UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The development of an open-source forensics platform

by

Renico Koen

Submitted in fulfilment of the requirements for the degree

MSc Computer Science

in the Faculty of

Engineering, Built Environment and Information Technology,

University of Pretoria

February 2009.

Contents

Summary

The rate at which technology evolves by far outpaces the rate at which methods are developed to prevent and prosecute digital crime. This unfortunate situation may potentially allow computer criminals to commit crimes using technologies for which no proper forensic investigative technique currently exists. Such a scenario would ultimately allow criminals to go free due to the lack of evidence to prove their guilt.

A solution to this problem would be for law enforcement agencies and governments to invest in the research and development of forensic technologies in an attempt to keep pace with the development of digital technologies. Such an investment could potentially allow new forensic techniques to be developed and released more frequently, thus matching the appearance of new computing devices on the market.

A key element in improving the situation is to produce more research results, utilizing less resources, and by performing research more efficiently. This can be achieved by improving the process used to conduct forensic research. One of the problem areas in research and development is the development of prototypes to prove a concept or to test a hypothesis. An in-depth understanding of the extremely technical aspects of operating systems, such as file system structures and memory management, is required to allow forensic researchers to develop prototypes to prove their theories and techniques.

The development of such prototypes is an extremely challenging task. It is complicated by the presence of minute details that, if ignored, may have a negative impact on the accuracy of results produced. If some of the complexities experienced in the development of prototypes could simply be removed from the equation, researchers may be able to produce more and better results with less effort, and thus ultimately speed up the forensic research process.

This dissertation describes the development of a platform that facilitates the rapid development of forensic prototypes, thus allowing researchers to produce such prototypes utilizing less time and fewer resources. The purpose of the platform is to provide a set of rich features which are likely to be required by developers performing research prototyping. The proposed platform contributes to the development of prototypes using fewer resources and at a faster pace.

The development of the platform, as well as various considerations that helped to shape its architecture and design, are the focus points of this dissertation. Topics such as digital forensic investigations, open-source software development, and the development of the proposed forensic platform are discussed. Another purpose of this dissertation is to serve as a proof-of-concept for the developed platform. The development of a selection of forensics prototypes, as well as the results obtained, are also discussed.

1 Development of an Open-source Forensics Platform

1.1 Introduction

As the processing capabilities of digital devices increase, so does the level of power that humans may obtain through the use of these devices. This immense power may be utilized by individuals or organizations to uplift and enrich countless lives, leading to a better quality of life for all. Unfortunately this immense power may also be used to enrich the lives of a crooked few by taking advantage of unsuspecting victims through computer-related crimes.

Digital forensics plays a crucial part in the investigation of crimes involving electronic equipment. Digital forensic techniques are used primarily by law enforcement agencies to capture, preserve and analyze evidence on digital devices. Digital evidence collected at a crime scene has to be analyzed, and connections between the recovered information, physical entities and physical events need to be made and proven.

The storage ability of digital computing devices is regularly improved along with their processing power; better and faster devices are typically able to store more data than their less technologically-advanced counterparts. The task of finding any evidence on high-capacity storage devices through inspection can be compared to the task of finding a needle in a haystack - a time-consuming process which may yield few or no results.

The search for digital evidence is thus a task of great proportions that consumes a considerable amount of manpower and time. When taking into account the loads of data that need to be processed in a short time in a digital forensics case, it is understandable that a huge backlog on digital evidence processing exists, ranging from a few months to a few years [30]. This is due mainly to the fact that an extremely large amount of evidence needs to be processed in a very limited time frame, which causes unimaginable delays in processing schedules.

1.2 Problem statement

Researchers in the field of digital forensics are constantly trying to find better and more efficient ways of uncovering evidence from digital sources. Unfortunately the research process itself is often time-intensive and consumes a considerable amount of resources. One of the aspects of forensics research that may typically consume a considerable amount of time is prototyping. The development of forensics prototypes is an extremely complex task, requiring an in-depth understanding of the inner-workings of the selected computing devices. The development of a forensics prototype would typically require a researcher to develop immensely complex code that would perform tasks similar to those of the operating system or application in question, in an attempt to uncover stored

data.

Specialized tools already exist to capture, preserve and analyze data from hard drives, memory and network streams. These tools typically already contain most of the functionality required by researchers to develop forensics prototypes. Unfortunately very few of these tools are extendable by third parties. This is unfortunate, as the reuse of existing tried-and-tested forensic routines could potentially decrease the amount of time required by researchers to produce prototypes. This reduction in time would ultimately help to increase the rate at which forensic research results could be produced, thus enabling the field of forensics to respond faster to advances in technology.

This dissertation discusses the development of a forensics platform that contains routines commonly associated with the development of forensics prototypes. The purpose of the intended platform is to supply a rich cross-platform forensics library that allows researchers to develop prototypes in a shorter amount of time, while requiring less resources and effort on behalf of the prototype developer.

1.3 Methodology

A good plan is like a road map: it shows the final destination and usually the best way to get there - H. Stanley Judd.

The previous section described the real-world problem of forensics prototyping that needs to be solved. To solve the stated problem efficiently, a simple methodology, or road map, is required. This ensures that key aspects surrounding the development of a solution to the problem are taken into account, in order to produce an ultimate solution that elegantly solves the defined problem.

A platform prototype was developed according to the specifications identified in this dissertation. The intention was to prove the potential usefulness of the platform in the research prototyping process. Forensic research prototypes were constructed using the platform to prove that its use can save precious development time while minimizing resource consumption.

A methodology was defined to ensure the delivery of a solution to solve the stated problem. The defined methodology consists of three simple steps, namely:

1. Perform a literature study to identify key aspects that will influence the prototype design.
2. Develop the prototype.
3. Prove that the developed prototype is of value to the research community by developing research prototypes using the platform.

1.4 Dissertation chapters

The impact of the defined methodology is noticeable when inspecting the content of each of the chapters in this dissertation. Each set of chapters focuses on one of the three steps in the three-step methodology. This is done to ensure that every chapter makes a valuable contribution to the development of the prototype.

The dissertation can be summarized as follows: chapters 2, 3 and 4 report on literature studies that were used to extract requirements for the forensics platform, using sources from the academic as well as the commercial world. These chapters introduce the concepts of digital forensics, open-source software development and digital forensics tool kits.

Chapters 5, 6 and 7 focus on the development of the platform architecture, the platform design and the implementation thereof. Various design and implementation issues are addressed that influenced the development of the research prototype. The final prototype and its features are also discussed to allow the reader to gain perspective on various aspects surrounding the physical platform prototype. The platform can be downloaded and used by researchers around the world.

Chapters 8, 9 and 10 serve as proof-of-concept chapters that report on research that was conducted using the platform. This is an attempt to prove that the platform can be used to produce research prototypes of relatively high quality, requiring less time and utilizing fewer resources. Chapter 11 concludes this dissertation by presenting a summary of the work discussed in all the chapters in this dissertation.

2 Introduction to digital forensics

2.1 Introduction

Forensic investigation is a complex and time-consuming task which requires investigators that are extremely skilled in multiple fields. Knowledge in the fields of information security, penetration testing, reverse engineering, programming and behaviour profiling are just a few of the sought-after skills that are considered invaluable in the field of digital forensics [18].

Even with the help of extremely skilled and highly competent individuals, it may still be very difficult for forensic examiners to form a picture of a crime that has been perpetrated using a computer, largely due to inconsistencies and a lack of incriminating evidence. In order to understand the problems associated with digital investigations and evidence processing, a study of the digital forensics investigation process is required. The study should reveal knowledge that may be crucial in the development of the proposed forensics platform.

This chapter serves as an introduction to digital forensics. Various topics, such as certainty, the Daubert standard and the use of forensic tools are discussed in an attempt to uncover key issues associated with the digital investigation process.

The rest of this chapter is structured as follows: Section 2.2 discusses the different phases that are likely to exist in any digital forensics investigation methodology. Section 2.3 discusses the need for an automated solution to process forensic data. Section 2.4 deals with certainty and methods that can be used to improve the level of certainty assigned to an evidence source. Sections 2.5 and 2.6 contribute to this discussion by elaborating on the characteristics that a forensics tool must exhibit in order for the results that are produced as a result of its use, to be admissible in court. Section 2.7 discusses the topic of acquisitions. Logical acquisitions, physical acquisitions and remote acquisitions are discussed in detail to allow the reader to understand the difference between each of the respective acquisition methods. Section 2.8 concludes this chapter with a summary of the issues discussed.

2.2 Digital forensics phases

Digital evidence collected at a crime scene has to be analyzed in an attempt to identify possible connections between the recovered information, physical entities and physical events. This is obviously a task of great proportions that consumes a considerable amount of time. When taking into account the loads of data that need to be processed in a short time in a digital forensics case, it is understandable that a huge backlog on digital evidence processing exists - backlogs have been recorded ranging from a few months to a few years [30].

This is due mainly to the fact that an extremely large amount of evidence needs to be processed in a very limited timeframe, which causes unimaginable delays in processing schedules.

The act of digital event reconstruction requires the skills of specialists with a strong technical background and a diverse range of additional skills [18, 93]. These skills may include reverse engineering, penetration testing, information security management, programming and behaviour profiling [18]. It can therefore be argued that the digital investigation specialist requires the same skills as computer criminals, in order to have a clear understanding of techniques that may be used to commit a crime.

The process of digital forensics may be described as the use of scientific methods to collect, validate, preserve, analyze, interpret and document digital evidence [79]. Yet another definition for digital forensics is the identification, preservation and analysis of digital information in a manner that is legally accepted [66]. These two definitions are similar in the sense that both describe two distinct phases: a phase in which evidence is acquired and a phase in which evidence is analyzed. Digital forensics can therefore be considered as a science that consists of various steps that need to be taken in order to reach a reproducible conclusion from the evidence at hand.

Wang [101] describes a basic methodology for digital forensics, consisting of the following steps:

- Acquire evidence without damaging it.
- Authenticate the recovered evidence.
- Analyze the acquired evidence.

Although the above methodology is very minimalistic, it describes the essence of more developed methodologies, in that evidence must be acquired, authenticated and analyzed, irrespective of the methodology used. This simplistic methodology therefore captures the essence of available methodologies and may serve as a good reference model, due to its simplicity.

2.3 The need for forensic tools

Consider the needle-in-the-haystack problem: imagine the needle to be a piece of digital evidence and the haystack a digital storage device. The problem usually assumes that a single needle is lost in a single haystack. This simplifies matters as the objective is known: find a single needle and stop searching when the needle is found. Nevertheless, finding a needle in a haystack remains a relatively complicated problem.

To make our problem more realistic, some further complexities need to be added. Firstly, it cannot be assumed that a single needle exists: more than one needle may be present in a haystack or no needles may be present at all. A device may conceal various pieces of crucial evidence, or no evidence at all.

Secondly, it cannot be assumed that only one haystack exists; every haystack needs to be located first, before it can be searched. It is unlikely that digital storage devices will simply be handed to investigators by suspects; investigators therefore have to locate digital devices that may contain valuable evidence.

Lastly, imagine the size of each haystack ranging from the size of a small car to the size of a large city. Digital devices may come in all shapes and sizes. Some devices will have a relatively small storage capacity (the size of a small car), while others will have a much larger storage capacity (the size of a large city). Without the help of some kind of automated tool, the search for possible needles in various haystacks of different sizes may prove to be a very difficult task indeed.

The purpose of the example was to illustrate the complexity faced by digital examiners in the execution of their duties. If digital evidence is collected without focus and attention to collection procedures, the situation may arise where criminals may go free due to the immense burden placed on human resources [101].

The problem described above summarizes a problem known as the “audit reduction problem” [28]. In essence, this is a situation in which the over-abundance of information may overwhelm the investigator with seemingly important information, while other critical pieces of information may simply be overlooked due to the fact that their visibility is overshadowed by the presence of thousands of other pieces of seemingly useful information.

A related problem is known as the “quantity problem” [13] which describes the situation where large amounts of data need to be analyzed. The solution to the quantity problem is to make use of data reduction techniques to minimize the amount of work that needs to be done. An example may include ignoring known operating system files in an attempt to decrease the number of files that need to be inspected by the investigator.

The quantity problem can also be alleviated by using automated tools. The manual analysis of hard drive images with sizes in gigabytes, is simply not feasible [3]. For this reason it seems to be a good idea to introduce a tool to perform parts of the analysis automatically while shielding investigators from unnecessary details. Investigators need to be trained to make effective use of such tools.

As noted by Moore [75], an important constraint on digital forensics is not technology, but economics. The employment and training of investigators places a financial burden on the institute that performs investigations. These institutes typically have a fixed annual budget that allows the employment of a limited number of investigators.

This situation may be considered the cause of current backlogs in digital forensics, which range in excess of months to years [30]. In an attempt to solve these problems, some form of automated processing must be introduced to alleviate the burden placed on the shoulders of digital investigators.

2.4 Certainty

Some evidence sources at a crime scene typically add more value to an investigation than others. This is due largely to the fact that some evidence sources are more likely to contain indubitable pieces of digital evidence, while other evidence sources may contain questionable pieces of evidence at best. This section discusses the concept of certainty as applied to digital evidence sources.

A person walking on a beach will leave a trail of footprints behind as a result of every single step that was taken. Although a large part of the trail may be washed away by the incoming tide, some footprints will remain. The trail of footprints left behind by the tide may be used by a tracker to determine various factors about the person, such as direction of travel, travelling speed and possibly the height and weight of the person walking on the beach. The information captured by the tracker could be used at a later point in time to locate and identify the person in question.

A computer-related crime is similar to the example of a person walking on a beach in respect of a trail of footprints that is left behind. Even though the perpetrator of a crime may succeed in removing a large part of the digital evidence contained in log files and other sources of digital evidence, some evidence may remain due to negligence, lack of time or lack of access rights [92].

It can therefore be argued that the perpetrator of a computer-related crime is likely to leave one or more pieces of evidence behind, irrespective of how minute or insignificant they may be. Unfortunately there is no guarantee that investigators will be able to trust the discovered evidence, since a level of uncertainty is associated with it [21].

This statement is based on the fact that if a crime scene should be reconstructed by making use of only the forensic information captured at the scene, a slight variation from the evidence collected at the original scene may be detected due to inconsistencies introduced as a result of the handling and processing of digital evidence. The only way to actually increase the certainty, and therefore the perceived usefulness of such findings, would be to identify and correlate other sources of evidence with the evidence in question.

Considering the extent of external factors that may have an impact on collected digital evidence, such as modification, misrepresentation or impersonation, it is in fact extremely difficult to prove the guilt of a suspect. The fact that digital examiners have to inspect and collect data and perform the difficult task of piecing together sections of a digital puzzle, makes the digital forensic process time consuming and resource intensive.

Casey [21] created a scale that may be used to determine the level of certainty associated with digital evidence. The scale consists of seven levels named C0 to C6 respectively. At the C0 level the evidence contradicts known facts, while evidence at the C6 level is assumed to be tamperproof and unquestionable. Casey's scale works on the principle that evidence that can be

correlated and verified with other evidence sources is more likely to be valid than evidence that cannot be correlated and verified. Casey's scale may therefore be used as a tool to determine the forensic value of evidence according to its awarded level of certainty.

2.5 The Daubert standard

Digital evidence cannot simply be extracted using any tool or technique due to the fact that some tools and techniques are not meant to be used in forensic environments. Evidence produced using such tools and techniques may be frowned upon by the scientific community and may even be declared as inadmissible in court.

A question that comes to mind is how to know if a tool or technique can safely be used in an investigation without compromising the integrity of the evidence in question. This section considers two well-known standards used to determine whether or not evidence should be admissible in court, namely the Frye test and the Daubert standard.

In 1923 the United States appellate court ruled that results produced by an early form of the polygraph test were inadmissible in court, due to its experimental nature at that point in time [59]. The Frye test was introduced (named after the Frye vs. United States, 1923 case) to help determine if evidence should be admissible in court. According to the Frye test, evidence can be declared as admissible if it was collected using scientific principles that have gained general acceptance in the scientific community [59]. The Frye test therefore places the burden of the evaluation of the acceptability of a technique on the scientific community and peer-reviewed journals [14]. The Frye test was used for a time period of 60 year until it was replaced by the Daubert standard in 1993 [59].

In 1993 the Daubert standard was introduced (named after the Daubert vs. Merrel Dow Pharmaceuticals Inc., 1993 case). The Daubert standard consists of four tests that a scientific technique should pass in order to allow evidence produced by the technique to be declared as admissible. The four tests require four characteristics to be present in the techniques used to produce evidence. The characteristics can be described as follows [14, 52, 86]:

- Has the theory or technique been tested?
- Has the theory or technique been peer-reviewed?
- Is the rate of error experienced with the theory or technique known?
- Has the theory or technique been accepted by the scientific community?

By inspecting the four tests it is evident that the Daubert standard may be seen as an extension or improvement on the Frye test. Both techniques rely on the fact that a technique should be accepted by the scientific community. In the

Daubert vs. Merrel case it was determined that the acceptance of a technique by the scientific community is not enough; therefore additional safeguards were introduced.

Although the two techniques discussed originated in the United States, they are relevant in a South African context. The admissibility tests discussed, in particular the Daubert test, serve as good guidelines in the evaluation of scientific techniques in terms of admissibility.

2.6 Digital investigation tools

Computer-driven evidence is admissible in court providing that the process and/or system(s) used to produce the evidence are known to produce reliable results [52]. According to Wang [101], internet criminal evidence must meet the following requirements to enhance reliability:

- It must be produced, maintained and used in a normal environment.
- It must be authenticated using standard authentication methods.
- It must meet the so-called “best evidence rule” which states that the best available evidence should be presented in court and may not be substituted for with evidence of lower quality.

Digital evidence may not be modified or damaged during any part of the investigation process. Hash sums should be calculated on extracted evidence data, as well as the source of the evidence and compared, to ensure the authenticity and integrity of the data [3]. The best evidence rule may be linked to the availability of data: the best evidence should be available for use in court. Violating of the guidelines specified by Wang may have a negative impact on the reliability of the evidence at hand.

The rate at which computer hardware and software develop exceeds by far the rate at which digital forensics techniques are being developed. In order to improve the situation, some investigators develop their own tools to help in the investigation process. These noble attempts may be rewarded with scepticism in court and it may be difficult to convince the court that the results produced by such developed software can be considered to be reliable [101].

Evidence generated by standard software is easier to admit than evidence generated using custom software for the simple reason that the capabilities of standard commercial software are well-known and unlikely to change [52]. Since commercial software cannot simply be adapted, as is the case for custom software, it is assumed that results produced through the use of commercial software will be more reliable. Chapter 4 presents an in-depth discussion on this topic.

2.7 Forensic acquisitions

Forensic tools acquire data from devices using one of two techniques, namely physical or logical acquisitions [51]. Physical acquisition performs a bit-by-bit copy of the device in question to effectively construct a duplicate image of the data contained on the drive. Physical acquisitions typically operate at a low level of abstraction and perform little translation of the acquired data. Logical acquisitions operate at a higher layer of abstraction and typically copy logical storage units, such as files, to a secure destination.

Physical acquisitions are preferred to logical acquisitions, as an image captured through physical acquisition represents the particular state of a device at a specific point in time. The use of the logical acquisition technique may cause abstraction errors at a later stage in the evidence analysis process. Chapter 7 of this dissertation presents a description of abstraction errors.

