td>23,567</td>
<td>633,709</td>
</tr>
<tr>
<td>terry-2009-12-09</td>
<td>10</td>
<td>216,394</td>
<td>419,257</td>
<td>612,090</td>
<td>23,561</td>
<td>635,651</td>
</tr>
<tr>
<td>terry-2009-12-10</td>
<td>10</td>
<td>222,091</td>
<td>432,699</td>
<td>581,367</td>
<td>73,423</td>
<td>654,790</td>
</tr>
<tr>
<td>terry-2009-12-11-001</td>
<td>10</td>
<td>215,600</td>
<td>421,207</td>
<td>583,057</td>
<td>53,750</td>
<td>636,807</td>
</tr>
<tr>
<td>terry-2009-12-11-002</td>
<td>10</td>
<td>215,355</td>
<td>423,161</td>
<td>584,539</td>
<td>53,977</td>
<td>638,516</td>
</tr>
</tbody>
</table>

### D.2 Path Normalisation Testing

Listing D.1 displays the full source code for the `FSPathDetection.py` script used to perform a comparison of path normalisation effectiveness for file system entries.

```python
# !/usr/bin/python
__version__ = "1.0.0"

import os
import sys
import collections

try:
    import FilePathNormaliser
except ImportError:
    print('Error: FSPathDetection.py')
    print('The FilePathNormaliser.py module is required."
    print('Now Exiting...')
    sys.exit(1)

try:
    import Objects
except ImportError:
    print('Error: FSPathDetection.py')
    print('The Objects.py module is required.')
    print('Now Exiting...')
    sys.exit(1)

try:
    import dfxml
except ImportError:
    print('Error: FSPathDetection.py')
    print('The dfxml.py module is required.')
    print('Now Exiting...')
    sys.exit(1)
```

356
try:
    import apxml
except ImportError:
    print('Error: FSPathDetection.py')
    print('The apxml.py module is required.')
    print('Now Exiting...')
sys.exit(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
class PathNormEffectiveness:
    def __init__(self, xmlfile=None, profile=None):
        """Initialise PathNormEffectiveness object. """

        self.xmlfile = xmlfile
        self.profile = profile

        self.pfos = list()
        self.pfos_norm = collections.defaultdict(list)
        self.pfos_actual = collections.defaultdict(list)

        self.norm_matches = list()
        self.actual_matches = list()

        self.all_files = 0

    def process_apxml(self):
        """Process all APXML documents. """

        apxml_obj = apxml.iterparse(self.profile)
        for pfo in apxml_obj:
            if isinstance(pfo, Objects.FileObject):
                self.pfos.append(pfo)
                self.pfos_norm[pfo.filename_norm].append(pfo)

                if pfo.filename.startswith("C:"):
                    pfo.filename = pfo.filename[3:]

                self.pfos_actual[pfo.filename].append(pfo)

    def process_target(self):
        """Parse the file system of the target data set. """

        print("DFXML: %s" % os.path.basename(self.xmlfile))
        print("APXML: %s" % os.path.basename(self.profile))

        # Process the target data set
        if self.xmlfile is not None:
            # If DFXML from fwiwalk, parse using Objects.iterparse
            for (event, obj) in Objects.iterparse(self.xmlfile):
if isinstance(obj, Objects.FileObject):
    self.process_target_fi(obj)

def process_target_fi(self, tfo):
    """Process each Target FileObject (TFO)""
    # Check if file is to be generically excluded
    if (tfo.filename.endswith("/.") or tfo.filename.endswith("/../")):
        return

    self.all_files += 1

    tfo.filename_norm = self.file_path_normaliser.normalize(tfo.filename)
    tfo.filename = tfo.filename.replace("/", "\\")

    if tfo.filename_norm in self.pfos_norm:
        self.norm_matches.append(tfo)
    if tfo.filename in self.pfos_actual:
        self.actual_matches.append(tfo)

def results(self):
    print("NORM MATCHES: %d" % len(self.norm_matches))
    for tfo in self.norm_matches:
        print(tfo.filename_norm)

    print("ACT MATCHES: %d" % len(self.actual_matches))
    for tfo in self.actual_matches:
        print(tfo.filename)

    print("ALL: %d" % self.all_files)
    print()

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="""FSPathDetection.py"""
    parser.add_argument("apxml", help = "An AXML document")
    parser.add_argument("dfxml", help = "A DFXML report")
    args = parser.parse_args()
    # Parse command line arguments
    xmlfile = args.dfxml
    apxmlfile = args.apxml

    test = PathNormEffectiveness(xmlfile = xmlfile,
Listing D.1: Full source code listing for FSPathDetection.py

Listing D.2 displays the full source code for the REGPathDetection.py script used to perform a comparison of path normalisation effectiveness for Registry entries.
print('Error: REGPathDetection.py')
print('The CellPathNormaliser.py module is required.')
sys.exit(1)

class PathNormEffectiveness:
    def __init__(self, hives=None, profile=None):
        """Initialise PathNormEffectiveness object. """
        self.hives = hives
        self.profile = profile

        self.pfos = list()
        self.pfos_norm = collections.defaultdict(list)
        self.pfos_actual = collections.defaultdict(list)

        # Initialize the file path normaliser object
        self.file_path_normaliser = FilePathNormaliser.FilePathNormaliser()

        # Initialize the cell path normaliser object
        self.cell_path_normaliser = CellPathNormaliser.CellPathNormaliser()

        self.norm_matches = list()
        self.actual_matches = list()

        self.all_cells = 0

        # Set of target hive files to process
        self.target_hives = set()

        # active_hive specifies the hive name being processed
        self.active_hive = None

        # active_rootkey is the common root key name (SOFTWARE, SYSTEM)
        self.active_rootkey = None

    def process_apxml(self):
        """Process all APXML documents. """
        apxml_obj = apxml.iterparse(self.profile)
        for pco in apxml_obj:
            if isinstance(pco, Objects.CellObject):
                self.pfos.append(pco)
                self.pfos_norm[pco.cellpath_norm].append(pco)

                normpath = pco.cellpath
                if normpath.startswith("HKLM\"):
                    normpath = normpath[5:]
                if normpath.startswith("HKU\"):
                    normpath = normpath[4:]
                normpath = normpath.split("\\")
                del normpath[0]
normpath = \"\".join(normpath)
normpath = "NTUSER.DAT\" + normpath
pco.cellpath = normpath
self.pfos_actual[pco.cellpath].append(pco)
rootkey = pco.cellpath_norm.split("\")[0]
self.target_hives.add(rootkey)

def process_target(self):
    """ Parse the file system of the target data set. """
    print("REGXML: %s" % self.hives)
    print("APXML: %s" % os.path.basename(self.profile))

    # Parse target Registry hive files
    print(">>> Processing target hives ...")
    self.to_process = collections.defaultdict(list)

    # Generate RegXML or fetch each needed target hive file
    if self.hives is not None:
        # Fetch all Registry related files
        registry_files = glob.glob(self.hives + "*")

    # Classify required target hives and hive files
    regxml_count = 0
    for fi in registry_files:
        for rootkey in self.target_hives:
            if fi.lower().endswith(rootkey.lower() + ".xml"):
                self.to_process[rootkey].append(fi)
                regxml_count += 1

    # Start processing each Registry hive
    for rootkey in self.target_hives:
        for hive in self.to_process[rootkey]:
            self.active_hive = hive
            self.active_rootkey = rootkey
            for (event, obj) in Objects.iterparse_CellObjects(hive):
                if isinstance(obj, Objects.CellObject):
                    obj.rootkey = rootkey
                    self.process_target_co(obj)

    def process_target_co(self, tco):
        # Check if file is to be generically excluded
        self.all_cells += 1

        # Normalize the TCO rootkey
        if tco.cellpath is not None:
            tco.cellpath_norm = self.cell_path_normaliser.normalize_rootkey(tco.cellpath)
tco.cellpath.lower(),
    self.active_rootkey)

    # Normalize the TCO cell path (full path)
tco.cellpath_norm = self.cell_path_normaliser.normalize_cellpath(
    tco.cellpath_norm,
    self.active_rootkey)

else:
    tco.cellpath_norm == None

# Normalize the TCO basename
if tco.basename:
    if (tco.basename.startswith("C:")) or
        tco.basename.startswith("P:")) or
        tco.basename.startswith("hrzr_ehacngu")):

        # Transform the basename (this ultimately calls file system
        path normaliser)
        normbasename = self.cell_path_normaliser.normalize_basename(
            tco.basename)

        # Replace backslashes with forwardslashes (for consistency)
        normbasename = normbasename.replace("\\", "/")
        tco.basename_norm = normbasename

        # Finally update the cellpath_norm for a basename
        transformation
        if tco.cellpath_norm:
            tco.cellpath_norm = tco.cellpath_norm.replace(
                tco.basename.lower(),
                tco.basename_norm)

    if tco.cellpath:
        normpath = tco.cellpath
        normpath = normpath.split("\\")
        normpath[0] = self.active_rootkey.upper()
        normpath = "/".join(normpath)
        tco.cellpath = normpath

    if tco.cellpath_norm in self.pfos_norm:
        self.norm_matches.append(tco)

    if tco.cellpath in self.pfos_actual:
        self.actual_matches.append(tco)

def results(self):
    print("NORM MATCHES: %d" % len(self.norm_matches))

    for tfo in self.norm_matches:
```python
print(tfo.cellpath_norm)
print("ACT MATCHES: %d\n\nlen(self.actual_matches))")
for tfo in self.actual_matches:
    print(tfo.cellpath)
print("ALL: %d \n\nself.all_cells")
print()

Listing D.2: Full source code listing for REGPathDetection.py
```
D.3 Hard Drive Read Speed Testing

Listing D.3 displays the full Windows batch scripts used to measure the drive read speed.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>winsat disk --seq --read --drive e --count 10 &gt;&gt; SATA_WDGreen.txt</td>
<td></td>
</tr>
</tbody>
</table>