Acquisition tools may be able to acquire locally available evidence sources as well as those available through a network infrastructure. The latter case, in which data is acquired over a network, is called a remote acquisition. Remote acquisition is a complex process that can only be performed under certain constraints. Jansen and Ayers [51] describe these constraints as follows:

- The original evidence may only be accessed in read-only mode.
- The remote operating system cannot be trusted.
- Forensic tools may not overload any part of the remote operating system.
- The remote operating system needs to be protected from unauthorized access.

The constraints under which remote acquisitions may be performed indicate that the process is complex, involving various factors that can have a negative influence on the validity of acquisitioned data.

As an example, consider the constraint which states that the remote operating system cannot be trusted. This constraint implies that none of the services supplied by the remote operating system can be trusted, not even services critical to remote acquisitions, such as communication functionality. This implies that the developer of a remote acquisition tool will not only have to develop functionality to access the source of evidence without making use of standard operating system procedures, but communication functionality, such as the development of a TCP/IP stack, will have to be included as part of such a tool, in order for it to function fully in such an environment. Failure to achieve this implies that the non-trusted operating system will have to be trusted to some extent, which may have a negative impact on the perceived validity of captured data.

2.8 Conclusion

This chapter serves as an introduction to digital forensics. Various topics surrounding the field of digital forensics were discussed, such as the levels of certainty awarded to evidence sources and phases in digital forensics. This was done in an attempt to allow the reader to understand the fragile nature of digital evidence and the need for a well-defined process to perform digital investigations.

The need for automated tools in a digital investigation was also discussed. The Frye test and the Daubert standard were discussed, with a brief historical overview of their emergence. This was done in an attempt to determine which special requirements are imposed on investigation tools in order to allow them to produce evidence that is considered to be admissible in a court of law. The differences between various acquisition techniques, such as logical, physical and remote acquisitions were also discussed in an attempt to differentiate between these three well-known forensic acquisition techniques. This introductory chapter contains concepts that are frequently referred to in the coming chapters, in order to clarify issues that are pertinent in the field of digital forensic investigation.

3 Open-source Concepts

3.1 Introduction

Eric Raymon best described the development process of open-source software in his influential book entitled *The Cathedral and the Bazaar* [83]. Raymond compared the development of open-source applications to a crowded and seemingly disorganized bazaar of differing agendas and approaches which by a succession of miracles produces seemingly stable software products. Raymond's comparison accurately reflects the open-source development process, as volunteers all around the word usually contribute small bits and pieces of a project to obtain a fully-functional working application.

Raymond is famous for a comment on the capability of the open-source development model to aid in the discovery of bugs, known as Linus's law, which states: "given enough eyeballs, all bugs are shallow" [83]. Linus's law describes the ability of programmers working together in a massive collaboration effort to discover and fix programming errors. Linus's law is probably one of the key motivators for the development and use of open-source products, as developers all around the world can contribute to and improve the products that they use in common. Although Linus's law is not necessarily applicable to all open-source projects, it has materialized in immensely popular and successful open-source projects, such as the development of Linux and the Apache web server.

Brookes [6] described in his iconic book, *The Mythical Man Month*, that some tasks are sequential by nature. He uses the metaphor of a woman taking nine months to bear a child, irrespective of how many additional women might be involved in an attempt to speed up the process. Brooks describes debugging of software applications as one such task that is sequential in nature. The debugging of an application may be a time consuming and expensive exercise for a development team to undergo, and it is not always possible to commence with development until known design or implementation issues have been resolved.

Brooke's work places a boundary on the amount of tasks that can be performed in parallel due to the sequential nature of various tasks. The Open-source development model should not be regarded as a solution to speed up software development as the development acceleration promised by Linus's law may not always be possible, due to the nature of various development tasks. This chapter describes the open-source phenomenon, how it is different from the traditional closed-source development model, and its impact on system security. The study is conducted to gain a clearer understanding of the open-source development model as work discussed in future chapters of this dissertation will involve open-source software and concepts.

The rest of this chapter is structured as follows. Section 3.2 discusses various open-source principles surrounding the open-source philosophy, including the topic of freedom applied to software. Section 3.3 discusses various open-source license types and the differences that exist among them. Popular licenses

such as the GPL, LGPL and BSD licenses are discussed, amongst others. Section 3.4 compares open-source with closed-source solutions. Section 3.5 describes the fitness of open-source development applied in the field of security. A discussion around whether the openness of a security application's source code is a strength or weakness is conducted, to determine whether or not the development of open-source security applications could be of value to the security community. The problems of open-source modification control, as well as a solution, are discussed in section 3.6. The chapter concludes with section 3.7 that presents a conclusion to the chapter.

3.2 Open-source principles

The Free Software Foundation (FSF) was founded by Richard Stallman as a non-profit organization committed to the development of free software [64]. The FSF pioneered the concept of free software in the 1980s and is still considered to be a major force behind the open-source movement. The foundation has made a tremendous contribution to the development of open-source ideas and principles.

Richard Stallman claims that “free” software is a matter of liberty, not price. Free software is not concerned with the price of software, but rather with the freedoms that are granted to its users [108]. These freedoms include [64]:

- The use of the software for any suitable purpose;
- Free redistribution of the application's source code;
- Freedom to study an application's source code;
- Improvement of the software for a particular purpose.

The principles defined by the FSF serve as a basis for open-source development strategies, regardless of the software license used. However, some open-source licenses may be very restrictive to both the developer of the software as well as the user thereof. Such restrictive licenses are therefore not compatible with the principles of the FSF.

To ensure that the freedoms identified by the FSF are honoured by the various available open-source licenses, the Open-Source Initiative (OSI) was created [64]. The purpose of the OSI is to review open-source licenses to determine whether or not they conform to the open-source definition (OSD). The OSD is a set of principles to which an open-source license must conform to be OSI approved. The principles specified by the OSD are based on the freedoms defined by the Free Software Foundation. Any open-source license certified as OSD compliant therefore allows the users of the software certain freedoms associated with free software.

3.3 Open-source licenses

Various open-source licenses have been defined; some are OSD compliant while other, less pure open-source licenses choose to ignore the basic principles and freedoms defined by the OSD. This section discusses a few well-known open-source licenses in an attempt to determine their respective advantages and disadvantages. The following licenses are discussed:

- GPL;
- LGPL;
- MIT;
- BSD;
- QPL;
- Artistic license.

3.3.1 GPL

The GNU General Public License (GPL) [39] was created by Richard Stallman in 1989 [64]. The GPL is a copy-left license, meaning that the same freedoms granted to the developers and users of an original piece of work should also experienced by the users and developers of derived works.

The GPL ensures the openness of source code by requiring that all code that makes use of GPL-licensed code must be placed under GPL as well. This characteristic of GPL has earned the license the nickname,"the virus license" due to the fact that any code that makes use of GPL-based code is automatically "infected" by the GPL license.

This characteristic of the GPL license may be attractive to true open-source activists, but it can be a hindrance to the inclusion of GPL-based code in commercial projects as companies prefer to protect their intellectual property by distributing binaries (without source code) as their product offerings. The inclusion of source code with their binary distributions may endanger their intellectual property. Thus management may view the inclusion of GPL-based code as pressure to expose the software offering's source code (and therefore a threat to the company's intellectual property).

3.3.2 LGPL

The Library GPL license (LGPL) [40] is a less restrictive form of the GPL. The purpose of the LGPL is to allow developers to use libraries published under the LGPL license in commercial application development, without requiring the commercial product to adapt its license [26].

The LGPL is also seen as a copy left license, but it does not impose the copy left restrictions on applications that use code published under LGPL licensing. The LGPL license does therefore not inherit the "virus" characteristic associated with the GPL license, thus making it possible for commercial applications to include libraries published under the LGPL license without

endangering any of the commercial software offering's intellectual property.

3.3.3 MIT

The MIT license [65] originated at the Massachusetts Institute of Technology (MIT). The MIT license is a permissive and free license which states that the work published under the license may be used in commercial work as long as the original license is distributed with the software.

The MIT license is therefore a non-restrictive license which allows code published under the license to be used in commercial products without having to change the license under which the commercial product is published. Code published under the MIT license could therefore be included in commercial products without endangering the product's intellectual property.

3.3.4 BSD

The BSD license [98] was created at the University Of California, Berkeley as the license under which original Berkeley Software Distribution (BSD) UNIX was published. The BSD license is similar to the MIT license due to the fact that it is a non-restrictive license that requires that a copy of the license is distributed with copies of the derived works. The BSD license contains three conditions [64, 98], namely:

- The distributed source code should retain its original copyright notice.
- Any binary distribution should also contain the original copyright messages, disclaimers and conditions under which the source has been published.
- The names of the developers of the source code may not be used to promote a product.

3.3.5 QPL

The Q Public License [95] was created by Trolltech as a non-copyleft free software license for its free edition of QT Toolkit. The QPL was used until QT version 3.0 but was later replaced by the GNU GPL version 2 licenses as the QPL was not compatible with the GPL license. This means that it is not legally possible to distribute code published under the QPL which makes use of sections of code published under the GPL.

The QPL is also considered to be an inconvenience as modified sources may only be distributed as patches [37]. All the factors combined makes the QPL a very unattractive licensing option from a developer's point of view. It can therefore be considered a wise decision by Trolltech to abandon the QPL in favour of the GPL.

3.3.6 Artistic license

The Artistic license was created by Larry Wall as the license under which Perl source code is published [100]. The Artistic license is similar to the GNU GPL, but distribution of any derived works under the same license is not required [108]. The Artistic license is often criticized for being rather vague; it is therefore recommended that the Artistic license is not used on its own, but should be used in conjunction with the GPL license.

3.4 Open vs. closed source solutions

The goal of any organization is to work smarter by utilizing fewer resources in an attempt to get a product on the market faster and more efficiently [67]. To achieve this goal, organizations have to make use of technological resources to increase the efficiency of their operations. Various hardware and software solutions may be used to achieve this feat.

Software-based solutions may consist of closed-source components, open-source components or both. This section discusses the advantages and disadvantages of closed-source and open-source software development models.

The open-source development model allows developers all over the world to contribute to various open-source projects [94]. The contributors to open-source projects also tend to constitute a project's primary user base [91]. This implies that the requirements of the developed features are likely to be understood by the programmers who implemented them, which may tend to rule out situations in which an application does not conform to its stated requirements.

This may not always be the case with the commercial development model, in which programmers tend to receive instructions from analysts explaining how to solve a problem. The programmer may not necessarily have extended knowledge regarding the domain in which the developed application operates. This may lead to a situation in which instructions that may be unclear or incorrect may have a negative impact on the correctness and quality of the project, as the programmer does not fully comprehend how to solve the complex problem.

Open-source software may also have an advantage from a code inspection point-of-view as an open-source project's source code is publicly available for anyone to inspect, modify and redistribute [25, 47, 63]. Unfortunately this quality may serve as a double-edged sword as the full disclosure of source code may be used by an attacker to find and exploit weaknesses in a system [14, 34]. The openness of source code can therefore be viewed as being both good and bad, as it can be either beneficial or harmful under certain circumstances.

Another problem with open-source that should be taken into account is that the mere fact that source code is available does not guarantee that anyone will actually take the time to identify and solve problems. Open-source projects

may consist of thousands to millions of lines of code. The likelihood that a code inspector can simply detect and repair programming errors through inspection dramatically decreases as the number of lines of source code increases. It can therefore be argued that the availability of an application's source code does not guarantee its correctness.

Only a small handful of people will have access to the source code of a closed-source product [34]. This creates the assumption that closed-source applications are more likely to be authentic as opposed to open-source applications in which the source code of the application is accessible and modifiable. It is also assumed that since closed-source applications are more difficult to modify, they are more likely to be authentic [52].

Closed source systems tend to be inspected by a closed group of individuals [47]. Although this situation is not necessarily problematic in all conditions, it may lead to situations in which the testing team simply misses program errors due to time constraints or other development factors.

The assumption is usually made that it is not possible to inspect a closed-source application for programming errors. From a legal perspective this statement is usually true, as most closed-source applications have a clause in their End User license Agreement (EULA) indicating that it is considered illegal to reverse-engineer, decompile or disassemble the application in question.

From a technical perspective it is actually possible to perform the inspection, given that the inspector has enough time and is familiar with reverse-engineering techniques. Various disassembler tools may be used to inspect the assembly code generated by the compiler from high-level source files. It can therefore be concluded that it would in fact be technically possible to inspect closed-source applications for programming errors.

This implies that the openness of the source code in open-source applications is not such a big advantage as previously thought from the point of view of source code inspection. Although the availability of source code enhances the user's ability to inspect the code [34], code may still be inspected, even without having the applicable source code available.

Another distinction between open-source applications and closed-source applications is the software development timeline. Commercial applications tend to have strict development timelines that developers must adhere to [24]. This may create a situation in which features are not properly implemented due to the lack of time available to developers [46].

Open-source developers do not have to work under the same time constraints as commercial application developers, which implies that they could potentially have more time to spend on the development of features [24]. One of the side effects of open-source development is the fact that it may take a considerable amount of time before a new feature has been implemented by open-source developers. This is due to the fact that there are usually no schedule constraints that developers must adhere to, since common open-source licenses typically do not force open-source developers to commit to a timeline. It

should also be noted that the relaxed time constraints experienced by open-source developers do not necessarily guarantee the quality of the produced code.

Closed-source commercial tool vendors usually supply their customers with manuals, online tutorials, white papers and technical support [63]. This level of support supplied by commercial software vendors is considered to be of higher quality and therefore better value than the level of support offered by the developers of open-source solutions. This situation is understandable, as customers pay the developers of commercial software for the privilege to use their software and consequently to be able to access help and support.

From a business perspective the support provided by software vendors may be extremely valuable. In a situation in which a production-stopping bug is detected, the software customer will simply inform the vendor about the problem. The vendor will respond by fixing the problem which means that the software customer will be able to continue to use the software within a defined period of time.

This may not be the case with open-source software as no contractual agreement is likely to exist between the developers of open-source applications and the users of the software [42]. This means that the developers of the open-source application are not required to perform program maintenance or create bug fixes, which essentially implies that a business may risk losing money if it uses software without proper technical support.

Furthermore, the developers of open-source software rarely receive any financial incentive from the users of their software. The situation may therefore be summarized by saying that the users of software will receive the level of support that they pay for - commercial software with a hefty price tag will come with better support, while open-source software with little upfront cost will have very limited support.

This is the key selling point for commercial vendors - the initial cost of a commercial product may be much higher than the price of an open-source product, but the risks associated with the use of commercial software packages are minimized by the provision of support [42].

Another downside to open-source software is that it tends to be more difficult to use than commercial variations. Software that is difficult to use typically requires the expertise of a highly skilled investigator to operate it. This is not considered to be a bad thing from a forensics perspective, as a skilled investigator is more likely to uncover critical evidence. Unfortunately a skilled investigator may be relatively expensive to appoint. Training costs may also be relatively high, as it would take the investigator a longer period of time to master a piece of software that is relatively difficult to use. The total cost of ownership of open-source applications may therefore turn out to be higher than that associated with commercial solutions [46].

The total amount of man hours spent on an investigation is not influenced by software alone, but rather by a combination of factors. It can therefore be

argued that the software used may be one of the contributing factors which determine the time that it takes to perform an investigation. Skilled investigators may actually claim to be able to perform investigations faster using freely available software compared to commercial applications. It should be noted that an expensive price tag of a software product alone should not be the determining factor when acquiring software. A bill of several thousand dollars for a piece of software may seem expensive, but if the advantages of the use of such software are considered in a forensic investigation setting, it may be worthwhile to invest in such software if it enables an investigator to produce results more efficiently. Tools should be acquired that are suited to the skill level of the investigator, in order to obtain the most value from their application and use.

Code inspection is facilitated by the open-source development model [34, 63] which basically means that the code is open for anyone to inspect in an attempt to spot design and implementation flaws. It can therefore be argued that it would be very difficult to hide an open-source application's history of bugs [14]. Determining the quality of a specific open-source package can therefore be facilitated by code inspections and a study of well-known program issues.

It may be more difficult to determine the error rate experienced with various commercial packages as the fear of losing sales may persuade software vendors not to release their error rate figures [14]. By studying the standard disclaimer shipped with most copies of commercial software, it should be clear that commercial software may not be as reliable as it would appear [52]. Some sort of mechanism is therefore required to help determine the quality of commercial software.

At this moment in time the problem is solved by using available market share metric figures [14, 52]. The idea behind the concept of quality estimation based on market share metrics is that reliable software tends to be used more than non-reliable software. The most widely-used software is therefore perceived to be of highest quality. However, the use of market share metric figures may not always be a good idea, as software with a well-designed GUI, good marketing and low quality code is likely to sell better than a high-quality application with little or no marketing and a less intuitive GUI [52].

It should be noted that software that is easy to use is likely to be perceived by the user as being of high quality which may not necessarily the case. A program with a good user-interface and large market share may be hiding thousands of potential bugs.

3.5 Open-source security

Cowan [29] describes the least damaging software vulnerability as the one that never exists. From a security perspective this statement is convincing as a vulnerability that does not exist cannot be exploited by an attacker. Cowan describes three actions that may be used to ensure that damage caused as a result of software vulnerabilities are minimized:

- Software auditing;
- Vulnerability mitigation;
- Behaviour management.

Software auditing is concerned with the detection of vulnerabilities before a product is released. This is done in an attempt to locate and eliminate vulnerabilities before they become a threat to the security of a system. One of the problems associated with software auditing is to find highly skilled individuals that actually possess the capability to detect and correct program errors [29].

Such an auditor needs to understand the requirements, design and implementation of the code in question in a very limited period of time. These skills are relatively scarce and those individuals that have the capability to perform such audits are usually experienced and therefore relatively expensive.

Vulnerability mitigation describes the use of compile-time techniques in an attempt to remove the effects of existing vulnerabilities from source code by adding additional run-time checks to the code. Vulnerability mitigation may, for example, be used to add range checking code to software that may be susceptible to buffer-overflow attacks, in an attempt to mitigate the effects that a buffer-overflow attack may have on a susceptible system.

Behaviour management describes the mechanisms employed by the operating system to control the execution of an application. The goal of behaviour management is to prevent damage to a system by revoking an application's right to perform dangerous operations. Behaviour management can therefore be used to minimize the risk of running a potentially exploitable target executable file by controlling the damage that the executable can do.

Open-source security depends on the notion that security experts around the world actually inspect applications for errors [47]. As discussed previously, although the availability of an application's source code makes it easier for an expert to inspect the code, it unfortunately does not guarantee inspection. As discussed by Cowan [29], the openness of open-source application code grants both the attacker and the defender greater analytical power than may be experienced with closed-source applications.

Both the attacker and defender have the same advantage: the characteristic openness of the code. If both the attacker and defender manage to find a program error and the defender chooses not to correct the problem, the advantage lies with the attacker as he will have the opportunity to exploit the program error.

Some closed-source advocates argue that the availability of source code may empower hackers to easily defeat security mechanisms built into open-source application [60]. This statement is not necessarily true, as closed-source systems can also be inspected by attackers. Surveys have shown that most defaced web pages are served by Microsoft IIS based servers [47].

Considering that the Apache web server is more widely used than IIS, it should be clear that closed-source software does not have an advantage over open-source applications from a security point of view in this particular example. This is due to the fact that it is possible to detect and exploit program errors even in closed-source applications, when enough time and energy are invested by the attacker.

Unfortunately a problem exists with the use of closed-source tools in a forensic environment. To inspect a closed-source application's source code would be impossible for anyone not involved with the development of the application. The Daubert standard introduces a dilemma for commercial software as it is difficult, if not impossible, to determine how reliable the results of an application are without inspecting its source code [52].

It is difficult, if not impossible, to determine whether or not an application is compliant with the Daubert standard without physically inspecting the application's source code. It can therefore be argued that the Daubert Standard can actually help to promote the use of open-source applications in forensic settings.

The counter argument is that it would be possible for a standards committee to physically inspect the code of a closed-source commercial application in a controlled environment, to determine whether or not it is Daubert Standard compliant. It should therefore be noted that the Daubert Standard does not necessarily disapprove of the use of closed-source commercial software in forensic settings, as might have been expected.

3.6 Open-source modification management

Open-source applications may potentially be modified and redistributed by anyone that possesses the required skills to perform the required modifications. This situation creates a problem, as malicious users may potentially modify an application's source code to make it more susceptible to certain kinds of attacks. These modifications may potentially be concealed due to the fact that an open-source project could easily consist of hundreds of thousands of lines of code. The inspection of a large open-source project for malicious changes could consume a considerable amount of resources which may have been put to better use elsewhere. This section focuses on the identification of a method that will help to control the extent of changes that can be made to an open-source project.

Kenneally [52] states that commercial software is typically called “standard software”, whilst open-source systems are known as “custom software”. The fact that commercial software is considered to be standard software relates to the fact that the same standard version of an application is used by people all over the world (assuming version numbers are the same and external modifications did not occur). The copy used by one person is therefore the same as the copy used by all the other software users using that version.

This is not necessarily the case with open-source software due to the fact

that potentially thousands of variations of an application may be in use all over the world. This can cause problems as inputs provided to different variations of the same application may potentially produce different results. This characteristic of open-source applications becomes a problem from a forensics point of view, since forensic applications need to be deterministic in the sense that they should always produce the same output for a given input set. If the likelihood of modification of open-source applications could be restricted in some manner, it could be argued that software produced under the imposed restrictions may also be seen as “standard software”, since the likelihood of it being modified is controlled. A complex solution may be devised to detect and prevent modifications from being made to open-source tools.

Occam's Razor describes a situation in which the simplest hypothesis is usually the best to explain a phenomenon [27]. When applying Occam's Razor to the context of open-source modification control, it can be argued that a much simpler solution is likely to produce results that may be just as good, or even better, than the results produced by a complex solution. A simple solution therefore needs to be identified and investigated in order to solve this challenging problem.

As discussed previously, the main difference between a closed-source application and its open-source counterpart is that the latter can be modified easily, which would in turn lead to a non-standard copy of the software. This situation may be prevented if the open-source software distribution model is adapted to be similar to the closed-source distribution model. More specifically, it may be possible to create standard copies of open-source applications through creating pre-built binaries by a trusted party.

This model has been used by open-source distributors, such as Redhat Inc. [84] for years. The advantage of this approach is that it allows one trusted party to create a standard copy of a product. All users of the standard software will have the same version with the same behaviour, features and bugs. It can therefore be argued that it may in fact be of advantage to the users of open-source software to use standard software packages as the behaviour of these standard packages may be documented and well-defined.

Consider the example of a web server, such as Apache, distributed as a pre-compiled binary by a trusted party. Due to the fact that the binary is used on potentially thousands of computers by technically experienced individuals, it can be argued that they will send bug reports to the distributor of the binaries, when a program fault is detected. The distributor can also obtain bug reports from the developers of the software and determine whether or not the bug is present in the binary package.

Bug reports associated with a standard package can therefore be published to allow the users of the software to devise workarounds to overcome problems associated with programming errors. Users of non-standard software may not necessarily have this luxury as they may have to test their non-standard application copies themselves when a new bug report is released to determine

whether or not their version of the application is affected.

A more concrete example may help to explain this scenario better. Consider the well-known Linux distribution named Fedora. Each version of Fedora is shipped with thousands of pre-compiled packages. Each of these packages is maintained by members of the Fedora project. An out-of-the-box Fedora installation comes standard with a software update manager that can be configured to check for software updates on a daily or hourly basis. Every time the update manager is executed, it connects to the Fedora application repository and determines which of the applications currently installed on the user's system needs to be updated. The user is allowed to select a few packages that need to be downloaded from the repository, which the software update manager will fetch and install. By default the software update manager checks for updates once every day.

The value of the update manager lies in the fact that the system administrator does not have to manually check for updates for every single piece of software installed on the system; this is done automatically by the package management software. In the absence of such an update manager, the system administrator would have to perform this tedious task for almost every program that was downloaded and built from source, to ensure that the system stays up to date. The presence of such a software utility for a standard system therefore helps to keep it up to date, while simultaneously saving administration time.

A reason why the use of standard software may be attractive in the field of forensics is the fact that a standard binary, which was verified through hashing to be the same binary distributed by the trusted party, could not have been modified to introduce additional behaviour. The net effect is that the same tool is likely to produce the same results when applied to a fixed data set, which means that the results would be reproducible. The behaviour of standard binaries is deterministic and therefore attractive in the field of digital forensics.

The approach described to standardize open-source software also has a few drawbacks. One of the most notable issues is portability. Programs are compiled to run on specific operating system platforms and processor architecture. Open-source applications have been known to be extremely portable and able to run on almost any platform that can compile or interpret the project's source code, assuming that no platform-specific dependencies exist in the code. The act of pre-packaging compiled solutions implies that the compiled binary is produced for a specific platform. A package would therefore have to be created for every single platform and architecture that needs to be supported.

Another problem associated with pre-packaged solutions is that they often fall behind in terms of updates. It may be a question of days, months or even years before cutting-edge source distributions are transformed into pre-packaged binary solutions. There are various reasons why this may happen, which include:

- Binary packages from trusted vendors are usually created from source that has been proven to be stable over time.

- Packages may be produced by users of the software on an ad-hoc basis, which implies that there is no guarantee as to whether or not new updates will be created in the future.
- The package creator may decide to stop producing the package.

From the discussion above it should be clear that pre-packaged solutions may be of value to the distributors of forensic tools as they help to create standard, verifiable tools that will produce repeatable results. Drawbacks of this approach include the portability issues associated with pre-compiled binaries and the fact that packages may not contain the most recent versions of open-source projects. Nevertheless, standardizing open-source software has been adopted in the industry for many years and seems to be a simple solution to a complex problem.

3.7 Conclusion

This chapter has discussed the open-source phenomenon, together with its advantages and disadvantages, as opposed to the closed-source development model. The open-source principles, popular licenses as well as some of the myths surrounding open-source development have been discussed.

For some developers, their preference for open-source or closed-source development models may be based on purely philosophical grounds. Developers may choose to commit their life to the development of free software if they are committed to offering freedom to the users of their software. From a commercial point of view the use of open-source software may save money on licensing fees, but will require more skilled users to operate and maintain the packages, which in turn may have a higher total cost of ownership than easy-to-use commercial applications. The openness of an application's source code may increase the chances that it will be inspected by its user base, but inspection cannot be guaranteed. This means that the quality of a particular open-source application may not necessarily be any better than that of a closed-source commercial application.

It should be clear that no simple answer exists to the complex problem of whether or not to go the open-source route when developing a new project, or when doing research on available systems that can be used in a production environment. Closed-source applications may be preferable in a business context, as they tend to be easier to use than open-source applications. This may result in a lower total cost of ownership than that of open-source applications. Open-source applications tend to be cheaper to obtain, but may require a considerable amount of skill to operate. The use of open-source applications may be of value in situations in which user interaction with an application is minimal, such as client/server based computing, in which a service is provided by an application without the need for user interaction.

There is no easy answer as to whether or not to go the open-source or

closed-source route. A possible solution may be to do a careful requirements study to determine which one of the two models is best suited to a particular development or implementation project.

4 Forensic Tools

4.1 Introduction

Computer forensics can be described as the acquisition, preservation, analysis and presentation of digital evidence [2]. Digital evidence is characterized by its fragile nature; it can easily be altered or destroyed, thus rendering it inadmissible in a court of law [2]. Investigators should therefore take care to ensure that evidence is not destroyed as a result of an ongoing investigation.

One of the most time-consuming tasks in a digital investigation is the search for digital evidence [17]. Various toolkits have been developed that contain enough tools to aid investigators in the process of identifying and evaluating digital evidence on computing devices [61]. The purpose of these toolkits is to automate manual processes as much as possible in an attempt to increase the efficiency of a digital investigation.

Locard's exchange principle [50] states that the perpetrator of a crime will come into contact with the crime scene, introducing something to the scene while taking something from the scene. Although Locard based his exchange principle on the assumption that the crime scene is of a physical nature, it can be argued that the principle is also applicable to digital crime scenes, characterized by their virtual nature.

Consider the example of an intruder breaking into a computer system. The intruder may try to execute various commands on the target system. Every single command that is executed by the attacker will leave some indication of its execution, whether it be a log file entry added by the application, or a timestamp that has changed. It can therefore be assumed that the victim system will show signs of intrusion; the extent of signs left behind will largely be determined by the skills of the attacker. The attacker's system, in turn, may also show signs that it communicated with the compromised system, such as copied files or even log entries.

From this discussion it can be concluded that Locard's principle applied to digital environments can be described as a situation in which an attacker may leave traces of the attack on the target system, while also receiving traces of the attack on the system that was used to perform the attack. Thus digital evidence may be found on both the compromised system as well as on the attacker's computer system. Forensic procedures may be used to extract these traces from the targeted system and the attacker's system in an attempt to find correlating evidence that may help to incriminate a suspect.

This chapter discusses the need for tools that automate the digital evidence extraction process. Topics such as evidence sources and the need for automatic evidence analysis are covered in detail. It is crucial to conduct a study of already-existing tools in an attempt to uncover the core functionality supplied by each. A few well-known forensic tools, namely Encase, FTK and Sleuth Kit

were inspected and their features are summarized.

A feature comparison matrix was constructed in an attempt to identify features commonly associated with investigation tools. The commonly identified functionality served as a foundation on which the work documented in this study was built.

4.2 Digital evidence sources

Sources of digital evidence may be found on almost every digital device that has some form of digital storage capability. Although the evidence source possibilities may seem limited at first, closer inspection of digital media reveals that less obvious sources of digital evidence also exist on most digital devices, providing that enough time and effort is invested in the process of uncovering such sources. This section discusses various sources of digital evidence in more detail.

An operating system is usually designed with efficiency in mind rather than security. This creates a situation where optimizations are performed in favour of speed rather than the core components of security, namely integrity, authenticity and availability of information [82]. According to Moore [75], traces of critical pieces of data may still be active on a system after the removal of an item by a user. This situation is evident in various file system drivers due to the fact that it is more efficient to simply mark a file as being deleted than to erase every single sector used by the file. If a file is simply marked as being deleted, it would in theory be possible to recover the file assuming that the sectors it used have not been claimed by another file created after the deletion of the file in question.

Another security problem introduced through careless programming may be linked to the usage of temporary storage files. As explained by Casey [20], it is possible that some applications with encryption capabilities will decrypt an encrypted file to a temporary location on disk and re-encrypt it again at a later stage, as required. From a user's perspective this situation makes sense as the user would typically want to access an encrypted file, edit it and save the changes back to the original encrypted archive. From a security perspective this method may be dangerous to say the least, as it could potentially render the encryption process useless due to the fact that an unencrypted copy of the file exists or has existed somewhere on disk and could be recovered if required. This is a desirable situation from a forensics perspective as it could potentially allow the examiner to view the content of encrypted files without the need for any encryption keys or passwords.

From this discussion it should be evident that data is generated as a result of actions taken by the user, which may be used later as digital evidence. The digital evidence mentioned in the examples were products of actions taken by a user. Traditional crimes usually yield evidence of a physical nature. This is not

necessarily the case in digital crimes, where evidence may be either tangible or intangible. To understand what the concept of digital evidence entails, it might be of value to understand its associated characteristics.

According to Wang [101], digital evidence has the following characteristics:

- Digital evidence can be copied and modified easily.
- Keeping digital evidence in its original uncompromised state is relatively difficult.
- It is difficult to prove the integrity of the source of information.
- Digital evidence is not easily preceived by the human senses.

It can therefore be concluded that simply presenting data as evidence is not enough, as it can be argued that the data in question could have originated from locations other than the device from which it was acquired. Some rigorous process needs to be introduced to ensure that evidence is collected in a manner that will help to ensure the integrity of the data, without modifying its source.

A concern raised by Wang's definition is the digital evidence perception problem. Information stored on a computer system is optimized for use with digital systems and not for human access. Although some attempts have been made to introduce technologies which allow the storage of information in a manner which makes it more humanly readable, such as XML, the over-abundance of information still poses a great problem for the forensic investigator.

To identify possible evidence sources, a complete and clear understanding of the concept of “digital evidence” is required. Carrier [14] identified three types of digital evidence, namely:

- Inculpatory evidence;
- Exculpatory evidence;
- Evidence of tampering.

Inculpatory evidence describes any evidence that helps to prove a theory regarding a digital event. An example of inculpatory evidence could be a list of passwords found in the possession of a suspect, that were used to gain unauthorized access to a computer system. The presence of such evidence may be a key piece in the puzzle that needs to be solved by investigators.

Exculpatory evidence can be viewed as the opposite of inculpatory evidence - the presence of such evidence contradicts a given theory about a digital event. Consider the previous example: if additional evidence was found that the password list was actually planted by a third party to implicate the suspect in question, it would basically destroy the original theory held by the

investigators.

The last type of evidence may be described as evidence that shows signs of tampering. Such evidence may indicate that an attempt was made by an attacker to hide his/her tracks or implicate an innocent party. The previous example also illustrated this form of evidence - the situation in which evidence was discovered showed that the suspect in question may have been framed, which can be viewed as evidence of tampering.

A direct approach to the collection of digital evidence is to inspect mechanisms which exist primarily for the purpose of event reconstruction. These mechanisms may include audit trails, firewall logs, intrusion detection systems (IDS), and application logs [92]. Such logs are usually a good place to start when collecting evidence, as they are likely to contain signs of foul play (providing that the mechanisms were active at time of interest and that they have not been tampered with) that can be used later to identify the guilty party. Unfortunately a direct approach cannot always be followed since it implies that mechanisms are in place on the system in question to capture event data as events occurs. Other, less direct approaches will therefore also have to be discussed.

Digital devices are designed with different processing and storage capabilities. From a forensics perspective this complicates the acquisition process, as different methods are required to extract evidence from different devices. Needless to say, various device types may contain various types of digital evidence. It is therefore safe to assume that the methods used to extract evidence from various devices such as cellular phones, PDAs, PCs and routers are different; the results obtained are also different, as the information conveyed by the extracted evidence is bound to have been influenced by the primary purpose of the device in question.

As an example, consider the information that could potentially be extracted from a cellular phone. Likely sources of evidence extracted from a cellular phone may include the phone's call records, contact list or inbox. Compared to the information extracted from a device such as a dedicated server, which may contain various request logs, it should be clear that the extracted evidence types will be different. Nevertheless, it may be possible to correlate evidence from various devices to form a clearer image of an event as it occurred.

Consider again the example of the cellular phone and the server: most modern cellular phones have the capability to access the internet using technologies such as GPRS, EDGE, 3G or HSDPA. This allows the user to visit web pages using a browser found on the phone. The browser may cache pages or keep a history of pages visited; if the cache or access history could be matched to access logs stored on a server, it may help to increase the level of certainty associated with the evidence in question [21].

Another source of digital evidence is timestamps; timestamps may be associated with files or contained in other sources of evidence, such as system log files. Timestamps are of value in the forensic investigation process as they

allow the examiner to construct an accurate timeline of events as they occurred, providing that the timestamps have not been tampered with [9].

A system's RAM can also be seen as a potential source of evidence [20]. Using RAM as a source of evidence has the following advantages:

- Applications and data will have been loaded into memory at some point in time, in order to be used. After use they may remain in memory for an undefined period of time.
- Data in memory is unlikely to be encrypted [20].
- Systems with large amounts of RAM may enable memory pages to remain in memory for days or weeks after use, without being replaced [3].
- Runtime information, such as the name of the logged in user, loaded libraries, open sockets and open files are stored in memory [81].

From the summary presented above it would seem that RAM is in fact a great source of evidence. Unfortunately the process of extracting information from RAM is relatively complex. This is due to the fact that memory extraction software cannot be used without consequences. To use an application to extract the main memory requires that the application itself is loaded into memory, thus destroying part of the evidence in question. Fortunately a solution exists to this problem.

A hardware-based solution was developed by Carrier [16] to overcome the problem of memory data destruction, by capturing data in main memory using special hardware. This allows investigators to capture the content of the system memory without destroying any evidence. The use of such a hardware-based solution therefore makes it possible to capture a system's memory content in a forensically sound manner. This would allow investigators to inspect the state of the system's memory at the exact point in time at which the acquisition was performed.

Another source of evidence is the Windows Registry. The Windows registry contains a wealth of information concerning the configuration and everyday use of a Windows-based computer [68]. The purpose of the Windows registry is to supply Windows-based applications with a single storage location that may be used to store application settings; some of these settings may only be found in the registry while a system is active, such as the path to a mounted network drive [68]. It should be noted that the information stored in the Windows registry may in fact be of forensic value, as it is unlikely to be found anywhere else on the system.

Due to the fact that the registry may contain thousands of entries, it can be assumed that the registry database will consume a considerable amount of

disk space. The Windows registry may be seen as a single entity, but its contents may actually be distributed in a binary format across several locations on a Windows system [68].

Accessing entries in the Windows registry is therefore a bit more complicated than simply locating the registry database file and performing text string searches. Registry access should therefore be performed using a special tool, such as a registry editor. Forensic tools thus require special support for the Windows registry to allow investigators to gain access to the registry settings stored on an acquired disk image.

4.3 Forensic tools

Investigators may use a collection of tools to acquire evidence at a crime scene [22]. Tools used by forensic investigators may consist of system utilities, data recovery software, file viewers, port scanners and firewall tools [69]. Before a tool may be used by investigators it needs to be evaluated to determine whether or not it is suitable for forensic purposes.

According to Kenneally [52], reliable tools have the following characteristics, each of which is discussed below:

- Integrity;
- Authenticity;
- Availability;
- Supportability.

Integrity is concerned with the accuracy of the data produced by the system in question. Applications suited for commercial purposes should not create faulty data, which implies that they have a high integrity rating. An application is considered to be authentic if it is an exact copy distributed by the developer and fulfils the purpose stated by the developers. Kenneally [52] adds that data used by the application should come from a source approved by the user. In a forensics context this implies that the correct source of evidence should be used when performing investigations.

Consider the example of a computer containing multiple drives. Imaging software should create a replica of the drive specified by the investigator. If a replica of the incorrect drive is created or parts of the source drive are simply skipped in the replica creation process, the effects on the digital investigation may be catastrophic. It can therefore be concluded that integrity is in fact a valuable characteristic that needs to be present in all forensic tools.

Availability is concerned with the accessibility of data on the system in question. From a software perspective, availability implies that the software does not fail or cause other software components to fail - a very important

characteristic of forensic software. As stated by the NIST reliable disk backup software criteria [62], in case software does fail, the failure should be made visible to the investigator to ensure that he/she is aware of possible complications that may arise as a result thereof.