Listing D.3: Windows batch script to determine hard drive read speed

Listing D.4 displays the results from executing the hard drive read speed test using the Western Digital Green 2.0 TB desktop 3.5-inch hard drive connected via SATA 3.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows System Assessment Tool</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&gt; Running: Feature Enumeration ' '</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&gt; Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&gt; Running: Storage Assessment ' --seq --read --drive e --count 10'</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&gt; Run Time 00:00:44.16</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&gt; Disk Sequential 64.0 Read 119.14 MB/s 7.2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&gt; Total Run Time 00:00:45.01</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Windows System Assessment Tool</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&gt; Running: Feature Enumeration ' '</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&gt; Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&gt; Running: Storage Assessment ' --seq --read --drive e --count 10'</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>&gt; Run Time 00:00:43.35</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>&gt; Disk Sequential 64.0 Read 119.26 MB/s 7.2</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>&gt; Total Run Time 00:00:43.74</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Windows System Assessment Tool</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>&gt; Running: Feature Enumeration ' '</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>&gt; Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>&gt; Running: Storage Assessment ' --seq --read --drive e --count 10'</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&gt; Run Time 00:00:42.07</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>&gt; Disk Sequential 64.0 Read 119.14 MB/s 7.2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>&gt; Total Run Time 00:00:42.57</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Windows System Assessment Tool</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>&gt; Running: Feature Enumeration ' '</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>&gt; Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>&gt; Running: Storage Assessment ' --seq --read --drive e --count 10'</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>&gt; Run Time 00:00:41.70</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>&gt; Disk Sequential 64.0 Read 120.41 MB/s 7.2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>&gt; Total Run Time 00:00:42.18</td>
<td></td>
</tr>
</tbody>
</table>
Listing D.4: Results from hard drive read speed testing
Listing D.5 displays the results from executing the hard drive read speed test using the Western Digital Green 2.0 TB desktop 3.5-inch hard drive connected via the external HDD caddy.

<table>
<thead>
<tr>
<th>Windows System Assessment Tool</th>
<th>Running: Feature Enumeration</th>
<th>Run Time 00:00:00.00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:35.58</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>31.88 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:35.77</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:35.30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>31.91 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:35.46</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:36.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>32.16 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:37.09</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:34.79</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>31.71 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:34.95</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:34.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>31.99 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:35.00</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:35.87</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk Sequential 64.0 Read</td>
<td>32.01 MB/s</td>
</tr>
<tr>
<td></td>
<td>Total Run Time 00:01:36.04</td>
<td></td>
</tr>
<tr>
<td>Windows System Assessment Tool</td>
<td>Running: Feature Enumeration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Run Time 00:00:00.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running: Storage Assessment</td>
<td>seq -read -drive d -count 10'</td>
</tr>
<tr>
<td></td>
<td>Run Time 00:01:35.00</td>
<td></td>
</tr>
</tbody>
</table>

366
Listing D.5: Results from hard drive read speed testing for the external HDD caddy

Listing D.6 displays the results from executing the hard drive read speed test using the Western Digital Green 2.0 TB desktop 3.5-inch hard drive connected to the triage field laptop using the external HDD caddy.

```plaintext
Windows System Assessment Tool
> Running: Feature Enumeration ''
> Run Time 00:00:00.00
> Running: Storage Assessment '-seq -read -drive d -count 10'
> Run Time 00:01:36.19
> Disk Sequential 64.0 Read 32.21 MB/s 4.5
> Total Run Time 00:01:36.37
Windows System Assessment Tool
> Running: Feature Enumeration ''
> Run Time 00:00:00.00
> Running: Storage Assessment '-seq -read -drive d -count 10'
> Run Time 00:01:38.44
> Disk Sequential 64.0 Read 31.80 MB/s 4.5
> Total Run Time 00:01:38.53
```
Disk Sequential 64.0 Read 32.82 MB/s 4.5
Total Run Time 00:01:38.67
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:38.05
Disk Sequential 64.0 Read 32.83 MB/s 4.5
Total Run Time 00:01:38.22
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:38.26
Disk Sequential 64.0 Read 32.90 MB/s 4.5
Total Run Time 00:01:38.44
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:38.94
Disk Sequential 64.0 Read 32.70 MB/s 4.5
Total Run Time 00:01:39.11
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:38.34
Disk Sequential 64.0 Read 32.80 MB/s 4.5
Total Run Time 00:01:38.53
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:37.47
Disk Sequential 64.0 Read 32.88 MB/s 4.5
Total Run Time 00:01:37.64
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:38.24
Disk Sequential 64.0 Read 32.86 MB/s 4.5
Total Run Time 00:01:38.41
Windows System Assessment Tool
Running: Feature Enumeration ' '
Run Time 00:00:00.00
Running: Storage Assessment '−seq −read −drive d −count 10'
Run Time 00:01:39.22
Disk Sequential 64.0 Read 32.92 MB/s 4.5
Listing D.6: Results from hard drive read speed testing of the triage field laptop system
D.4 Additional fiwalk Development

A collection of issues were discovered when performing experimental testing with fiwalk while attempting to increase system efficiency. Listing D.7 displays the patch file authored to solve two bugs found in the latest fiwalk release: 1) By default fiwalk prints the input file name to standard output which means that the DFXML API (Objects.py) will crash due to an XML error when ingesting; and 2) An error was found in handling command line arguments and fiwalk was unable to parse only file system metadata and always exported data location (byte_runs). Both these problems were solved using the simple patch provided.