Supportability is concerned with the level of dependence experienced with the software in question. The level of supportability reflects the competence of the software developers to maximize the level of stability of their software by minimizing software bugs. A high-level of supportability indicates that a software company is capable of producing high-quality software by embracing a process that supports the elimination of software errors and faults as they are detected.

The presence of the characteristics discussed above may help to determine if a tool is suitable for forensic use. The reliability characteristics as defined by [52] are based on the assumption that the software operates in a trusted environment. Although forensic investigators try to control the environment in which evidence analysis and acquisitions are performed, it may not always be possible, as some systems cannot go off-line (see chapter 8 for a discussion of the problem). The investigator may therefore be forced to work in a non-trusted environment for a period of time. To understand this problem and its consequences, a better understanding of the concept of trusted environments is required.

Burmester and Mulholland [10] summarize the concept of trusted environments by specifying a minimum of three characteristics that must be present on a system (such as an operating system or development framework), in order for it to be considered as trusted. These characteristics are:

- Protected capabilities;
- Integrity measurement;
- Integrity reporting.

In essence, these characteristics allow applications to execute in an environment in which the data supplied to the application is exactly what it is supposed to be. Not only should applications be free from interference, but measurement and reporting will help to identify possible interferences. The problem is that operating systems may lack these characteristics, which would in effect classify them as a non-trusted environment.

As an example, consider a simple root kit, which may be present on Windows-based systems, Linux-based systems, as well as systems running various flavours of UNIX. A root kit can in effect manipulate data sent as well as data received from the application in question. The possibility that such a filtering mechanism may be present on an operating system is enough to cast a shadow of a doubt on the evidence acquired using trusted tools in a non-trusted environment. Chapter 8 presents an in-depth discussion of this problem.

The following subsections discuss three well-known forensic toolkits, namely Encase, FTK and the Sleuth Kit. Each of the tools is discussed under the following headings:

- Supported file systems;
- Compression/decryption support;
- Search support;
- Extendibility;
- Hashing;
- Other features.

A feature comparison matrix is presented at the end of the discussion in an attempt to identify common functionality provided by all three toolkits.

4.3.1 Encase

Encase is a commercial forensic investigation toolkit that is widely used within law enforcement circles [45, 49]. Encase is a fully integrated software suite that is capable of assisting investigators with acquisition, analysis and reporting based on acquired evidence sources [45, 90]. According to Jennifer Higdon, spokesperson for Guidance Software, the manufacturer of Encase, Encase is used by about 2000 law enforcement agencies around the world [45]. Encase can therefore be considered to be a well-respected and popular investigation tool that contains the functionality required to perform digital investigations.

Supported file systems

Encase allows the investigator to inspect mirrored hard drive images [45]. The use of a mirror copy of a hard drive image allows investigators to inspect the content of a hard disk without risking the possibility of destroying the integrity of the original evidence source. Although Encase also allows the investigator to access the content of a hard drive directly without the creation of a mirror image, the practice is not recommended as the content of the drive may still be changed due to swap file activity [48].

The following file system types are supported by Encase, as of version 2.0 [48, 54]:

- Fat 12/16/32;
- NTFS;
- EXT2/3;
- HFS/HFS+;
- UFS;

- ReiserFS;
- JFS/JFS2;
- Palm;
- CDFS;
- Joliet;
- UDF;
- ISO9660;
- FFS.

By inspecting the list of supported file systems, it is clear that Encase supports file systems used by basically every popular operating system. Although the information stored by each of the file systems may be different, Encase is able to display the extracted information in a consistent, user-friendly manner [48].

Encase can also detect and rebuild dynamic disks (RAID) through the process of partition mapping [48]. Each of the disks used by the configuration may be searched and inspected separately [48]. This allows the investigator to manually inspect suspicious disks used in dynamic disk configurations.

Compression / decryption support

Encase supports the decompression of compressed archives. This allows investigators to perform searches on files contained within compressed archives. As of Encase version 4, all compressed files are mounted as virtual devices [48]. This allows the investigator to access the compressed data in an easy, intuitive manner.

Encase data decryption support can be enhanced via an add-on module created by Guidance Software called the Encase Decryption Suite (EDS). EDS allows investigators to decrypt files and folders that are stored on Microsoft's EFS file system [48]. This powerful module makes it possible for investigators to inspect evidence that would not otherwise be accessible, due to the cryptography.

Search support

Encase allows the user to search for text strings or patterns contained within evidence sources. Encase is shipped with a list of predefined keywords that may be of interest in an investigation [48]. This allows investigators to easily construct search queries using common keywords that have been known to produce good search results.

Encase uses an advanced search algorithm that allows the investigator to perform searches quickly and efficiently. According to Guidance Software [48], Encase version 4 took about 2 minutes to perform a search for a 15-term keyword on a 1 GB drive, which means that the Encase search algorithm

processes about 8.5 MB a second.

Although not much technical information about their search experiment is available, it can be assumed that the search was performed on a commonly available personal computer using search terms of average length. Although optimized, the search operation is still is relatively expensive in terms of time to perform, as it would take Encase about 10 hours to perform the same search using on a 300 GB hive image (calculated using the processing rate of 8.5 MB/second).

Extendibility

Encase can be extended using a built-in scripting language shipped with it, called EnScript. EnScript is an advanced macro language that allows advanced users to create custom functionality running on the Encase platform [45]. This functionality allows Encase to be extendible as developers are able to create rather sophisticated application scripts that run on the Encase platform.

Hashing

Encase uses the MD5 hashing algorithm to calculate hash values for files under investigation [48]. Encase can use the calculated hash values to identify common files that may or may not be of forensic value. Known files can be identified, even if they were renamed, due to the fact that the checksum calculated by the MD5 hashing algorithm is not dependent on the name of a file. Encase can also make use of external hash databases, such as the HashKeeper database [32] and the National Software Reference Library [76] to help identification of commonly known files.

The file naming convention under Windows allows the user to specify a descriptive name for a file followed by an extension indicating the file type. An investigator who is looking for a list of images stored on a disk may simply search for files with extensions commonly associated with images (*.bmp, *.jpg, *.png etc.). However, this primitive method can easily be fooled when a suspect changes the extension of the image file to some other extension. Encase can compare the signatures of files with the types associated with their extensions to identify files that were renamed in an attempt to conceal their content.

Other features

Encase has built-in support for the acquisition and inspection of Palm OS devices [90]. Encase can mount the image extracted from a Palm OS device as a virtual drive to allow examiners to thoroughly inspect the content of the device. Encase version 4 is known to support the following Palm OS devices [90]:

- Palm IIx;
- Palm IIIxe;
- Palm V;

- Palm Vx;
- Palm m.

Calendar information, contact information and notes can be easily extracted using this capability. This functionality therefore allows investigators to capture personal information from mobile Palm OS-enabled devices.

Encase also supports the analysis of the Windows registry, which is a central location commonly used to store application settings. Encase can mount Windows registry files to allow investigators to inspect the values stored by applications in the Windows registry [48]. This functionality therefore allows investigators to identify various application settings or the presence of malicious applications on a system in an intuitive manner, without having to perform a time-consuming binary search for a text string that may exist in one of the database storage files associated with the Windows registry.

A problem commonly associated with multinational investigations is the presence of time zones. Performing an investigation that spans different time zone boundaries may be challenging as the investigators will typically be required to perform time zone conversions in order to construct a timeline. Encase version 4 has support for multiple time zones, which means that timestamps from multiple time zones can be converted automatically to allow investigators to view timestamps relative to one consistent time zone [48].

Encase also keeps a log of the evidence acquisition and inspection process, which allows investigators to refer to a record of the process followed to extract evidence [54]. This feature allows investigators to perform investigations without actually having to perform detailed logging of every single action that they take, as this is taken care of by Encase.

Another feature of Encase is that it supports the emulation of storage devices via the physical device emulator (PDE) [48]. The PDE allows the investigator to mount an image file as a read-only, emulated hard drive. Investigators using this feature are therefore able to access the files stored on a mounted image, as if the files were stored on a locally available hard drive. The PDE allows investigators not only to mount image files that were stored locally, but also to mount image files stored on a remote server.

Various investigators are therefore able to mount the same evidence file on a remote server, which implies that this feature not only promotes flexibility, but also enhances an investigation team's ability to work concurrently in distributed environments. The capability provided by the PDE can be combined with the powerful features supplied by VMWare, allowing investigators to boot from a captured hard drive image. By using this feature, investigators can gain access to the operating system environment used on the source of acquisition.

Encase also has a module called the Virtual File System (VFS) [48]. which allows investigators to access an evidence source as an off-line, read-only

network drive. The powerful features provided by the PDE and the VFS allow an investigator the freedom to use external third party investigation tools to extract information from the mounted evidence source as if the tool was applied to the original evidence source. The advantage of this capability should be obvious - not only is evidence accessible in a generic manner, but additional functionality can be provided by third party tools in areas where Encase may lack functionality.

4.3.2 FTK

The Forensic Toolkit (FTK) distributed by AccessData is recognized by law enforcement and security professionals as a great tool to perform forensic investigations [1]. FTK allows an investigator to perform thorough examinations of digital evidence by performing tasks such as the extraction of files, the creation of forensic images, and the recovery of deleted files [49].

Digital investigations are usually conducted by investigators following a defined methodology. Although a single investigation process does not exist at this moment in time, some similarities exist among all such processes. It would therefore be possible to describe a generic process by studying available methodologies. The generic process should look something like the following (see chapter 2.2 for a discussion on this topic):

- Acquire evidence;
- Analyze evidence;
- Present the evidence.

FTK and FTK imager support this generic framework by supplying the user with enough tools to perform the required steps [2]. It can therefore be argued that it would, in theory, be possible for an investigator to potentially use any investigation methodology with FTK, as long as it contains elements of this generic process.

Supported file systems

FTK Imager, a tool shipped with FTK, allows the investigator to create images of physical devices [2]. The creation of images is important as it allows the investigator to analyze the content of a suspect's storage device without actually modifying its content. FTK Imager simply makes a mirror copy of the content found on a drive; it does not have the capability to translate the image into a standard, generic format supported by forensic tools. This implies that FTK has to support various file systems to allow the investigator to inspect the content of captured images. FTK currently supports the following file systems [2]:

- FAT 12/16/32;
- EXT2/3;
- NTFS;
- ISO.

FTK also supports images created by external tools, such as Encase, Snapback and dd [2]. This allows investigators to use the imaging tool of their choice to acquire evidence while using FTK to perform the actual investigation.

Compression / decryption support

FTK supports the decryption of Microsoft's Encrypted File System (EFS) [1]. This allows investigators to inspect the content of files that may not otherwise be accessible, due to the protection supplied by the applied cryptography technique employed by EFS. FTK therefore allows investigators to uncover evidence that may not necessarily be accessible by tools that do not support the decryption of EFS.

FTK can perform entropy tests on files to determine the presence of files that are compressed or encrypted. The entropy test is useful from an investigator's point of view, as it allows him/her to determine which files could be compressed or encrypted. From a technical point of view the test is also of relatively high value, as it allows the FTK search indexer to skip indexing any files that are compressed or encrypted, resulting in the conservation of investigation time and resources.

Search support

FTK Supports live and indexed searches [2, 54]. Live searching allows the user to search for a specified pattern or text on an acquired image file. Live searching may be used by investigators if indexed searching fails, or in situations in which investigators try to prove or disprove the presence of specific evidence on an evidence source.

Live searching is an inherently slow process, which means that indexed searching is the preferred method of evidence discovery. Tests performed by Richard [85] show that it takes about 2 hours to perform a live search on a 6 GB hard disk image and about 4 days to complete a live search on an 80 GB image. Very little is known about the experiment quoted, except for the figures that indicate the poor performance of the search functionality. The experiment may have been performed on a relatively old computer system using slow hardware when compared to currently available computer hardware. Regardless of this fact, it should be noted that live searching is a tediously slow process which should only be used when no other feasible option exists.

The indexed search method performs an exhaustive search on an acquired image to create a database of keywords called a “dictionary”. The initial

dictionary creation process is extremely slow as it can essentially be described as a live search for specific patterns. Once the dictionary has been created, the investigator is allowed to perform a search of the content contained within the dictionary.

This method is preferred to live searching as it is much faster [1]. Due to its efficiency, it also allows investigators to experiment with search queries to find evidence that they may not have been looking for initially. An interesting application of the indexed search is that the created dictionary can actually be used as a database for password-cracking applications [1].

Regular expressions are mathematical statements that can be used to describe data patterns. The use of regular expressions can help investigators to mathematically describe the characteristics of the evidence that needs to be found. Regular expressions are supported by FTK's search functionality.

FTK is shipped with several predefined regular expressions that define the form of items commonly stored on digital devices, such as telephone numbers, credit card numbers and IP addresses [2]. The predefined regular expressions therefore allow investigators to easily find information that may be considered to be of value to an investigation.

Extendibility

Extendibility is not one of FTK's major concerns. Unlike Guidance Software (creators of Encase), who target both skilled and unskilled IT professionals, AccessData strives to make it as easy as possible for IT professionals with relatively few technical skills to perform investigations. As a result more complex features, such as scripting support, are not supported directly. FTK was designed to be a boxed solution with a large, but fixed, feature set that allows investigators to perform activities common to the digital investigation process. FTK can therefore not be extended by external parties.

Hashing

FTK implements a feature called the Known File Filter (KFF). The purpose of the KFF is to compare hashes of well-known files against those of files found on a disk image in an attempt to detect files that are irrelevant to an investigation [1, 2]. The KFF functionality can be expanded by importing various external databases, such as the HashKeeper database.

KFF allows the investigator to create alerts that are associated with various hash sums [1]. This powerful feature allows investigators to detect the presence of mischievous files or tools that are commonly used to perform harmful actions.

Other features

FTK supports bookmaking which allows the investigator to mark files that should be referenced in a case report [1]. This feature allows investigators to include files that may be of value in the report that is generated when the

investigation is finished. FTK also has a report wizard that allows the investigator to easily generate custom reports [1] with relatively little effort.

Logging of actions taken by an investigator is important in a forensic investigation as this type of work may be questioned in a court of law. Thus an investigator should constantly log the actions taken while performing the investigation in an attempt to document the complete process used to obtain certain results. This may be a time-consuming process, but FTK automatically takes care of the logging of an investigator's actions [2] to allow him/her to focus on the case at hand, rather than on additional constraints enforced by the investigation process.

4.3.3 Sleuth Kit

Sleuth Kit is an open-source forensic toolkit created by Brian Carrier to perform forensic investigations in UNIX environments [36, 58]. The first version of Sleuth Kit was called the @stake Sleuth Kit (TASK). TASK was based on The Coroner's Toolkit (TCT) and was distributed with similar command-line tools [58].

The utilities provided by the Sleuth Kit may be used by investigators directly or indirectly together with other applications which may add additional features to the toolkit [58]. Two examples of such applications are the Autopsy Forensic Browser [12] and PTK [31]. Both these applications improve the user-experience associated with the use of Sleuth Kit by supplying easy-to-use graphical user interfaces.

Supported file systems

The Sleuth Kit supports disk images acquired using the well-known dd command [36]. The Sleuth Kit was designed to allow investigators to inspect file systems captured from Windows, BSD, Mac, Sun and Linux Systems [58].

More specifically, the Sleuth Kit formally supports the following file systems [36, 58]:

- FAT12/16/32;
- NTFS;
- EXT2/3;
- UFS.

By inspecting the list of supported file systems, it is clear that the Sleuth Kit supports file systems used by commonly available operating systems. Due to the fact that Sleuth Kit is an open-source product, it could be argued that support for virtually any file system can be added, if required. The list of file systems supported by Sleuth Kit is therefore not cast in stone; file system support may be added by users of the toolkit as required.

Open-source software may allow users to identify current software and hardware trends, due to the fact that open-source developers tend to develop

functionality which they require to perform a task. Due to the fact that the development of new features is a time consuming process, and that open-source developers usually do not get compensated for their effort, it can be assumed that new code will only be developed to address a real need that may exist in the software world.

It is possible to argue that the most popular file systems at the moment are the FAT, NTFS, EXT and UFS variants, on the basis that they are supported by this well-known open-source tool. When support for a new file system is added to such an open-source project, it may serve as an indication that the file system has achieved a relatively high level of user acceptance. Chapter 3 presents a discussion of open-source development.

Compression / decryption support

Sleuth Kit does not support encrypted file systems directly; all disk images should be decrypted by an external application before they are processed by Sleuth Kit.

Search support

Sleuth Kit is a set of command-line applications. Command-line applications promote the use of scripting to "glue" several applications together. This powerful feature allows advanced users to perform complex processing using a collection of tools. Unfortunately command-line applications are not as easy to use as graphical user interface (GUI) applications. This may cause problems for less experienced users in using command-line applications. Fortunately a tool called Autopsy exists that provides a GUI for the Sleuth Kit.

Autopsy is a Perl-based graphical interface for the Sleuth Kit that allows users to inspect information extracted by the toolkit [58]. Autopsy supplies the user with a browser-based front end which allows investigators to access Autopsy from almost any operating system platform that supports a browser. The web-based interface also makes it possible for investigators to access their cases from anywhere in the world (assuming that the web server that hosts Autopsy has been configured to allow access to the outside world). Autopsy allows the user to search the information generated by the Sleuth Kit in a user-friendly manner.

Extendibility

Sleuth Kit and Autopsy are both open-source applications. This means that anyone is free to extend the tools as they please. This characteristic makes it possible for users of the application to add any feature that may be required. Chapter 3 provides an in-depth discussion of open-source applications and its their advantages.

Hashing

Sleuth Kit supports the use of hash databases for file verification and

identification purposes [58]. Hashes can easily be added to the hash database by adding the results produced by the well-known md5sum tool to the hash database file. Sleuth Kit also supports the use of the NIST NSRL [76], Hashkeeper [32] and custom hash databases.

Other features

The Sleuth Kit source code is capable of compiling on both Windows and Linux operating systems. This means that a developer should not experience any trouble using the tool on either of the platforms. No license fees are required to use Sleuth Kit which means that the use of the application is not limited by expensive licensing fees.

4.3.4 Tool comparison matrix

Before any new product is developed, it is critical for the developers of the proposed new product to study existing solutions in order to determine common functionality required by the existing market. Table 1 provides a comparison of features supplied by the various forensic tools discussed in this chapter.

Table 1: Common features supplied by Encase, FTK and Sleuth Kit

	Encase	FTK	Sleuth Kit
Scripting Capability	X		X
Includes Disk Mirroring Tools	X	X	
FAT16/16/32 Support	X	X	X
NTFS Support	X	X	X
EXT2/3 Support	X	X	X
HFS/HFS+ Support	X		
UFS Support	X		X
JFS/JFS2 Support	X		
Palm Support	X		

EFS Support	**X**	**X**	
Compressed File Support	**X**	**X**	
Live Search	**X**	**X**	**X**
Indexed Search		**X**	
Supports Hash Databases	X	X	X
Registry Viewing	**X**		
Windows	**X**	**X**	**X**
Linux			**X**
File Carving Support	**X**	**X**	

By inspecting the matrix, it is clear that support for the three common file systems flavors, namely FAT, NTFS and EXT is required. Some common access functionality is also required to access the information stored on the images, namely some form of search functionality. The toolkit should also be usable in Windows to allow the majority of computer users to take advantage of its supplied features. These identified features may be considered to be the minimum requirements that a forensic toolkit must have in order to be of use to an investigator; any tool that contains less functionality may be of little or no use in a forensic context.

4.4 Conclusion

This chapter has described some characteristics of digital evidence as well as the tools used to perform digital investigations. The features of three well-known forensics tools, namely Encase, FTK and Sleuth Kit were discussed in an attempt to uncover common functionality that exists among them. A feature comparison matrix was constructed in an attempt to identify common features that are considered to be of value in the current marketplace, based on the features identified in the tool discussion section.

The minimum requirements of a forensic toolkit were identified by inspecting the feature matrix produced by studying the common features of the three well-known tools. When developing a new toolkit, it is of the utmost importance that the minimum requirements identified here should be adhered to, in order to ensure that the toolkit is of use to investigators.

5 An Open-source Forensics Platform

5.1 Introduction

Digital forensics plays a crucial part in the investigation of crimes involving electronic equipment. Digital evidence collected at a crime scene needs to be analyzed, and connections between the recovered information, physical entities and physical events need to be made and proven. Investigators of digital crimes usually have a lot of complex questions to answer in a short amount of time [18]. The amount of time taken up by the investigations may be attributed to the complexity and extent of digital evidence collected.

As computing technology improves and the storage capacities of digital devices increases, it becomes less feasible to manually inspect devices with sizes ranging from gigabytes to terabytes [3] in capacity. Examiners therefore need to constantly improve their collection and examination methodologies and tools in an effort to improve their efficiency [53].

This chapter proposes an open-source digital forensics platform that may be used by academics to develop digital forensics prototypes, and by industry to perform digital investigations. The remainder of this chapter is structured as follows: Sections 5.2 and 5.3 discuss the need for a forensic platform, while section 5.4 expands on the functionality that needs to be available in such a platform. Section 5.5 discusses a proposed architecture to address forensic needs specified in the literature. Section 5.6 discusses future work that needs to be conducted and section 5.7 concludes this chapter with a brief summary.

5.2 The need for a forensics platform

Different digital forensics tools are available to help forensic examiners to perform forensic investigations. Although these tools may have been tested and proven to perform a specific task well in a specific environment [107], it cannot be assumed that they perform equally well when used in conjunction with other digital tools. This is definitely a problem, since digital investigators tend to make use of a collection of tools to perform their investigations [22]. The existing tools were not necessarily designed to function together as a single cohesive unit to perform the acquisition and analysis of digital evidence. As a result, there may be irregularities and inconsistencies in gathered evidence, which may ultimately lead to the exclusion of some digital evidence from a case due to a lack of trustworthiness. Walker [99] claims that it is critical that all collected digital evidence should be in an uncompromised state; a single file with a timestamp later than the time of evidence acquisition may lead to a situation where the evidence is excluded from a case due to inconsistencies. It is therefore crucial that the tools chosen for the acquisition and analysis of data are able to work together without compromising the digital evidence in any way.

Some investigators may even attempt to create their own acquisition and analysis tools [99]. These noble attempts at creating tools that increase

investigation efficiency are usually rewarded by scepticism in court due to the fact that it is extremely difficult to prove that a custom-made digital forensics tool is forensically sound.

Rogers and Seigfried [86] report that the U.S. Supreme court supplied certain criteria in the Daubert vs. Merrel case that may be used as guidelines by courts to determine whether or not evidence is admissible in court. Custom applications therefore have to adhere to the requirements stipulated by the Daubert Standard to allow the evidence they collect to be admissible in court. Chapter 2 presents an in-depth discussion of the Daubert Standard and the effects of the standard on investigation tools.

Relatively few investigators have the time and skill to evaluate and analyze their chosen tools to determine whether or not it conforms to the criterion stated by the Daubert Standard [22]. Even if the tools do conform to the criteria and perform perfectly in a trusted environment, they may give inconsistent results in an untrustworthy environment [22]. This is due largely to the fact that software applications rarely contain all the operating logic needed to perform basic functionality that can be supplied by external drivers or the operating system; the applications rely rather on libraries or low-level drivers to perform trivial tasks. Unfortunately these libraries and drivers may be compromised to produce results that are inconsistent with the digital evidence.

5.3 Commercial forensic tools

Commercial easy-to-use forensic toolkits tend to be extremely expensive while their open-source (or free) counterparts tend to have limited functionality and are very difficult to use [63]. This scenario creates a problem, since not every investigation team has the funds to invest in an extremely expensive software package and may have to turn to the available open-source alternative. The latter generally has less functionality and requires investigators with more technical skills.

Another problem with forensic tools at the moment is extensibility. Researchers continually invest their research efforts into finding ways to improve the forensic analysis process. Once a new theory has been developed, it needs to be proven. Although there are various ways to prove a theory, it makes sense to create a prototype to demonstrate the theory in action. At present, it is an extremely complicated and technically challenging process to actually create a prototype that conforms to a list of criteria.

This is especially true if the theory is not based on the bit or byte-level, but on a higher, more abstract view of a computer system. The experienced complexity is due largely to the fact that lower-level functionality also needs to be implemented to serve as a basis for higher-level theoretical processes developed through research, that need to be proven.

It would be ideal to build on existent functionality supplied by commercial forensic tools, as their base functionality has already been tested and proven. Unfortunately this is virtually impossible due to the fact that the source code of

these tools is not available to the general public. Only a handful of researchers possess the technical abilities to actually create a fully-functional forensics tool.

Some may attempt to modify available open-source tools to conform to their requirements; others may try to write a tool using a simulated environment to try to prove their theory. Although these steps are in the right direction, a solution is needed to allow researchers to perform rapid prototyping on a tried-and-tested forensics platform in an attempt to speed up digital forensics research efforts. Such progress will ultimately lead to more digital forensic science breakthroughs in shorter amounts of time.

5.4 Functionality needed in a forensics platform

According to Eckstein [33], what the term “digital forensic analysis” entails depends largely on the source of evidence at hand. As an example, consider the two different digital crime scenarios: the first is a denial-of-service attack executed by an individual located at a remote location; the second is the unauthorized modification of a resource located on a local computer not connected to a network.

Analysis of the first scenario will largely consist of scanning through network logs and captured network traffic, while analysis of the second scenario will rely on the analysis of a captured hard drive image. Although the forensic analyses in the two different cases focus on different forensic media types, they are both equally valid evidence sources which may be used to implicate the parties involved. It is therefore crucial that a forensics platform should support the analysis of sequential data (such as a captured network trace) as well as relational data (such as captured disk images).

5.4.1 Digital evidence characteristics

The characteristics of digital evidence should be taken into account when the possible functionality of a digital platform is defined, in an attempt to capture the needed functionality more accurately. Chapter 4 provides a complete discussion on the characteristics of digital evidence. As mentioned in chapter 4, digital evidence has the following characteristics:

- Digital evidence can be copied easily; unfortunately the copying of digital evidence does not guarantee a consistent copy of the original evidence.
- Digital evidence is difficult to authenticate.
- Digital information is not well perceived by human senses. Humans therefore experience difficulty understanding captured digital evidence.

A forensics platform should address the stated characteristics by supplying functionality that attempts to solve the problems experienced with digital evidence. A simple solution to the first two problems may be to supply the

forensics platform with secure hashing algorithms that can be used to prove the authenticity of captured evidence.

The last problem holds a bigger challenge: how to structure captured data in such a way that relevant information is more visible to an investigator than data that may be unhelpful in solving a case. This characteristic is extremely important, as it is crucial for investigators to create an abstract view of the digital evidence in question [92]; without such an abstract view, much more time would be needed to analyze and interpret the data, which ultimately increases the cost of the investigation process.

5.4.2 Evidence timelines

Another aspect that should be taken into account when examining evidence is the relationship that exists between the collected evidence and the time of collection. A clear distinction should be made between the various stages or timeframes that surround a digital investigation, in order to create a clearer forensic vision of key aspects of the digital crime under investigation. These aspects may include: the possible suspects, the digital events as well as the connections between the suspects and the digital events that led to the perpetration of the crime.

Evidence collected at various stages will be related differently to these aspects. As an example, consider evidence taken before the actual perpetration of a crime and evidence taken after the event in question occurred. Surely the type of information that will be extracted from the two different evidence sources will be different, each with different forensic intentions.

Evidence taken before the event took place will describe the functioning (or malfunctioning) system, its users and its environment. According to Pfleeger and Pfleeger [82], an attacker must have three things to be able to perform a malicious attack, namely: method, opportunity and motive. Although the type of evidence taken before a crime has been committed may show indications that a computer crime may be committed in the very near future, it shows no concrete evidence of a crime at the moment of capture. However, it may be used to indicate motive, opportunity or method, all of which are needed to implicate possible suspects once a crime has actually been committed.

Evidence taken after the event in question will show the state of the system after the event took place. By examining the captured system state, investigators should be able to deduce which individuals were responsible for committing the act in question. The example illustrates that it is extremely important to make a distinction between evidence captured at different stages in an investigation. Three different stages have been identified to illustrate the distinction between the information conveyed by the captured data at different stages of the investigation. These stages are:

- Pre-incident stage;
- Incident stage;

• Post-incident stage.

Figure 1: Various incident stages

Figure 1 presents a visual representation of the three stages. It should be noted that the first two stages are characterized by the capturing and analysis of live data, while the last phase stage is characterized by the analysis of static data captured at the crime scene.

Forensic readiness describes the extent to which a system is able to supply forensically-sound information to aid any digital investigation process [74]. Special software and hardware can be installed to monitor user actions. Policy management and the enforcement of restrictions can minimize the likelihood that the users of systems can participate in mischievous activities without being noticed. Suspicious activities may be captured and logged as required.

The incident stage is concerned with the capture of digital evidence while a crime is being committed. The incident stage is primarily responsible for the capture and archiving of events as they occur in real time. The primary goal of the incident stage it to implicate involved parties by capturing identity-revealing information as the digital crime is being committed. This stage is likely to be associated with the capture of network traffic at the time the crime is being committed.

According to Corey et al. [28], instead of capturing a subset of live data, it is better to capture all the available data and analyze a subset of the data at a later point in time. This is done to prevent potentially crucial pieces of evidence from being “tossed away” during the capture process which may cause investigators to reach false conclusions during later stages of the investigation.

The last stage is the post-incident stage in which the entire suspect's and/or victim system's state is captured and analyzed after the digital crime has been committed. This stage is characterized by the mass archiving of the states of the systems involved in the digital crime in an attempt to determine how the systems were used and by whom.

A digital platform should be able to make a distinction between these stages to facilitate the possible automatic identification of links between evidence captured at the various stages of an investigation and to help investigators pinpoint crucial evidence located in masses of digital data.

From a more technical point of view, a forensics platform should support

formats commonly associated with digital forensics. These formats may include (but are not limited to) FAT or NTFS for file system formats, TCPDump format (the de-facto standard for captured network traffic [28]) and other formats that are considered to be in common use. By supplying a platform that supports these file formats by default, the task of the researcher trying to develop a new and revolutionary forensics prototype would be greatly simplified, as he/she would need to focus only on the research question at hand, and not on the small surrounding details.

5.5 A proposed platform architecture

According to Casey [19], there is a difference between the examination and the analysis phases of forensic evidence. The examination phase is concerned with the extraction of digital evidence from the scene of a crime, while the analysis phase focuses on finding relationships between the evidence, the events that took place as part of the crime, and the parties involved.

From a technical point of view, the platform, named the Reco Platform, should physically support the examination phase, and supply the low-level functionality needed to perform the analysis phase with ease. It would therefore be a good idea to develop a platform that supports the acquisition, examination, storage and analysis of digital evidence.

In this study, a layered architecture was chosen to allow researchers to easily build their research prototypes on top of existent lower-level processing layers. Not only will this layered approach save researchers valuable time when implementing a prototype, but it will also help to decrease the amount of errors introduced as a consequence of programming mistakes. This is due to the fact that lower-level layers are likely to have been tried and tested by many, whereas freshly written code may not have been analyzed for errors as vigorously as its layered counterparts.

Four different layers have been identified, namely:

- Physical;
- Interpretation;
- Abstraction;
- Access.

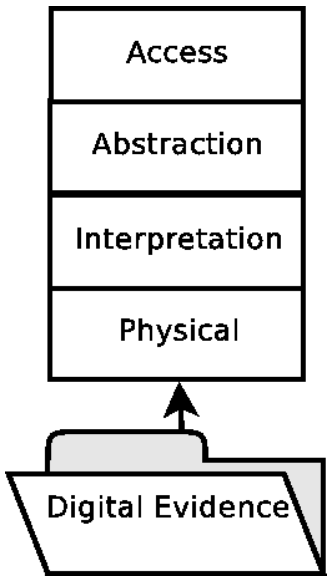


Figure 2: The layered architecture used by the planned platform

Figure 2 visually depicts the layers and the relationship between them. The functionality supplied by each of the respective layers is discussed in detail in the following sections.

The physical layer

Digital evidence in a popular forensic format (such as a disk image or TCPDump trace) is supplied to the physical layer as input. Because digital evidence is likely to be supplied in byte-by-byte copies of physical devices (RAM, ROM, hard drive), or the state of a physical communication devices such as an Ethernet card at a particular point in time, it would be useful to develop a software emulation layer that emulates the original physical device from which the evidence was captured.

The advantage of this approach is that it may be possible to adapt already existing device driver implementations and to use the supplied software emulation layer, which may once again save implementation time. This is due to the fact that a custom driver would not have to be developed from scratch as existing tried-and-tested driver code may be modified and used.

The interpretation layer

The interpretation layer will typically consist of software that performs the same task as traditional device drivers on a conventional operating system. The purpose of the interpretation layer is to read the block or stream-level data supplied by the physical layer and to convert it into file or entry-level information which is commonly accepted by and understood by programs as well as individuals.

The abstraction layer

The purpose of the abstraction layer is to supply functionality that is not specific to any operating system or computing platform in an attempt to hide unnecessary details that may obscure an investigator's perception of the information depicted by the digital evidence. Another purpose of the abstraction layer is to assist investigators in identifying relationships that may exist among different pieces of digital evidence. As Tallard and Levitt [92] state, this functionality may be crucial to help filtering out data that is not relevant to the case. It also to help to create abstract objects that can be interpreted in a relational manner to other objects, thus saving valuable investigation time.

The access layer

The purpose of the access layer is to supply investigators with access to the information interpreted by lower-level system layers. Searching and indexing, as well as access control functionality are expected to be implemented on this layer. Visual abstraction may also be present on this layer in an attempt to display captured digital information in a way which is perceived better by humans. As mentioned before, digital evidence is not well perceived by the human senses; therefore this functionality is needed to help investigators understand the collected evidence in a shorter timeframe.

5.6 Future extensions

The architecture of the planned forensic platform has been discussed in some detail. More prototyping as well as the actual design and development of the various layers will take place in the following chapters of this dissertation. A definite need exists for a logging mechanism that will allow external evaluators to actually determine if a closed-source software component is functioning correctly or compromising the integrity of captured data, by simply reviewing the log file generated by components above or below it in the layering model.

5.7 Conclusion

This chapter has emphasized the great need currently experienced by the academic as well as the forensic investigation community for an open-source forensic investigations platform on which forensic research prototypes can be built and tested. With the help of such a platform, the development time of forensic research prototypes could dramatically decrease, leading to an increase in digital forensic investigation breakthroughs in a shorter period of time.

The architecture has been defined based on industry needs identified in the literature. A layered architecture has been proposed that will be used to implement the platform prototype. The layered approach will benefit researchers by supplying them with the necessary common forensic functionality so that they can take the focus off the technical aspects regarding forensics and focus on what is really important - their research questions. Overall the forensic platform addresses a need experienced in the community and development should therefore continue to ensure a better and safer digital environment for all.

The next chapter will expand on the work discussed in this chapter in an attempt to define a design that conforms to the architecture discussed in this chapter, while conforming to forensic requirements as discussed in previous chapters of this dissertation.

6 Prototype Design

6.1 Introduction

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity - Brooks [6] .

In his book called *The Mythical Man Month*, Brooks [6] makes the case that there has never been, nor will there ever exist a tool or technique capable of dramatically improving the efficiency of the development process. This statement has held true for about two decades, and is still true today.

The purpose of the platform proposed in this study is to allow researchers to improve the rate at which prototypes are developed. Although the use of such a platform cannot guarantee order-of-magnitude increases in productivity, it may allow users to change their development focus from purely technical to a more process-oriented style of development. This shift in development focus may help to encourage prototype developers to focus on the development of new research ideas instead of the development of sophisticated code, and thus ultimately increase the rate at which research prototypes are developed.

This chapter focuses on the design of a forensics platform according to the platform architecture described in chapter 5. The design of the platform components are discussed in an attempt to allow the reader to gain a better understanding of the internal working of the platform.

This chapter is structured as follows. Section 6.2 discusses the purpose of the platform and various goals that should be achieved through the development of the prototype platform. Section 6.3 presents an in-depth discussion on the internal design of the various components found on each of the platform's defined layers. Section 6.4 discusses work that will have to be performed in the near future to ensure that the platform's design is optimal. Section 6.5 offers a conclusion to this chapter.

6.2 Prototype purpose

In his book called *The 80/20 Principle*, Koch [55] argues that 80 percent of all results are obtained by 20 percent of all effort. This is due to the fact that people tend to focus their efforts on aspects that add little value to the overall result, when in fact they should devote their efforts to the aspects that make the biggest contribution to the overall result. Applied to software engineering, the principle states that 80 percent of any application's user base makes use of only 20 percent of its features, while only 20 percent of users make use of 80 percent of an application's features [46].

Simple design tends to be easier to understand than complex design and is less costly to implement [35]. This is due largely to the fact that complexity

tends to obscure the focus point of developers. The prototype in this study was designed to be as simple as possible, while proving that the architecture described in chapter 5 can be used as a platform on which developers can build forensic prototypes. The purpose of the prototype was not to serve as a competitor for well known tools such as Encase or FTK, but rather to be a proof-of-concept for ideas described in earlier chapters of this dissertation. More specifically, the purpose of the platform prototype is as follows:

- The prototype should prove that the described architecture makes it possible for researchers to develop forensic prototypes using the developed platform technology.
- The prototype should prove that the platform's design allows developers to develop prototypes without having to extensively focus on unnecessary details, such as file system implementation details.
- The prototype should prove that the designed platform can easily be extended.

The prototype was designed to be extendible through plug-ins which allow developers to potentially add any required feature without too much development effort. The platform is not intended to be specific for any particular methodology or procedure; it is left up to the platform developers to extend the platform to support specific methodologies and procedures as required.

From a platform point of view, this design is appropriate as it may not always be possible to develop a set of features that will be accepted by all users of the platform. Effort should rather be applied to the development of features that allow other researchers to develop their own features, as required. If the developed platform actually allows researchers to develop their own custom functionality, then it has served its purpose.

6.3 Design

Chapter 5 describes the architecture that was used to develop the platform. Development of a prototype that conforms to the stated architecture is a complex, time-consuming process in which hundreds of new source files containing thousands of lines of code are developed. The sheer magnitude of such a project would be enough to discourage any newcomers from improving the prototype. The aim of this section is to supply the reader with a high-level overview of the design of the implemented system, in an attempt to increase the understandability of the work. Details are intentionally omitted from class diagrams in order to minimize the complexity associated with the discussion of the design.