Listing D.7: Patch file to fix two fiwalk bugs
In addition to the discovered programming bugs it was found difficult to cross-compile fiwalk for Microsoft Windows. Listing D.8 provides a simple overview of how the author cross-compiled fiwalk for Windows using MinGW.

```
# Download MinGW installer from
download
#
# Make sure to install common build packages
#
# REGEX
# Manually download and install regex, other options will fail
# Download from:
https://sourceforge.net/projects/mingw/files/MSYS/Base/regex/regex-
1.20090805-2/
#
# Change to directory
C:\msys\1.0\home\dsb\msys-build-regex
#Locate the following lines (using a '#' )
if [ "$MSYSTEM" != "MSYS" ]
then
  echo "You must be in an MSYS shell to use this script"
  exit 1
fi
#
# The five lines should look like this:
# if [ "$MSYSTEM" != "MSYS" ]
# then
#  echo "You must be in an MSYS shell to use this script"
#  exit 1
#fi
#
# In the MSYS shell, execute the following commands:
cd /home/dsb
./msys-build-regex regex-20090805.tar.xz
cd /home/dsb/regex-20090805
./configure --prefix=/mingw --with-curses
make
make install
#
# THE SLEUTH KIT (TSK)
# Download from GitHub
https://github.com/sleuthkit/sleuthkit
#
# Extract then configure and install
tar xvf sleuthkit-master
./configure --build=i586--mingw32msvc --disable-java --without-afflib --
  disable-shared --enable-static
```
Listing D.8: Quick Windows cross-compilation guide for fiwalk
D.5 Real-World Data Set Sourced Hard Drives

Table D.3 displays all second-hand hard drives collected for system evaluation. The naming convention for each disk is provided with the digital device type, whether the device could be forensically captured, if the device has a valid file system (Has FS), the file system type (if applicable) and if the device was found to have a *Windows* directory in the root file system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Imaged</th>
<th>Has FS</th>
<th>FS Type</th>
<th>Has Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_K_1</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>A_K_2</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>A_K_3</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_39</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_40</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_41</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_42</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_43</td>
<td>IDE</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDE_44</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_45</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_46</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_47</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_48</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_49</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_50</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_52</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat16</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_53</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_54</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_58</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat16</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_59</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_60</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_61</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_66</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat16</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_69</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_70</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_71</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_72</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_73</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_74</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ext4</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_75</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_76</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_77</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>hfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_78</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_79</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Imaged</td>
<td>Has FS</td>
<td>FS Type</td>
<td>Has Windows</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>IDE_80</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_81</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_82</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_83</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_84</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_85</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_86</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat12</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_87</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_88</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_89</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_90</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_91</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_92</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>IDE_93</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>IDE_94</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>SATA_36</td>
<td>SATA</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>SATA_37</td>
<td>SATA</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>SCSI_01</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_02</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_03</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_04</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_05</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_06</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_07</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_08</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_09</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_10</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_11</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_12</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_13</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_14</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_15</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_16</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_17</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_18</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_19</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_20</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_21</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_22</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_23</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_24</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_25</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_26</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_27</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_28</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_29</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

374
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Imaged</th>
<th>Has FS</th>
<th>FS Type</th>
<th>Has Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_30</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_31</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_32</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_33</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_35</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_56</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>SCSI_65</td>
<td>SCSI</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0101ST</td>
<td>SATA</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0102ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0103ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0104ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0105ST</td>
<td>SATA</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0106ST</td>
<td>SATA</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0107ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0108ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0109ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0110ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0111ST</td>
<td>SATA</td>
<td>NO</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0112ST</td>
<td>SATA</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0113ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0114ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0115ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0116ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0117ID</td>
<td>IDE</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0118ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0119ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0120ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0121ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0122ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0123ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0124ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0125ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0126ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0127ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0128ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0129ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0130ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0131ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0132ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Imaged</td>
<td>Has FS</td>
<td>FS Type</td>
<td>Has Windows</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>TM0133ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0134ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0135ST</td>
<td>SATA</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0136ST</td>
<td>SATA</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0137ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0138ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat16</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0139ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>hfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0140ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0141ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0142SC</td>
<td>SCSI</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0143SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0144SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0145SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0146SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0147SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0148SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0149SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0150SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0151SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0152SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0153SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0154SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0155SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0156SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0157SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0158SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0159SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0160SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0161SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0162SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0163SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0164SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0165SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0166SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0167SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0168SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0169SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0170SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0171SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

376
Table D.3: Overview of the drives sourced for the real world data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Imaged</th>
<th>Has FS</th>
<th>FS Type</th>
<th>Has Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM0172SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0173SC</td>
<td>SCSI</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0174SC</td>
<td>SCSI</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0175SC</td>
<td>SCSI</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM0201ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0202ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0301ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0401ID</td>
<td>IDE</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0501ID</td>
<td>IDE</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM0601ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0701ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0801ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM0901ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1001ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM1101ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1102ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>TRUE</td>
</tr>
<tr>
<td>TM1103ID</td>
<td>IDE</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1201ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1202ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1203ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM1401ID</td>
<td>IDE</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TM1501ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>fat32</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM2001ST</td>
<td>SATA</td>
<td>YES</td>
<td>FALSE</td>
<td>None</td>
<td>FALSE</td>
</tr>
<tr>
<td>TM2101ID</td>
<td>IDE</td>
<td>YES</td>
<td>TRUE</td>
<td>ntfs</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

D.6 Real-World Data Set Contents

Table D.4 displays the complete file system counts for the real-world data set including the number of hive files, keys, values, allocated entries, unallocated entries and a total count.
<table>
<thead>
<tr>
<th>Name</th>
<th>Dirs</th>
<th>Files</th>
<th>Other</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE_50</td>
<td>1,616</td>
<td>9,492</td>
<td>1,073</td>
<td>10,078</td>
<td>2,103</td>
<td>12,181</td>
</tr>
<tr>
<td>IDE_52</td>
<td>2,281</td>
<td>19,114</td>
<td>6</td>
<td>15,233</td>
<td>6,168</td>
<td>21,401</td>
</tr>
<tr>
<td>IDE_53</td>
<td>10,844</td>
<td>51,957</td>
<td>2,133</td>
<td>34,157</td>
<td>30,777</td>
<td>64,934</td>
</tr>
<tr>
<td>IDE_59</td>
<td>10,825</td>
<td>48,678</td>
<td>2,132</td>
<td>38,919</td>
<td>22,716</td>
<td>61,635</td>
</tr>
<tr>
<td>IDE_60</td>
<td>1,100</td>
<td>19,251</td>
<td>3</td>
<td>19,604</td>
<td>750</td>
<td>20,354</td>
</tr>
<tr>
<td>IDE_61</td>
<td>11,520</td>
<td>55,282</td>
<td>4,265</td>
<td>49,362</td>
<td>21,705</td>
<td>71,067</td>
</tr>
<tr>
<td>IDE_66</td>
<td>136</td>
<td>2,012</td>
<td>3</td>
<td>2,002</td>
<td>149</td>
<td>2,151</td>
</tr>
<tr>
<td>IDE_69</td>
<td>2,075</td>
<td>8,729</td>
<td>272</td>
<td>10,787</td>
<td>289</td>
<td>11,076</td>
</tr>
<tr>
<td>IDE_70</td>
<td>9,612</td>
<td>44,159</td>
<td>1,676</td>
<td>52,773</td>
<td>2,674</td>
<td>55,447</td>
</tr>
<tr>
<td>IDE_71</td>
<td>5,679</td>
<td>31,077</td>
<td>2,392</td>
<td>35,922</td>
<td>3,226</td>
<td>39,148</td>
</tr>
<tr>
<td>IDE_73</td>
<td>9,205</td>
<td>32,667</td>
<td>1,010</td>
<td>37,485</td>
<td>5,397</td>
<td>42,882</td>
</tr>
<tr>
<td>IDE_75</td>
<td>25,466</td>
<td>145,497</td>
<td>14,944</td>
<td>170,346</td>
<td>15,561</td>
<td>185,907</td>
</tr>
<tr>
<td>IDE_76</td>
<td>6,248</td>
<td>29,920</td>
<td>1,831</td>
<td>34,544</td>
<td>3,455</td>
<td>37,999</td>
</tr>
<tr>
<td>IDE_78</td>
<td>47,740</td>
<td>176,242</td>
<td>10,709</td>
<td>191,906</td>
<td>42,785</td>
<td>234,691</td>
</tr>
<tr>
<td>IDE_81</td>
<td>6,205</td>
<td>69,842</td>
<td>6</td>
<td>32,758</td>
<td>43,295</td>
<td>76,053</td>
</tr>
<tr>
<td>IDE_82</td>
<td>1,683</td>
<td>17,440</td>
<td>3</td>
<td>15,967</td>
<td>3,159</td>
<td>19,126</td>
</tr>
<tr>
<td>IDE_83</td>
<td>2,091</td>
<td>15,156</td>
<td>3</td>
<td>10,659</td>
<td>6,591</td>
<td>17,250</td>
</tr>
<tr>
<td>IDE_84</td>
<td>11,076</td>
<td>162,595</td>
<td>6</td>
<td>82,707</td>
<td>90,970</td>
<td>173,677</td>
</tr>
<tr>
<td>IDE_88</td>
<td>9,393</td>
<td>34,595</td>
<td>1,566</td>
<td>32,470</td>
<td>13,084</td>
<td>45,554</td>
</tr>
<tr>
<td>IDE_89</td>
<td>40,079</td>
<td>125,700</td>
<td>4,866</td>
<td>158,807</td>
<td>11,838</td>
<td>170,645</td>
</tr>
<tr>
<td>IDE_91</td>
<td>33,731</td>
<td>118,072</td>
<td>8,553</td>
<td>68,342</td>
<td>92,014</td>
<td>160,356</td>
</tr>
<tr>
<td>SATA_38</td>
<td>2,863</td>
<td>18,359</td>
<td>841</td>
<td>15,451</td>
<td>6,612</td>
<td>22,063</td>
</tr>
<tr>
<td>SCSI_65</td>
<td>4,817</td>
<td>25,558</td>
<td>899</td>
<td>23,162</td>
<td>8,112</td>
<td>31,274</td>
</tr>
<tr>
<td>TM0101ST</td>
<td>434</td>
<td>3,174</td>
<td>36</td>
<td>3,607</td>
<td>37</td>
<td>3,644</td>
</tr>
<tr>
<td>TM0103ID</td>
<td>17,209</td>
<td>101,253</td>
<td>7,659</td>
<td>91,513</td>
<td>34,608</td>
<td>126,121</td>
</tr>
<tr>
<td>TM0106ST</td>
<td>34,997</td>
<td>228,910</td>
<td>40,799</td>
<td>254,303</td>
<td>50,403</td>
<td>304,706</td>
</tr>
<tr>
<td>TM0107ID</td>
<td>10,271</td>
<td>68,424</td>
<td>3,511</td>
<td>78,694</td>
<td>3,512</td>
<td>82,206</td>
</tr>
<tr>
<td>TM0108ID</td>
<td>53,366</td>