As discussed previously, the platform makes use of a layered architecture. A characteristic of a layered architecture is that the level of abstraction

experienced by the user increases as more as more layers are added. A layered architecture is considered to be a very attractive choice, as it allows developers to choose the level of abstraction that they require when developing their prototypes using the platform. Higher-level layers supply the developer with a higher degree of abstraction, but less configuration capabilities, while lower-level layers supply the user with less abstraction, but a higher degree of control.

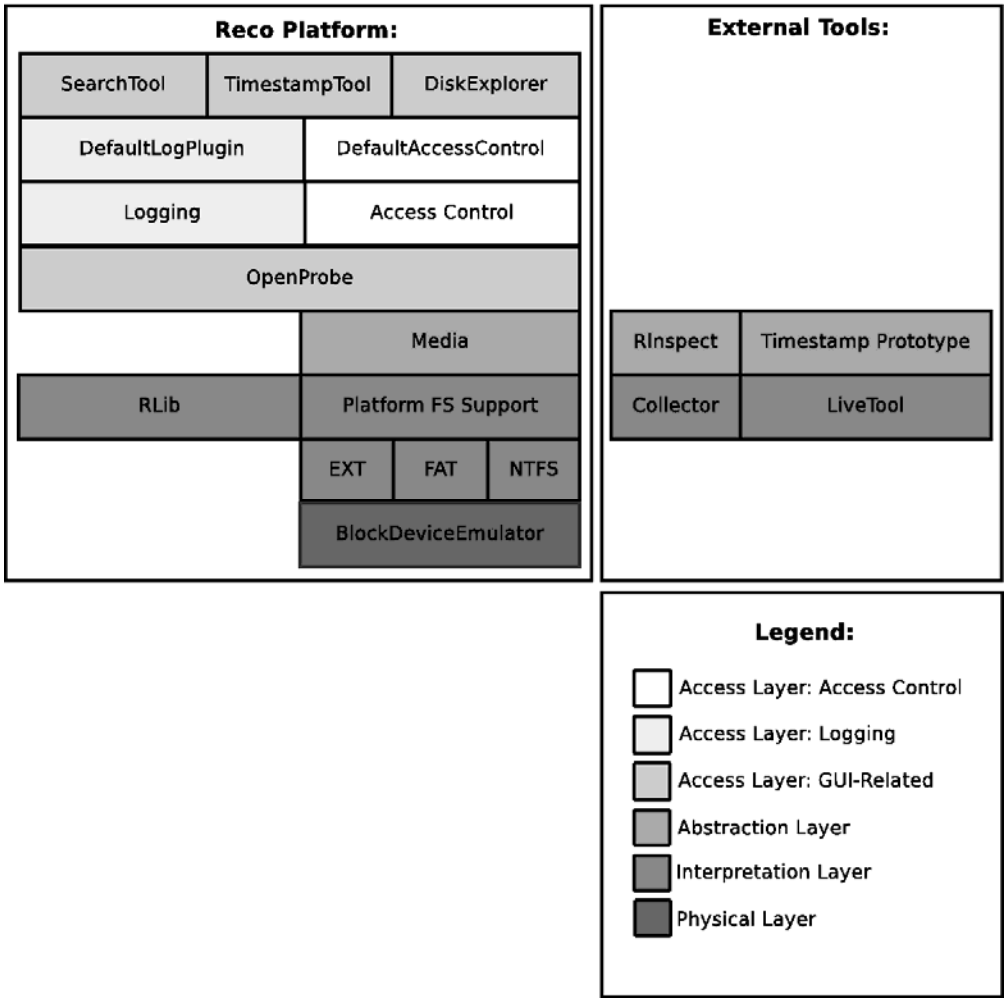


Figure 3: The developed platform components on each defined layer

Various components of the prototype were developed, each of which can be placed on one of the defined layers, according to the functionality that it provides to the platform. Figure 3 illustrates components that are both internal and external to the platform. The internal components were developed as part of the platform while the external components were developed in an attempt to illustrate the usefulness of the developed platform components. Each of the platform's layers and supplied components is discussed in the following subsections.

6.3.1 Physical layer

The purpose of the physical layer is to emulate the functionality supplied by physical devices in an attempt to create virtual devices that are accessible in a similar fashion to a real non-emulated device. The reasoning behind this decision was that it is easier to adapt already-existing open-source drivers to make use of emulated devices, rather than to recreate drivers specifically for this platform.

Block devices can be described as devices that move data in the form of blocks [103]. Common examples of block devices include hard drives, CD-ROMs and memory modules. Block devices are emulated by the platform. Functionality is supplied that emulates common functionality associated with block devices, such as reading blocks of data.

Adapting existing open-source drivers to make use of the block device emulator extremely simple. All that the developer needs to do is to substitute the operating-system specific block device access code used, with the code supplied by the block device emulator.

It should be noted that the block device emulator does not support writing of blocks of data as this may cause the evidence at hand to be compromised. The current version of the platform only supports file systems, but future versions may include code in an attempt to access memory images and network traffic dumps.

6.3.2 Interpretation layer

The purpose of the interpretation layer is to receive the data supplied by the physical layer and convert it into information that is more understandable to human investigators. The purpose of the interpretation layer can therefore be summarized as the layer which contains the mechanisms that have the capability to interpret raw data in a similar way that a device driver interprets data received from a device. Due to the close resemblance between the purpose of an operating system device driver and code on the interpretation layer, it is actually possible to adapt existing operating system drivers for use with the platform.

There are various advantages to this approach. Not only does the modification of existing driver code save valuable development time, but it may also contribute to a higher quality solution, due to the fact that the driver may already be in use in other projects and is therefore likely to be relatively stable. The development of a device driver is an extremely complex process with potentially thousands of pitfalls that may cause developers to produce code of relatively low quality. Rather than waste resources on the development of a custom driver, it was decided that existing drivers should be modified to accommodate the needs of the platform.

The platform should allow extendibility; rather than forcing developers to develop extensions to the platform in a specific way, it was decided not to enforce any restrictions on the development style or methods used to interpret data supplied by the block device emulator. This means that the developer is free to use existing, modified, open-source driver implementations or create

custom driver implementations if required.

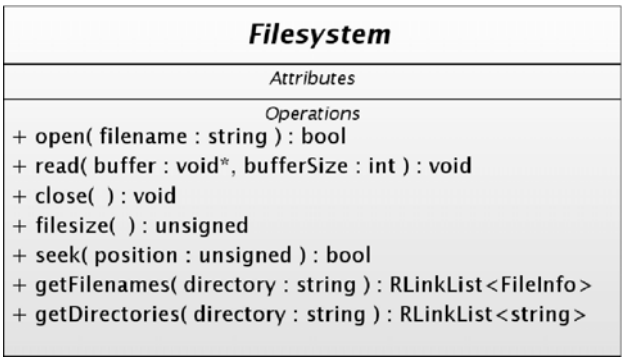


Figure 4: Functionality expected from all file system drivers

As discussed in chapter 2, evidence should only be accessed in read-only mode to ensure that the evidence is not accidentally changed upon inspection. Drivers supplying file system access to the platform should therefore only be able to grant read-only access to captured disk images. None of the drivers are required to implement any special non-driver related features, such as logging and access control, but developers are free to include this, if required. The only requirement is that all platform drivers should supply a set of common functions that are normally associated with all platform drivers.

Figure 4 illustrates the interface that should be supplied by all platform file system drivers. It should be clear that the file system interface was designed to mimic the way files are accessed on an object-oriented experiment. Common file operations, such as opening a file, reading from a file and viewing the size of a file are supported by the defined interface. More specifically, the interface supplies the following functionality:

- Opening and closing of files located on an emulated block device;
- Reading blocks of data from opened files;
- Getting the size of the current open file;
- Moving the file read pointer to a desired location;
- Getting a list of files and directories located within a specific directory.

The interface that is required from file system drivers will have to be developed by the person integrating the driver with the platform. The interface may be seen as a Facade [44] design pattern, with its primary purpose being to supply an easy way to access the large body of driver code hidden beneath.

The feature comparison matrix in chapter 4 identified that all of the inspected forensic toolkits already have support for FAT, EXT and NTFS file systems. The standard platform is distributed with support for the FAT, EXT and

NTFS range of file systems to be able to conform to the identified baseline features present in all three toolkits.

The FAT and EXT file system drivers were taken from well-known open-source projects, while the NTFS support was developed from unofficial NTFS specifications, created primarily by Russon and Fledel [88]. The following subsections provide more information on the file system drivers.

EXT file system support

Support for the EXT2 and EXT3 file systems are supplied by a modified version of the libext2fs [97] library. The libext2fs library was developed by Theodore Ts'o [96] to be used by a set of EXT2/EXT3 file system utilities called e2fsprogs. Ts'o has been involved with Linux kernel development since 1991, primarily in the field of file systems [106].

Ts'o is also a contributor of a large part of the Linux ext2fs kernel code [11]. Based on Ts'o's reputation, it can be assumed that the library has been developed by an EXT file system expert and is highly likely that the library conform to EXT specifications. The inclusion of libext2fs therefore poses relatively little threat to the stability of the platform.

FAT file system support

FAT file system support was supplied by extracting and modifying the FAT file system driver shipped with the FreeDOS project [41]. The specifications for the FAT12/16/32 file systems have been published. The published white papers allowed the FreeDOS developers to develop a file system driver from the official specifications. The availability of an official specification minimizes the risk of an implementation error as a result of the use of vague or incorrect specifications in the development process [14].

If the specification is implemented properly, the device driver should be able to produce deterministic results as required. This would allow the inclusion of the driver in the platform, without compromising the perceived integrity of the results produced by the platform.

The FreeDOS operating system has been known to have been tested and used by big names in the computer industry such as Dell, HP and ASUS [104]. The mere fact that these vendors are willing to ship some of their products bundled with FreeDOS implies that FreeDOS was tested by the vendors to ensure that FreeDOS is a quality product. It can therefore be assumed that the file system driver is relatively stable and can be used to supply FAT support to the platform.

NTFS file system support

TThe NTFS file system was developed by Microsoft as a replacement for the FAT12/16/32 range of file systems. The NTFS file system is vastly superior to its FAT file system predecessor in various aspects, such as operating speed, stability and ease of recovery. Unfortunately the specifications of the NTFS file system have been kept private by Microsoft, which means that no official NTFS description is available.

Fortunately unofficial documents have been made available for developers, most notably the documentation written by Russon and Fledel [88]. An NTFS driver was developed for the platform according to the available documentation. Chapter 10 provides more information regarding the development of the file system driver.

Interaction with the file system drivers

The use of various file system drivers may be a daunting task as each of the file system drivers is likely to supply different interfaces to access its supplied features. The use of non-standard interfaces for each file system driver may introduce unnecessary complexity to the platform as each of the file system drivers will require different methods of access.

The platform should supply a simple way to access its features to allow the prototype developer to create prototypes without having to consider the underlying platform complexities. It was therefore decided to introduce a standard interface that should be supplied by all file system drivers used by the platform. The interface should help the prototype developer to access disk information in the same manner for all file system drivers, irrespective of the file systems supported by the various drivers.

The platform supplies a standard method to access file system drivers. A single class is responsible for the creation of file system objects that may be used to access the file system supplied by a block device emulator. All that is required by the user to create a file system object is a reference to the block device emulator which contains the disk image of interest. Figure 5 is a visual representation of the design. Although the class diagram depicted in figure 5 is not complete, it supplies sufficient information to understand how file system drivers are created.

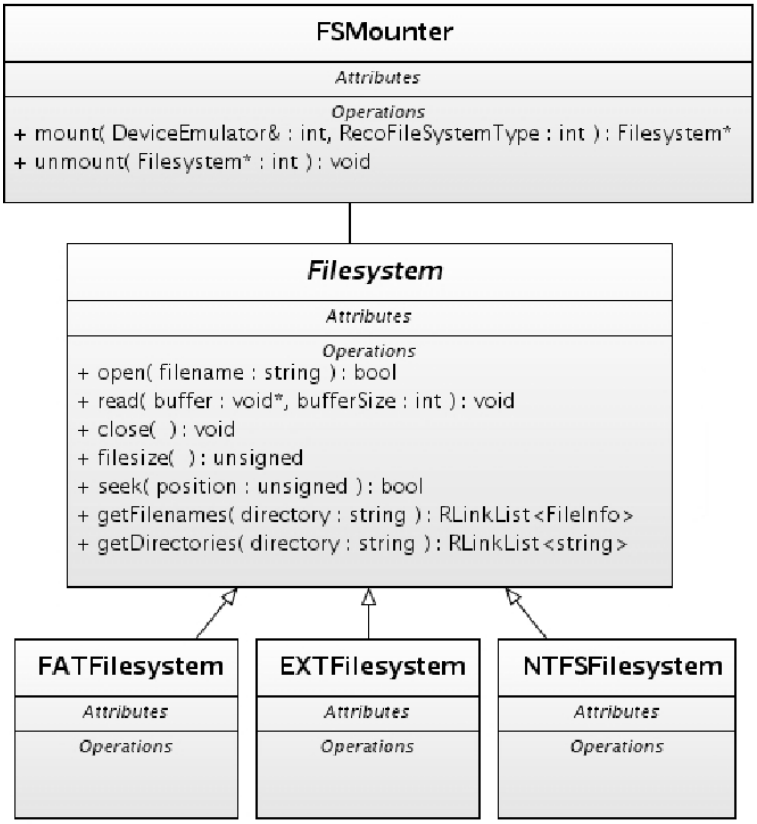


Figure 5: File system objects and the associated factory object

The FSMounter class acts as a factory class that produces instances of file system objects, according to the file system type specified. This design allows the user of the code to easily mount file system images containing various types of file systems, without exposing the user to the intricate details surrounding the implementation of the file system drivers.

Once a file system object has been obtained, the developer can access files located on the file system in question by using the method supplied by the file system class. This design allows developers to access files located on a disk image without having to implement any of the underlying file system access code.

6.3.3 Abstraction layer

Masses of digital information may exist on a system that could potentially take weeks or months to be processed by investigators. To speed up this process information needs to be interpreted to create abstract objects [92]. The purpose of the abstraction layer is to supply platform users with enough abstraction to encourage the development of code that is able to extract information from evidence sources supplied by lower-level layers. Code included at this level will typically extract and process metadata contained within various high-level file types, such as executable files, audio files and video files. The abstraction layer

therefore makes it possible to extract human-understandable information from files located on evidence sources.

As discussed in chapter 5, the creation of abstract views of digital evidence may help investigators to analyse and interpret evidence in shorter periods of time. Various file types contain metadata that offer a high-level description of the file as well as its contents. Music files may contain additional information, such as the song title and artist. Digital photos may contain information about the camera used to create each of the respective photo, the location at which the photos were taken as well as the time at which each of the photos were taken. Extraction of the metadata contained within the media files may help investigators to form abstract views of the evidence at hand.

The current version of the platform is distributed with code that is capable of extracting metadata from media files found on disk images. This allows developers to extract file meta-information about various media types, which may later be used by the search functionality provided by the platform to locate such files. Table 2 supplies a complete list of supported media types.

Table 2: Various media types supported by the platform

Media formats:	ASF, AVI, WMV, WMA, MP3 (ID3v1 and ID3v2), OGG Vorbis
Image formats:	BMP, JPEG
Executable formats:	Windows EXE, Linux ELF
Documents:	HTML, PDF, PS
Archive files:	Zip

6.3.4 Access layer

It is likely that more than various investigators may work on a case during its lifetime. Some of the captured evidence may only be accessible to specific investigators, due to policy or confidentiality reasons. To accommodate this requirement, an access control system was introduced to allow investigators access to information which they are authorized to access.

The purpose of the access layer is to allow the end user to obtain access to processed digital evidence in a useful manner. Access layer code consists of code that processes data and displays the results for inspection purposes. It can

therefore be assumed that components on the access layer will be driven by graphical user interfaces and may require input or user response in order to perform their designated tasks.

Logging and access control are found on this layer; if for some reason the user of the platform requires logging and access control facilities on a lower-level layer, this will have to be developed independently by the user. The access layer also supplies the user with facilities to store case-related information using an SQL database. The end user only obtains access to database objects if he/she is authorized to do so. Each time read or write access is required to an object stored in the database, the access control plug-ins are consulted to determine whether or not a user has sufficient privileges to do so.

Every database read or write access attempt will be logged by the log plug-ins that may be used at a later stage for inspection purposes. The supplied logging facility and access control facilities are discussed in more detail in the following subsections.

The log facility

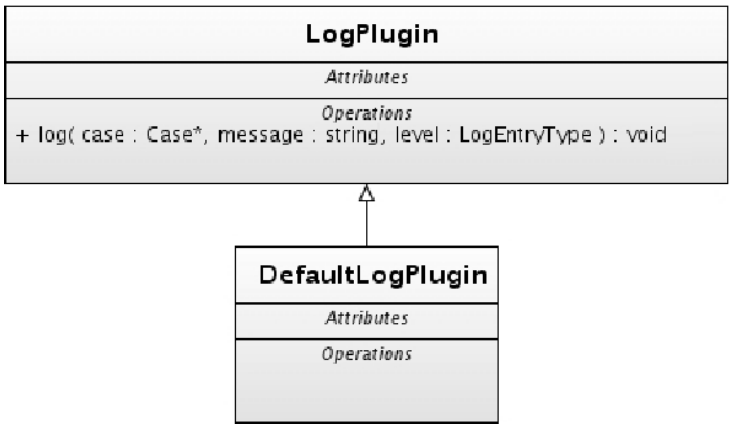


Figure 6: The interface that should be implemented by all log plug-ins

Event logging is supplied by the built-in log facility. The logging facility was designed to be extendible; users of the platform may create their own log functionality by creating log plug-ins. The custom log plug-ins may simply be selected by the user when a new case is created.

All log plug-ins should provide a standard feature set that can be used by the platform as required. Figure 6 is an illustration of the interface expected to be implemented by all log plug-ins. The platform is distributed with a default log plug-in. The default log plug-in which is able to print log messages to either the system console or a log file.

As pointed out by Stallard and Levitt [92], the following problems may be experienced with logging facilities:

- The facilities may be disabled due to configuration error or malicious intent.
- Improper configuration of logging facilities may cause crucial pieces of event information to be ignored.
- Log rotations may delete evidence of attacks.
- Log files may be tampered with.

The supplied log plug-in is capable of logging all case data access and modification requests. The supplied log plug-in can therefore be seen as very primitive, as no additional features are supplied to combat the problems with the perceived usefulness of logging facilities. Fortunately the simplicity of the default log plug-in is not such a big threat as it may seem at first, as developers may simply replace or modify the existing plug-in, or create a new plug-in to supply the desired functionality.

Access control facility

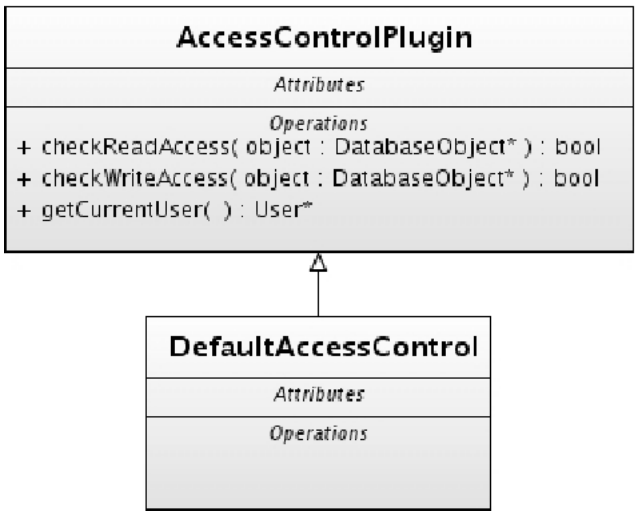


Figure 7: Interface that should be implemented by access control plug-ins

The purpose of the access control facility is to determine whether or not a user is authorized to access a database entry before a database read or write operation is performed. This implies that the access control facility is also responsible for user authentication. The access control facility is similar to the log facility in the sense that it can easily be extended by users of the platform through the creation of plug-ins. Figure 7 is an illustration of the interface that is expected to be implemented by all access control plug-ins.