<td>408,956</td>
<td>8,428</td>
<td>164,191</td>
<td>306,559</td>
<td>470,750</td>
</tr>
<tr>
<td>TM0110ID</td>
<td>26,057</td>
<td>118,360</td>
<td>4,244</td>
<td>80,423</td>
<td>68,238</td>
<td>148,661</td>
</tr>
<tr>
<td>TM0113ID</td>
<td>2,265</td>
<td>10,242</td>
<td>319</td>
<td>9,822</td>
<td>3,004</td>
<td>12,826</td>
</tr>
<tr>
<td>TM0114ID</td>
<td>10,700</td>
<td>39,491</td>
<td>2,677</td>
<td>49,119</td>
<td>3,749</td>
<td>52,868</td>
</tr>
<tr>
<td>TM0115ID</td>
<td>5,427</td>
<td>30,812</td>
<td>3</td>
<td>26,309</td>
<td>9,933</td>
<td>36,242</td>
</tr>
<tr>
<td>TM0116ID</td>
<td>20,456</td>
<td>96,753</td>
<td>4,410</td>
<td>71,848</td>
<td>49,771</td>
<td>121,619</td>
</tr>
<tr>
<td>TM0118ID</td>
<td>11,727</td>
<td>101,337</td>
<td>15,185</td>
<td>77,339</td>
<td>50,910</td>
<td>128,249</td>
</tr>
<tr>
<td>TM0119ID</td>
<td>13,907</td>
<td>163,038</td>
<td>6</td>
<td>63,796</td>
<td>113,155</td>
<td>176,951</td>
</tr>
<tr>
<td>TM0120ID</td>
<td>13,109</td>
<td>39,559</td>
<td>3,086</td>
<td>48,160</td>
<td>7,594</td>
<td>55,754</td>
</tr>
<tr>
<td>TM0122ID</td>
<td>12,611</td>
<td>56,547</td>
<td>5,924</td>
<td>68,504</td>
<td>6,578</td>
<td>75,082</td>
</tr>
<tr>
<td>TM0123ID</td>
<td>12,885</td>
<td>61,043</td>
<td>6,274</td>
<td>72,705</td>
<td>7,497</td>
<td>80,202</td>
</tr>
<tr>
<td>TM0125ID</td>
<td>1,799</td>
<td>9,351</td>
<td>265</td>
<td>11,123</td>
<td>292</td>
<td>11,415</td>
</tr>
</tbody>
</table>
Table D.4: Overview of file system counts for the real-world data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Dirs</th>
<th>Files</th>
<th>Other</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM0126ID</td>
<td>4,196</td>
<td>15,184</td>
<td>744</td>
<td>19,378</td>
<td>746</td>
<td>20,124</td>
</tr>
<tr>
<td>TM0127ID</td>
<td>11,996</td>
<td>49,888</td>
<td>4,208</td>
<td>54,981</td>
<td>11,111</td>
<td>66,092</td>
</tr>
<tr>
<td>TM0129ID</td>
<td>7,241</td>
<td>31,726</td>
<td>1,198</td>
<td>38,967</td>
<td>1,198</td>
<td>40,165</td>
</tr>
<tr>
<td>TM0130ID</td>
<td>25,583</td>
<td>99,367</td>
<td>3,886</td>
<td>124,879</td>
<td>3,957</td>
<td>128,836</td>
</tr>
<tr>
<td>TM0131ID</td>
<td>23,772</td>
<td>62,634</td>
<td>3,692</td>
<td>79,110</td>
<td>10,988</td>
<td>90,098</td>
</tr>
<tr>
<td>TM0132ID</td>
<td>5,372</td>
<td>18,919</td>
<td>524</td>
<td>24,284</td>
<td>531</td>
<td>24,815</td>
</tr>
<tr>
<td>TM0133ID</td>
<td>23,515</td>
<td>49,888</td>
<td>4,208</td>
<td>54,981</td>
<td>11,111</td>
<td>66,092</td>
</tr>
<tr>
<td>TM0134ID</td>
<td>33,682</td>
<td>61,916</td>
<td>2,657</td>
<td>92,186</td>
<td>6,069</td>
<td>98,255</td>
</tr>
<tr>
<td>TM0135ST</td>
<td>11,861</td>
<td>43,426</td>
<td>3,310</td>
<td>54,915</td>
<td>3,682</td>
<td>58,597</td>
</tr>
<tr>
<td>TM0140ID</td>
<td>5,975</td>
<td>40,987</td>
<td>1,588</td>
<td>46,312</td>
<td>2,238</td>
<td>48,550</td>
</tr>
<tr>
<td>TM0141ID</td>
<td>431</td>
<td>1,687</td>
<td>61</td>
<td>2,118</td>
<td>61</td>
<td>2,179</td>
</tr>
<tr>
<td>TM0142SC</td>
<td>33,229</td>
<td>177,386</td>
<td>7,554</td>
<td>129,505</td>
<td>88,664</td>
<td>218,169</td>
</tr>
<tr>
<td>TM0175SC</td>
<td>12,725</td>
<td>61,475</td>
<td>3,310</td>
<td>54,915</td>
<td>3,682</td>
<td>58,597</td>
</tr>
<tr>
<td>TM1001ID</td>
<td>355</td>
<td>2,953</td>
<td>33</td>
<td>3,288</td>
<td>33</td>
<td>3,321</td>
</tr>
<tr>
<td>TM1102ID</td>
<td>653</td>
<td>10,457</td>
<td>3</td>
<td>3,230</td>
<td>7,883</td>
<td>11,113</td>
</tr>
<tr>
<td>TM2101ID</td>
<td>59,840</td>
<td>2,235</td>
<td>32,557</td>
<td>41,346</td>
<td>73,903</td>
<td></td>
</tr>
</tbody>
</table>