The platform is distributed with the default access control plug-in which performs password-based authentication. Any user that was able to successfully

log in will be granted full read and write access to the data in the database by the access control plug-in.

As discussed previously, the purpose of the platform is not to supply users with a full set of features that may rival commercial products, but rather to supply users with the tools and mechanisms to develop the functionality that they require themselves. More advanced access control models such as RBAC [4] and Bell-LaPadula [5] may be implemented by the users of the platform, as required.

6.3.5 Database access

Data generated throughout the investigation process should be stored by the platform in an efficient manner to allow investigators to retrieve it at a later stage. A flat file may be the simplest storage solution as it would be relatively easy to implement. Unfortunately the flat file solution would not be suitable for use in a distributed environment as data sharing and consistency may become a problem.

It was decided to use already-existing SQL database engines as the storage medium for case-related data. The use of a SQL database engine should simplify data access and accommodate data access in distributed environments, due to network and locking support supplied by popular database servers.

The access layer allows plug-ins to store generated results in a relational database. At the time of writing, the platform supports the use of three well-known relational database engines, namely SQLite, MySQL and PostgreSQL. The platform supplies the user with a standard interface that is similar for all three database engines, which allows the user to perform database operations.

This design shields the platform user from technical database complexities, while facilitating the use of potentially any ANSI-SQL compliant database engine, as long as a database wrapper is created for the engine that conforms to the specified interface. Figure 8 is an illustration of the database abstraction design implemented by the platform.

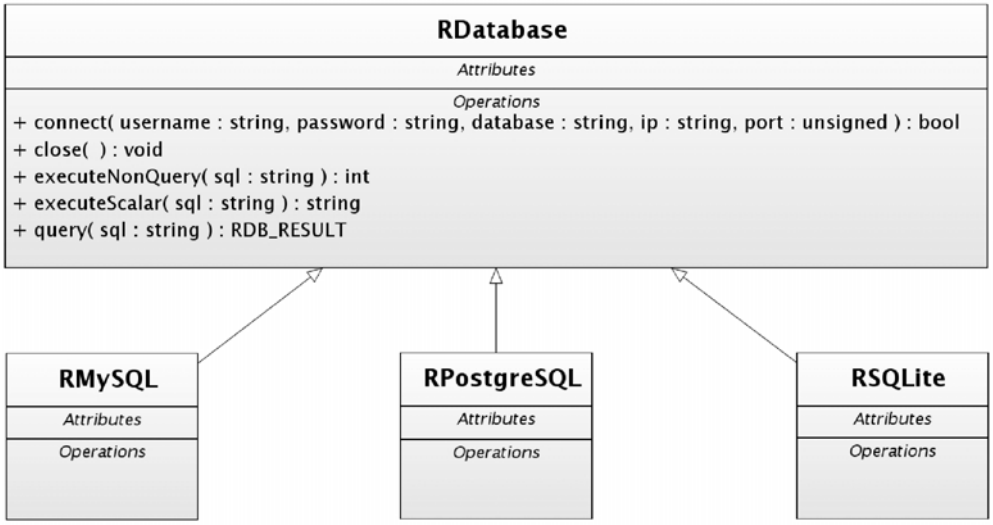


Figure 8: Database abstraction implemented in the platform

SQL queries may be used to gain access to data stored within a database. The creation of SQL queries to perform database operations may be a tedious process to perform. Already existing queries may simply be duplicated in other code locations due to the fact that the same database functionality may be required at various locations within program code.

The modification of a database table may create chaos within code in which queries are distributed among hundreds of source files as programmers may simply forget to update the queries as required. This creates the situation in which some database queries may simply stop working due to the fact that they were not updated when the database structure changed.

It was decided to keep database queries as close together as possible through object orientation. Database queries should be placed within classes specifically designed to perform database access operations. Each table in the database should have a database access class associated with it. All database queries relevant to a table should be placed in its associated database access class.

Each case will have a database access object associated with it. This object should be accessible by all objects associated with a case, in order to allow them to gain read and write access to the database. In an attempt to improve the efficiency and ease-of-use of the platform, it was decided to introduce a base database object class that should be inherited by all classes wishing to save and retrieve data from the database.

This base class performs critical database and platform-related operations which may easily be overlooked by developers when writing code. The operations performed include:

- Table level locking/unlocking before and after database access is performed.

- Tight integration with the security model used. Access control is performed before every read and write operation to ensure that a user can only access information that he/she is allowed to access, according to the implemented access control plug-in.
- Automatic logging of read and write operations. Log entries are generated for every database read or write operation performed.

Figure 9 may be consulted for an illustration of the database object class.

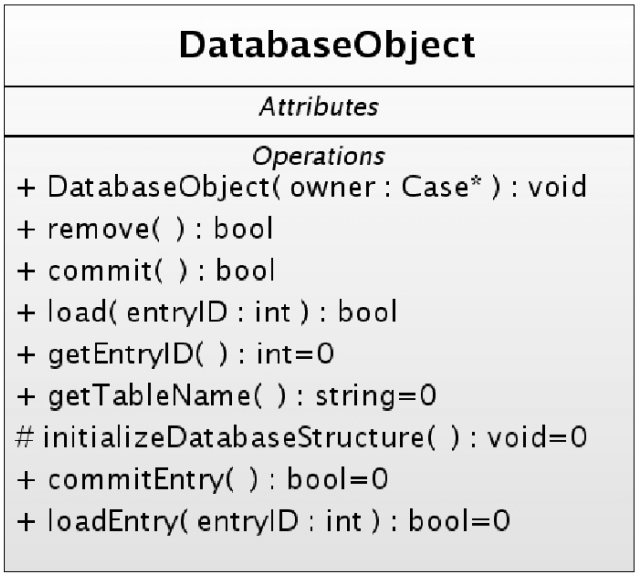


Figure 9: The DatabaseObject class

Each object should represent one single entry in a database table. An instance of the DatabaseObject subclass should allow users to obtain access to database entries without requiring them to write any SQL queries. Every DatabaseObject subclass should implement the interface provided by the DatabaseObject class. More specifically, each DatabaseObject should provide the following functionality:

- The creation of database tables required by the storage requirements of the class;
- The performance of read and write operations on the created database structure.

The design can effectively be described as a database table wrapper object that is capable of performing basic read, write and locking capabilities, without exposing the user to the underlying database implementation details. This allows platform users to focus their development efforts on solving problems

instead of the construction of complex database queries.

6.3.6 Database searching

As identified in chapter 4, the presence of search capabilities is one of the basic functionalities expected from forensic tools. It should be possible to query a case database for specific information. Due to the fact that the data in question may potentially be a hidden table entry among literally thousands of entries, the task at hand may not be as simple as it seems at first.

A simple solution to the problem is to force every database object to implement search functionality. This is a feasible solution, but forces the platform developer to supply functionality which may not necessarily be applicable to every situation.

Consider the situation in which a database table stores generated binary data. The execution of an ASCII search on binary that is not meant to be accessed in such a fashion may produce random results at best. Forcing the developer of the database object to implement search functionality in such a case may do more harm than good, as it can be argued that the developer may simply ignore the query and return no results, or worse: the developer may actually implement the method to return values that are likely to appear random when viewed out of context. Forcing developers to implement search functionality is therefore not an option.

To improve the situation, it was decided to allow developers to create plug-ins that can be used to search database tables. Each database table can therefore by made searchable by adding the relevant plug-in. The plug-in is registered at a central location which allows all plug-ins to make use of this search capability. Each of the plug-ins that possess database searching capabilities needs to implement a specified interface, as illustrated in figure 10.

Figure 10: The Searcher and SearchFactory interfaces

Each class that possesses the capability to search a database table for a specific value should inherit the interface provided by the Searcher class. Each Searcher subclass should also have an associated factory class that is registered at a central location. Every time a plug-in needs to utilize the search capabilities provided by the platform, it simply needs to get a list of Searcher factories that are registered for use with the platform. The plug-in can proceed by constructing

searcher objects as required to perform queries on stored case data.

6.3.7 Storage of settings

Plug-ins may need to store user settings. The settings will typically consist of configuration data that will be used to determine how each of the respective plug-ins will operate.

Each plug-in can use configuration files that contains various configuration settings. Although this solution may solve the problem, it is rather crude. The platform was designed to allow extendibility through plug-ins. This means that literally thousands of plug-ins may be added to the platform. If every single platform plug-in created a configuration file, the result may be an unmaintainable mess.

A better solution was to create a central repository that allows plug-ins to store their associated cross-case related settings. Please refer to figure 11 for an illustration of functionality provided to allow each plug-in to save and retrieve system-wide settings.

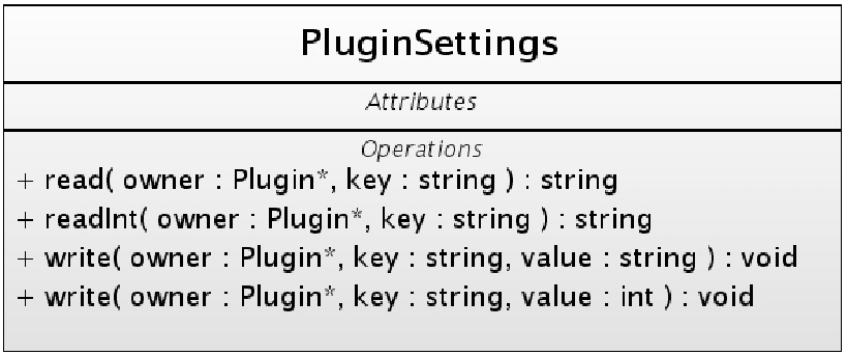


Figure 11: Interface provided to allow plug-ins to save and retrieve settings

The plug-in storage facility is similar to the Windows Registry in the sense that it serves as a common storage facility for application settings. Each piece of data is associated with a plug-in and a key. The plug-in/key combination is used to uniquely identify the information stored by the various platform plug-ins. No more than one data item is allowed to be stored by the plug-in/key configuration. This means that no more than one value may be associated with the plug-in/key combination at any time.

The storage facility stores the settings received from various plug-ins in a compressed XML file. The compression is provided by the Zlib [43] compression library to reduce the size of the stored data.

6.3.8 Access layer plug-ins

The access layer allows the user to develop custom plug-ins that can be

displayed within the existing GUI framework provided by the platform. Evidence artefacts can be accessed as abstract objects that are more understandable to human operators. Lower-level layers allow the user access to more technical details, such as physical blocks of an acquired image, without performing interpretation through the use of file system drivers. Although it would be possible to bypass the provided layers of abstraction from within the access layer, great care was taken to supply the user with a way to access evidence that is more in line with real-world investigation processes.

It is considered good practice to place code that will be used by more than one application in an external library which may be accessed by applications as required. Not only does this help to reduce the size of individual applications, but it also creates the possibility of updating faulty library code with an updated library version, without replacing any of the applications, depending on the library being replaced.

Windows-based applications typically have dependencies on well-known dynamic link libraries, such as kernel32.dll, gdi32.dll and msvcrt.dll. Linux-based applications, on the other hand, typically have static linkages associated with libc, libstdc++ and libpthread. It is important to notice that although Windows and Linux use different designs, both operating systems allow their applications to be linked to external libraries.

From a forensics perspective the use of external libraries is of great concern as the external libraries may be modified or simply replaced by a version that adds a form of filtering, similar to the filtering supplied by root kits. Tools used by investigators that have to run in non-trusted environments should have their dependencies statically compiled into an executable file [3, 15]. Although the size of the executable will be considerably larger, the dependencies on non-trusted libraries would be minimized.

The technologies used to implement the platform were carefully chosen to allow the use of libraries that do not require the platform to link against external dynamic link libraries or static objects. All the chosen technologies actually allow static linkages between the libraries and the platform. This allows the functionality provided by the libraries to be linked into the compiled binary distribution.

From within the context of the access layer, various cases may exist, each with associated evidence objects. Various plug-ins may be applied to the evidence objects to produce visual results. These plug-ins are statically linked into the platform at compile time, which implies that the platform utilizes a static design. Each of the plug-ins should supply the platform with a specific interface. Figure 12 illustrates the specified interface that is expected of all access-layer plug-ins.

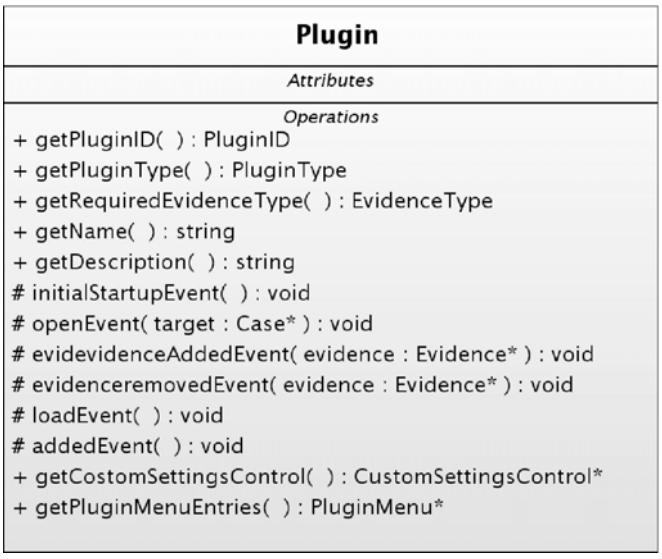


Figure 12: Interface provided by the Plugin superclass

Salus [89] describes the UNIX philosophy as: “Write programs that do one thing and do it well”. This philosophy should also be taken into account when developing plug-ins using the platform. Each plug-in should be able to perform a specific task, and perform it well.

Each plug-in is designed and built to process a specific type of evidence. A plug-in designed to process acquired disk images would not be capable of processing other evidence sources, such as network traces or memory dumps. For this reason it was decided to let each plug-in specify the type of evidence that it can process. The GUI therefore only allows a plug-in to be applied to evidence sources that it was designed to process.

Each plug-in may need configuration data to be supplied by the user, such as the type of operations that should be performed on evidence as well as temporary locations that may be used to store processing information. Each plug-in is allowed to supply the GUI with an object that will be used to manage the configuration of the plug-in. The purpose of the plug-in configuration object is to render the GUI controls needed to capture user input on a window control supplied by the platform.

The configuration object should also be able to verify and save the data supplied by the end user, when required. The use of the configuration object is optional; any plug-in that needs custom configuration data from the user needs to implement the getCustomSettingsControl() method defined in the Plug-ins super class.

When a user wants to apply a plug-in to an evidence source, he/she performs a right click operation on the evidence source of interest. A menu appears containing a list of plug-ins that are capable of processing the selected evidence source. Each plug-in is allowed to add entries to the right click menu by

implementing the getPluginMenuEntries() method. The entries specified by the plug-in will be listed every time a user performs a right click on an evidence source that the plug-in is capable of processing.

The GUI allows each plug-in to be notified when certain events occur. All that needs to be done by the user is to supply an implementation for the event handlers of interest. Events that can be processed by plug-ins are summarized in table 3.

Table 3: Plugin Events

Event:	Description:
initialStartupEvent	Triggered the first time a plug-in is added to a case.
finalCloseEvent	Triggered when a case is closed.
openEvent	Triggered each time a case is loaded.
evidenceAddedEvent	Triggered when evidence is added to a case.
evidenceRemovedEvent	Triggered when evidence is removed from a case.
loadEvent	Triggered each time a plug-in is loaded by a case.
removedEvent	Called just before a plug-in is removed from a case.

6.3.9 External tools

Various tools have been developed to illustrate the level of ease with which research prototypes can be developed using the platform. Table 4 contains the details surrounding the demonstration tools that were developed. The tool name, purpose, reference chapters and highest level of abstraction have been summarized to allow the reader to comprehend the purpose of the developed tools. Some of the tools were also developed as prototypes for research discussed in future chapters of this dissertation (see the chapter column in the table).

The development of the external tools indicates the ease with which the required level of abstraction can be chosen by the developer of a prototype. Four demonstration applications were developed. The real beauty of the layered architecture becomes clear when inspecting each application's usage of supplied layers of abstraction.

Two applications require the support provided by the interpretation layer while the other two rely on support provided by the abstraction layer. None of the demonstration applications rely on the functionality provided by the access layer. The demonstration applications therefore prove that developers are not forced to implement a specific design or use any components of the platform,

such as the supplied graphical user interface. This allows developers to create prototypes without having to implement or use functionality that they do not require.

Table 4: Demonstration applications developed using the platform

Tool:	Purpose:	Chapter:	Required layer:
Collector	Tool to acquire hard disk images.		Interpretation layer
LiveTool	Prototype that demonstrated how to develop a live acquisition tool using the platform.	8	Interpretation layer
RInspect	Tool to visually acquire and analyze disk images.		Abstraction layer
Timestamp Prototype	Tool used to calculate the last possible execution time of applications.	9	Abstraction layer

6.4 Future work

A considerable amount of work has been done to design a platform that conforms to the specified layered architecture, while allowing a user to easily use and extend it as required. The platform currently utilizes a static design which means that the platform cannot be extended through the use of external loadable plug-ins.

The use of a static design has advantages as well as disadvantages; future work will include a feasibility study to determine whether or not a static design in an open-source forensics context is desirable. The future study should also suggest alternative solutions as well as their advantages and disadvantages. A comparison between the various solutions should indicate the solution that is best suited for use within the open-source forensics context.

6.5 Conclusion

This chapter has described the design of a platform that conforms to the architecture defined in chapter 5. The design of each layer specified by the layered architecture was discussed in detail. Various design decisions were made based on the level of abstraction that users of the platform should experience. Users of lower-level layers may typically find themselves working with data that underwent little processing, such as blocks of data, while users of higher-level layers, such as the access layer, will typically work with abstract representations of information, such as information objects.

Very little was discussed about the low-level design of each of the components, such as the classes included in each component or the methods contained in each of those classes. This information was purposefully excluded from this chapter as there are currently hundreds of classes that are included with the platform; an in-depth discussion of those classes may obscure the focus point of this chapter which should contain only high-level concepts of relatively high importance. The platform documentation may be consulted for an in-depth discussion of the classes included with the platform.

The next chapter will discuss the implementation of the high-level design discussed in this chapter. The results obtained as well as screenshots of the implemented prototype will also be presented.

7 Prototype Implementation

7.1 Introduction

Previous chapters of this dissertation have been devoted to the definition of a platform that allows researchers to rapidly develop forensic research prototypes. Aspects regarding the development of such a platform, such as architecture and design were described, to allow the reader to gain a higher-level understanding regarding the development of a forensic platform that is capable of supporting the rapid development of forensic prototypes. This chapter describes the prototype that was developed according to the specified architecture and design, over a time period of 17 months.