Table D.5 displays the complete Registry counts for the real-world data set including the number of directories (dirs), files, other file system data types, allocated entries, unallocated entries and a total count.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE_50</td>
<td>12</td>
<td>33,295</td>
<td>62,069</td>
<td>95,363</td>
<td>1</td>
<td>95,364</td>
</tr>
<tr>
<td>IDE_52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE_53</td>
<td>24</td>
<td>46,567</td>
<td>107,150</td>
<td>152,301</td>
<td>1,416</td>
<td>153,717</td>
</tr>
<tr>
<td>IDE_59</td>
<td>31</td>
<td>52,822</td>
<td>142,598</td>
<td>192,473</td>
<td>2,947</td>
<td>195,420</td>
</tr>
<tr>
<td>IDE_60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE_61</td>
<td>28</td>
<td>138,510</td>
<td>275,563</td>
<td>411,783</td>
<td>2,290</td>
<td>414,073</td>
</tr>
<tr>
<td>IDE_66</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE_69</td>
<td>18</td>
<td>70,268</td>
<td>112,857</td>
<td>183,057</td>
<td>68</td>
<td>183,125</td>
</tr>
<tr>
<td>IDE_70</td>
<td>26</td>
<td>141,363</td>
<td>246,348</td>
<td>375,538</td>
<td>12,173</td>
<td>387,711</td>
</tr>
<tr>
<td>IDE_71</td>
<td>22</td>
<td>102,136</td>
<td>189,825</td>
<td>291,396</td>
<td>565</td>
<td>291,961</td>
</tr>
<tr>
<td>IDE_73</td>
<td>25</td>
<td>39,354</td>
<td>81,582</td>
<td>120,762</td>
<td>174</td>
<td>120,936</td>
</tr>
<tr>
<td>IDE_75</td>
<td>36</td>
<td>189,641</td>
<td>397,263</td>
<td>548,006</td>
<td>38,899</td>
<td>586,905</td>
</tr>
<tr>
<td>IDE_76</td>
<td>18</td>
<td>40,331</td>
<td>85,318</td>
<td>124,848</td>
<td>801</td>
<td>125,649</td>
</tr>
<tr>
<td>IDE_78</td>
<td>39</td>
<td>96,610</td>
<td>205,230</td>
<td>300,904</td>
<td>936</td>
<td>301,840</td>
</tr>
</tbody>
</table>

379
<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE_81</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE_82</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE_83</td>
<td>16</td>
<td>79,140</td>
<td>135,184</td>
<td>214,309</td>
<td>15</td>
<td>214,324</td>
</tr>
<tr>
<td>IDE_84</td>
<td>12</td>
<td>39,040</td>
<td>80,614</td>
<td>119,063</td>
<td>591</td>
<td>119,654</td>
</tr>
<tr>
<td>IDE_88</td>
<td>13</td>
<td>105,545</td>
<td>158,748</td>
<td>264,026</td>
<td>267</td>
<td>264,293</td>
</tr>
<tr>
<td>IDE_89</td>
<td>19</td>
<td>185,526</td>
<td>315,685</td>
<td>501,159</td>
<td>52</td>
<td>501,211</td>
</tr>
<tr>
<td>IDE_91</td>
<td>95</td>
<td>198,946</td>
<td>486,562</td>
<td>681,570</td>
<td>3,938</td>
<td>685,508</td>
</tr>
<tr>
<td>SATA_38</td>
<td>16</td>
<td>97,829</td>
<td>168,763</td>
<td>266,454</td>
<td>138</td>
<td>266,592</td>
</tr>
<tr>
<td>SCSI_65</td>
<td>6</td>
<td>3,918</td>
<td>11,533</td>
<td>15,348</td>
<td>103</td>
<td>15,451</td>
</tr>
<tr>
<td>TM0101ST</td>
<td>2</td>
<td>3,927</td>
<td>7,175</td>
<td>11,096</td>
<td>6</td>
<td>11,102</td>
</tr>
<tr>
<td>TM0103ID</td>
<td>22</td>
<td>160,977</td>
<td>292,371</td>
<td>449,300</td>
<td>4,048</td>
<td>453,348</td>
</tr>
<tr>
<td>TM0106ST</td>
<td>71</td>
<td>230,657</td>
<td>586,603</td>
<td>766,837</td>
<td>50,423</td>
<td>817,260</td>
</tr>
<tr>
<td>TM0107ID</td>
<td>26</td>
<td>127,499</td>
<td>223,621</td>
<td>350,895</td>
<td>226</td>
<td>351,121</td>
</tr>
<tr>
<td>TM0108ID</td>
<td>19</td>
<td>80,848</td>
<td>140,028</td>
<td>220,700</td>
<td>176</td>
<td>220,876</td>
</tr>
<tr>
<td>TM0110ID</td>
<td>51</td>
<td>168,164</td>
<td>330,016</td>
<td>475,292</td>
<td>22,888</td>
<td>498,180</td>
</tr>
<tr>
<td>TM0113ID</td>
<td>11</td>
<td>52,484</td>
<td>103,987</td>
<td>156,391</td>
<td>80</td>
<td>156,471</td>
</tr>
<tr>
<td>TM0114ID</td>
<td>35</td>
<td>137,856</td>
<td>267,784</td>
<td>404,700</td>
<td>940</td>
<td>405,640</td>
</tr>
<tr>
<td>TM0115ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TM0116ID</td>
<td>47</td>
<td>163,638</td>
<td>323,229</td>
<td>465,733</td>
<td>21,134</td>
<td>486,867</td>
</tr>
<tr>
<td>TM0118ID</td>
<td>30</td>
<td>122,634</td>
<td>210,243</td>
<td>328,998</td>
<td>3,879</td>
<td>332,877</td>
</tr>
<tr>
<td>TM0119ID</td>
<td>18</td>
<td>146,737</td>
<td>247,912</td>
<td>380,843</td>
<td>13,806</td>
<td>394,649</td>
</tr>
<tr>
<td>TM0120ID</td>
<td>24</td>
<td>125,793</td>
<td>241,389</td>
<td>364,269</td>
<td>2,913</td>
<td>367,182</td>
</tr>
<tr>
<td>TM0122ID</td>
<td>18</td>
<td>148,955</td>
<td>270,041</td>
<td>417,349</td>
<td>1,649</td>
<td>418,998</td>
</tr>
<tr>
<td>TM0123ID</td>
<td>18</td>
<td>201,907</td>
<td>360,764</td>
<td>562,020</td>
<td>651</td>
<td>562,671</td>
</tr>
<tr>
<td>TM0125ID</td>
<td>16</td>
<td>79,359</td>
<td>130,920</td>
<td>210,267</td>
<td>12</td>
<td>210,279</td>
</tr>
<tr>
<td>TM0126ID</td>
<td>16</td>
<td>133,943</td>
<td>232,472</td>
<td>364,056</td>
<td>2,359</td>
<td>366,415</td>
</tr>
<tr>
<td>TM0127ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TM0129ID</td>
<td>23</td>
<td>122,857</td>
<td>206,713</td>
<td>329,243</td>
<td>327</td>
<td>329,570</td>
</tr>
<tr>
<td>TM0130ID</td>
<td>24</td>
<td>184,445</td>
<td>319,796</td>
<td>503,791</td>
<td>450</td>
<td>504,241</td>
</tr>
<tr>
<td>TM0131ID</td>
<td>40</td>
<td>192,576</td>
<td>371,924</td>
<td>554,316</td>
<td>10,184</td>
<td>564,500</td>
</tr>
<tr>
<td>TM0132ID</td>
<td>18</td>
<td>114,310</td>
<td>201,197</td>
<td>315,393</td>
<td>114</td>
<td>315,507</td>
</tr>
<tr>
<td>TM0133ID</td>
<td>51</td>
<td>219,542</td>
<td>416,122</td>
<td>616,698</td>
<td>18,866</td>
<td>635,564</td>
</tr>
<tr>
<td>TM0134ID</td>
<td>49</td>
<td>194,194</td>
<td>366,233</td>
<td>559,037</td>
<td>1,390</td>
<td>560,427</td>
</tr>
<tr>
<td>TM0135ST</td>
<td>32</td>
<td>105,872</td>
<td>222,691</td>
<td>329,798</td>
<td>18,765</td>
<td>328,563</td>
</tr>
<tr>
<td>TM0140ID</td>
<td>21</td>
<td>138,480</td>
<td>236,286</td>
<td>359,629</td>
<td>15,137</td>
<td>374,766</td>
</tr>
<tr>
<td>TM0141ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TM0142SC</td>
<td>17</td>
<td>116,154</td>
<td>237,905</td>
<td>324,696</td>
<td>29,363</td>
<td>354,059</td>
</tr>
<tr>
<td>TM0175SC</td>
<td>20</td>
<td>277,985</td>
<td>498,076</td>
<td>775,870</td>
<td>191</td>
<td>776,061</td>
</tr>
<tr>
<td>TM1001ID</td>
<td>2</td>
<td>3,713</td>
<td>6,940</td>
<td>10,646</td>
<td>7</td>
<td>10,653</td>
</tr>
</tbody>
</table>

380
Table D.5: Overview of Registry counts for the real-world data set

<table>
<thead>
<tr>
<th>Name</th>
<th>Hives</th>
<th>Keys</th>
<th>Values</th>
<th>Alloc</th>
<th>Unalloc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM1102ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TM2101ID</td>
<td>16</td>
<td>156,253</td>
<td>276,061</td>
<td>339,509</td>
<td>92,805</td>
<td>432,314</td>
</tr>
</tbody>
</table>