The rest of this chapter is structured as follows. Section 7.2 discusses the use of various technologies which are incorporated into the platform. The use of each of the technologies is briefly discussed as well as the value that they add to the platform. Section 7.3 briefly discusses the platform documentation that describes the platform API. Section 7.4 describes the various platform distribution options offered to the end-users of the platform. Section 7.5 investigates various testing procedures that should be taken into account when testing the platform. Section 7.7 and 7.8 discuss the physical implementation of the platform.

Topics such as the functionality provided by the implemented platform prototype and various technical difficulties that had to be bridged are presented in order to highlight the complexity of the work that was conducted. The implemented platform's limitations are also discussed in detail. This chapter concludes with a short summary presented in section 7.9.

7.2 Technologies

As discussed in previous chapters, the purpose of the platform is to supply prototype developers with a rich set of basis components with the intention of promoting the rapid development of forensic prototypes. The platform was designed in a way to shield the platform user from unnecessary technical details. This was an attempt to decrease the amount of technical expertise needed to develop forensic tools, which may ultimately lead to the creation of prototypes at a more desirable cost in terms of time and resources.

Platform extensions developed by third party users should in effect be able to run on any operating system to which the platform is ported, as long as the plug-ins use only features provided by the platform. This is due to the fact that the platform prototype provides the same classes and interfaces to plug-ins in all supported operating systems. At the time of writing, Linux and Windows are the only supported operating systems; support may be extended to other operating systems if such a demand exists in the security community.

Various well-known technologies were included in the platform to supply

functionality that may be of value to users. Table 5 gives a brief summary of the technologies used and their contribution to the platform. The technologies used by the platform helped to decrease some of the development complexities, but a considerable amount of code still had to be developed in order to provide a platform that could actually be considered to be useful by members of the security community.

Table 5: Technologies used by the platform

Technology:	Purpose:
FreeDOS32	The FreeDOS32 FAT driver is used to supply FAT file system support to the platform.
Theodore Ts'o LibEXT2FS	LibEXT2FS provides EXT2/EXT3 file system support to the platform.
L. Peter Deutch's MD5 implementation.	The MD5 implementation supplies MD5 calculation support to the platform.
SQLite	SQLite provides lightweight database support to the platform.
MySQL and PostgreSQL	MySQL and PostgreSQL provide heavyweight database support to the platform.
wxWidgets	wxWidgets provides a cross-platform windowing toolkit that is used by the platform to supply user-interface support.
Zlib	Zlib is used to compress and decompress relatively large files used to store settings (such as the file used to store cross-case plug-in settings).

It should be evident that a number of well-known database engines are supported by the platform to store case-related information. The choice to support various database engines stems from the fact that different platform users will have different storage requirements. As an example, consider the requirements of a single user working in a closed environment as opposed to a group of individuals working in a distributed environment.

Heavyweight database engines, such as MySQL and PostgreSQL allow multiple users to simultaneously connect to the same database, from various locations. When used in conjunction with virtual private network technologies, it would actually be possible for various users to analyze the same pieces of digital

evidence at the same time in a distributed environment. The platform can therefore be used in multi-user distributed environments if the required infrastructure is provided and configured correctly.

The configuration of such a solution may be a daunting task which requires a relatively high level of skills and resources to perform. The use of such an approach may seem unnecessary in situations in which collaboration with other users are not required. It was therefore decided to also include support for a lightweight database engine, to allow platform users simple requirements to use the platform without having to actually install and configure heavyweight database servers and virtual private networks.

Thousands of lines of code were developed in an attempt to provide a platform that may be of value to the forensic research community. In an attempt to quantify the amount of effort that went into the development of the platform, a line count was performed on the existing code base. Table 6 describes the total amount of lines of code (SLOC) revealed by a line count performed on the platform release 0.41.

Table 6: The Platform's SLOC

Source File Type:	Purpose:	SLOC:
C++ Source Files	Platform support functions.	46481
C/C++ Header Files	Provide the interfaces implemented by the C/C++ source files.	21348
C Source Files	The bulk of the C source files were included from the SQLite project to provide lightweight database support.	87056
Total:		**154885**

Although the SLOC count cannot be considered an effective measurement tool to describe a software product's complexity or quality [87], it gives a rather good estimate of the amount of effort that was required to deliver the finished product. All C source files were taken from external projects, while a large portion of the C++ source files and header files (34834 lines of code in total) were the products of custom development in an attempt to implement the architecture and design defined in previous chapters of this dissertation.

7.3 Documentation

The Reco Platform has been developed as part of this dissertation. The source code has been well documented using doxygen-style comments to improve it's understandability. Doxygen produced 397 pages of source documentation by extracting comments found in the source code. This extent of pages should indicate that the platform code is well documented, which will allow developers to take advantage of the supplied platform features in a relatively short period of time.

Various platform usage tutorials and platform development tutorials have also been created. The purpose of these tutorials is to allow the user to understand how the platform is used from an end-user's perspective as well as from a developer's perspective. User-centric tutorials focus on the physical use of features supplied by the platform, such as creating cases or adding evidence to cases. Developer-centric documentation, on the other hand, is more technically focused and explains how plug-ins can be developed for the platform. These tutorials are available on Sourceforge [56] as well as the accompanying disk.

7.4 Package distribution

The entire platform is distributed as a single compressed archive which contains all the source files needed to build the platform using freely available development tools. The archive is available on Sourceforge and can simply be downloaded, extracted, configured and built by developers interested in using the platform. Cutting-edge versions of the platform are also available through the Sourceforge Subversion repository.

A pre-compiled Windows-based version of the platform has also been made available for downloading. The pre-compiled version is distributed along with a Windows-based installer to allow for easy installation of the platform. The purpose of the pre-compiled distribution is to allow interested users to view the platform without having to build the project source code.

Different system configurations may cause problems when an application is built or executed. Obvious errors, such as missing libraries or header files can be detected at compile-time by the compiler. Detectable errors are a minor cause for concern, as the compiler will usually inform the user about the error that exists. To fix the problem, the user is usually required to install the missing files, or change some configuration settings if the files are already installed. Detectable errors are therefore nothing more than an inconvenience to the user.

Unfortunately a more dangerous threat exists: undetectable configuration errors. If such an error exists, a binary will be generated by the compiler that seems to be in working order. This presence of such a configuration error will typically only become evident once the binary produces detectable incorrect results. An example of such a silent threat is the compilation of code developed for 32-bit machines on 64-bit machines.

At first glance this should not pose a threat to the code compilation process. Unfortunately this is not the case, as the size of some 64-bit elementary data types is larger than their 32-bit counterparts. The danger in this change in type sizes lies in the fact that data structures that contain these elementary types will be larger on 64-bit systems than on 32-bit systems. This misalignment will cause a 64-bit program to be incompatible with the same structure used in the equivalent 32-bit system.

To overcome this problem, as well as various other types of configuration problems, a configuration application is distributed with the source code to automatically detect and correct problems that may exist, which may cause a build to fail. The configuration script is responsible for the creation of a custom makefile that can be used to build a binary for the specific target platform on which the source code will be compiled.

Unfortunately the configuration script is available only for use on UNIX-based systems. This implies that Window users may have a slightly harder time compiling the source code, as the automatic configuration script is not able to function in the Windows environment.

Fortunately a sample makefile is distributed with the platform that can be used to compile the platform under Windows. Although some minor modifications will have to be made to the makefile along with some configuration files in order for the build to succeed, it would be possible to generate a Windows-based binary from the platform source code.

7.5 Testing

Testing is a crucial part of any development process as it allows the developers to uncover and correct program errors. The successful identification and correction of errors requires an understanding of an application's requirements, architecture and design to allow the developer to identify areas of implementation that deviate from the planned product. In an attempt to identify errors with greater efficiency, a deeper understanding of the concept of program errors is required, which may help developers to diagnose and describe such errors more efficiently.

Two types of errors may be introduced in the software development process, namely implementation errors and abstraction errors [13]. Implementation errors are a result of the use of vague or incorrect specification in the development process [14]. Application developers are not always security experts and may have limited knowledge of processing techniques important to digital forensics [23]. Implementation errors may therefore occur as the developers may fail to fully understand the complexity surrounding the development of an application with forensic processing capabilities.

Abstraction errors are usually introduced as a product of decision making based on uncertainty [14]. A typical cause of abstraction errors is making decisions based on data produced using data reduction techniques. Because a section of data is disregarded in such a process, the possibility is introduced that

an incorrect conclusion may be reached. A good example of an abstraction error is a situation in which a forensic tool copies a file using standard operating system libraries that was intercepted by a root kit.

The operating system helps to create an abstract view of the file system; unfortunately this level of abstraction may be manipulated by root kits to return any data to the tool in question as it pleases. It would be virtually impossible for the tool to detect such a problem due to the level of abstraction on which it operates. In this example, it occurred that the developer of the application failed to consider the impact that the operation of a forensic application, operating on such a high level of operating-system abstraction, may have on the validity of data.

From the discussion it should be clear that implementation errors are caused as a direct result of the programming done by the developer; abstraction errors may be caused by external factors, which may not always be under the control of the developer. This implies that implementation errors could be located through debugging while other techniques, such as verification, may be suitable for detecting abstraction errors.

Comprehensive long-term testing strategies may help to improve the quality of forensic applications [14]. To design a comprehensive and effective testing strategy, attention must be given to the type of tests that are performed. A lot of time and energy have already been invested in the development of test strategies.

The NIST has a dedicated work group called the Computer Forensics Tool Testing (CFTT) group [14] which is dedicated to the development of test methodologies to ultimately enhance the quality of forensic tools. It is therefore clear that test methodologies exist, but it is unclear which type of forensic errors these tests attempts to uncover. A closer look will therefore have to be taken at the types of forensic errors that may occur in order to get a better idea how to identify and eliminate such problems.

Two types of digital forensic tests need to be performed, namely the test for false negatives and the test for false positives, to determine if a given procedure produces accurate results [14]. False negative tests are performed to determine if a tool will return all possible input data. As an example, consider an application that displays the names of deleted files. An application that returns only some of the names of the deleted files will surely fail the false negatives test, as it is extremely important for the investigator to know about the existence of all deleted files, even if some of them cannot be recovered.

False positive tests are performed to ensure that the tool in question does not introduce additional data to the expected program output. An example of a false positive failure may be in the case where additional files are added to the list of deleted files that do not exist or were not deleted at all.

The test for false negatives is relatively easy: simply plant data in a prepared evidence image and determine if the application is able to detect it. The test is relatively simple because the variables surrounding the test are

known: the creator of the test knows how many pieces of data have been planted. If the application detects all the pieces of data, it passes the test; if it does not detect all the pieces of data, it fails the test.

The test for false negatives requires an extensive collection of test data that includes all known data permutations that is likely to cause systems to fail. Any condition that would cause a system to fail which is not included in the test data collection can be seen as a risk to the validity of the results obtained through the execution of the false negative test due to the fact that the application were not tested for potentially problematic situations.

Testing for false positives is not as simple as testing for false negatives. This is due to the fact that it cannot be determined beforehand under which circumstances additional data was introduced to produce output, without examining an application's source code. Since popular forensic applications are closed-source, a method is required to test for false positives without having to inspect the application's source code.

Carrier [14] recommends that the output produced by one application be compared with the output produced by another application that is considered to be correct. The only problem with this approach is that the false positives will go undetected in the application in question if the problem also exists in the application against which validation is performed.

The prototype was not tested vigorously, as more important aspects of development had priority over the testing process. All platform features were tested on an ad-hoc basis, but no formal testing, such as the test for false negatives or false positives, has been done yet. Due to the fact that the platform serves as an academic proof-of-concept, this lack of structured testing can be considered to be acceptable. It should be borne in mind that the platform will have to undergo vigorous testing exercises before it can be declared suitable for commercial use.

7.6 Prototype

This section describes the standard functionality provided by the prototype from an end-user's perspective. The use of standard platform building blocks is discussed to allow the reader to form a mental image of the platform from a non-technical point of view.

The platform supplies standard functionality that cannot be altered or replaced through the development of plug-ins. This standard functionality provides services to the end user that are crucial for the successful operation of the platform. Basic functionality supplied by the platform, such as case management, evidence management and user management, may be utilized by platform end-users to manage the way that the platform and its plug-ins operate. The following is a complete list of services provided to platform end-users:

- Case management;

- Evidence management;
- Background process management;
- User management;
- Plug-in management and configuration.

The standard functionality provided by the platform from an end-user's perspective is critical to the management of cases, as it can determine which cases exist, which evidence and which users are added to those cases, and which plug-ins may be applied to evidence sources. The following subsections provide a more detailed overview of the standard platform functionality from an end-user's perspective.

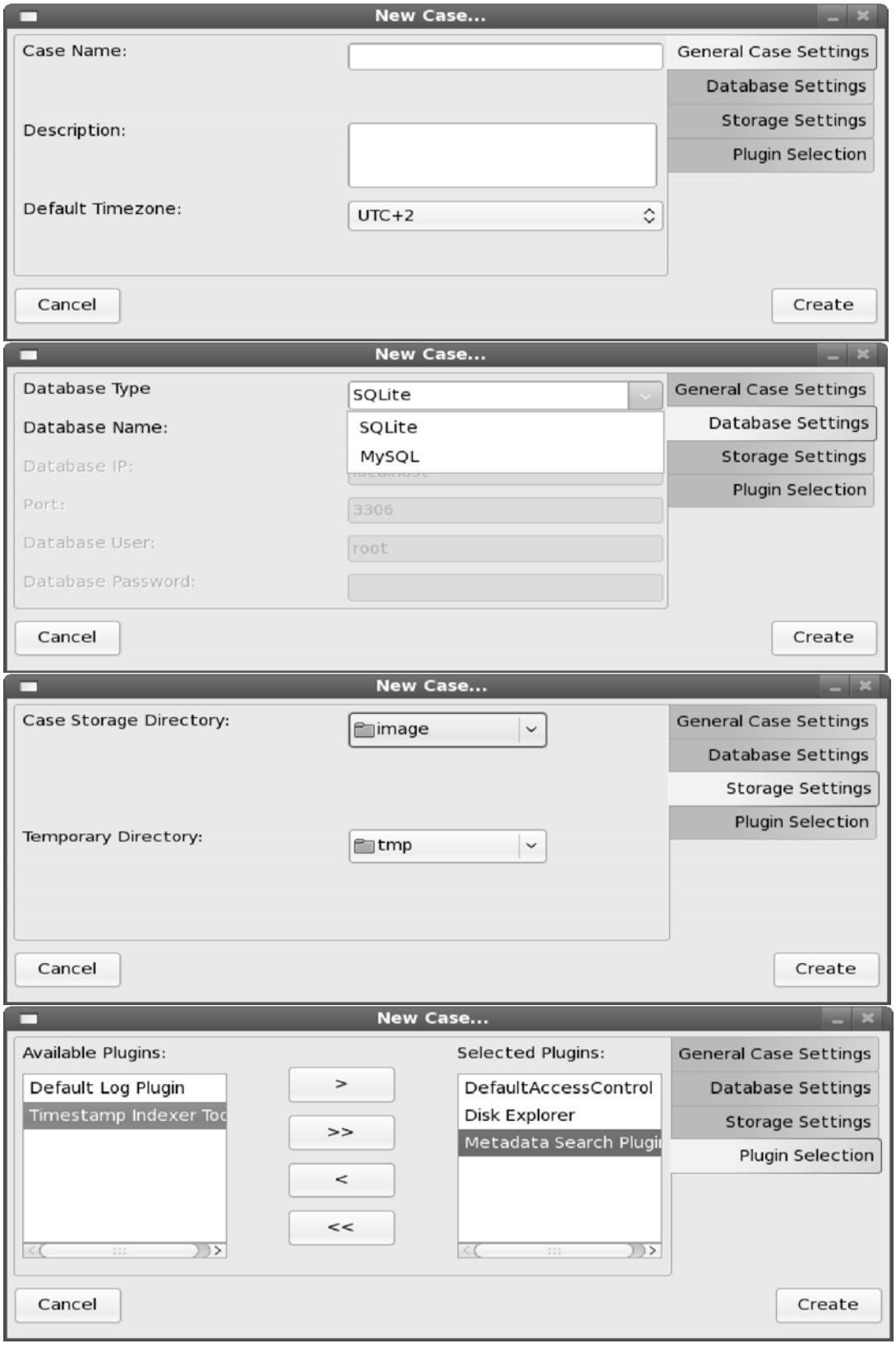
7.6.1 Case wizard

The case wizard allows the user to easily create cases by supplying the platform with mandatory information necessary for the case creation process. The following information is required in order to successfully create a new case:

- Case identification information, such as the case name, description and default time zone;
- Database related information, such as database type, database location and the applicable connection information;
- Physical storage settings, such as temporary directory and directory used to store captured evidence files;
- A list of plug-ins that may be applied to a case.

Figure 13 contains screenshots of the physical design of the case wizard used to create new cases within the platform. Close inspection of the figures reveals that the user is only allowed two options when choosing a database engine, namely SQLite and MySQL. This is due to the fact that the platform was designed to be configurable; functionality can simply be removed as necessary, in an attempt to decease the size of generated executable files.

Figure 13: Four screenshots of the case wizard



The user is allowed to specify various options surrounding the platform, such as supported file system types and database engines, before the platform source code is compiled. This allows users to generate binaries that are customized to their requirements. The platform used to create this example was compiled without PostgreSQL support in an attempt to illustrate the configurability of the platform.

7.6.2 Adding evidence

Acquired evidence sources, such as disk images or captured network traffic needs to be added to a case before it can be inspected using the platform. Adding evidence to a case is a simple process in which the investigator specifies an evidence source file, identifies the evidence type and gives a brief description of the evidence at hand. Figure 14 displays the dialog that captures evidence information from the platform end-user. Once evidence has successfully been added to a case, it can be inspected using the functionality provided by various platform plug-ins.

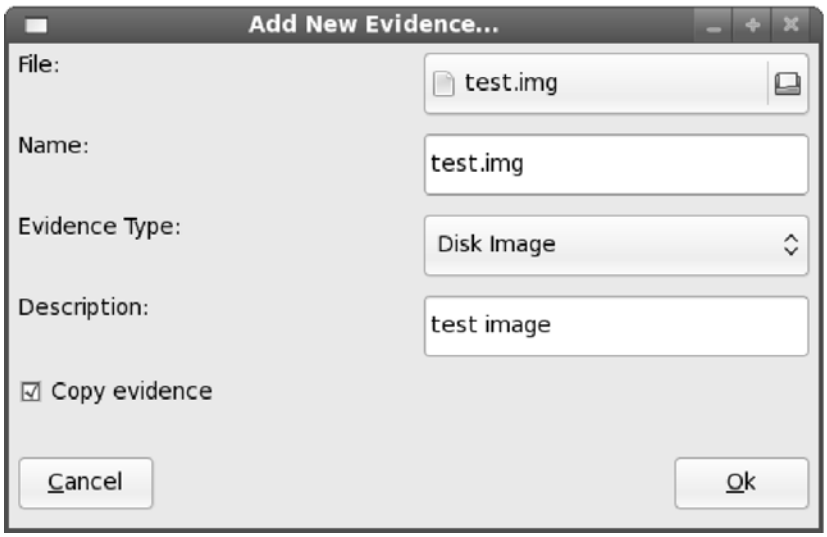


Figure 14: Adding evidence to a case

When the platform end-user performs a right click operation on an evidence source (see figure 15), a list of plug-ins appears in a generated menu, containing all the plug-ins that can be applied to the selected evidence type. The user may choose to apply any one of the plug-ins to the selected evidence source.

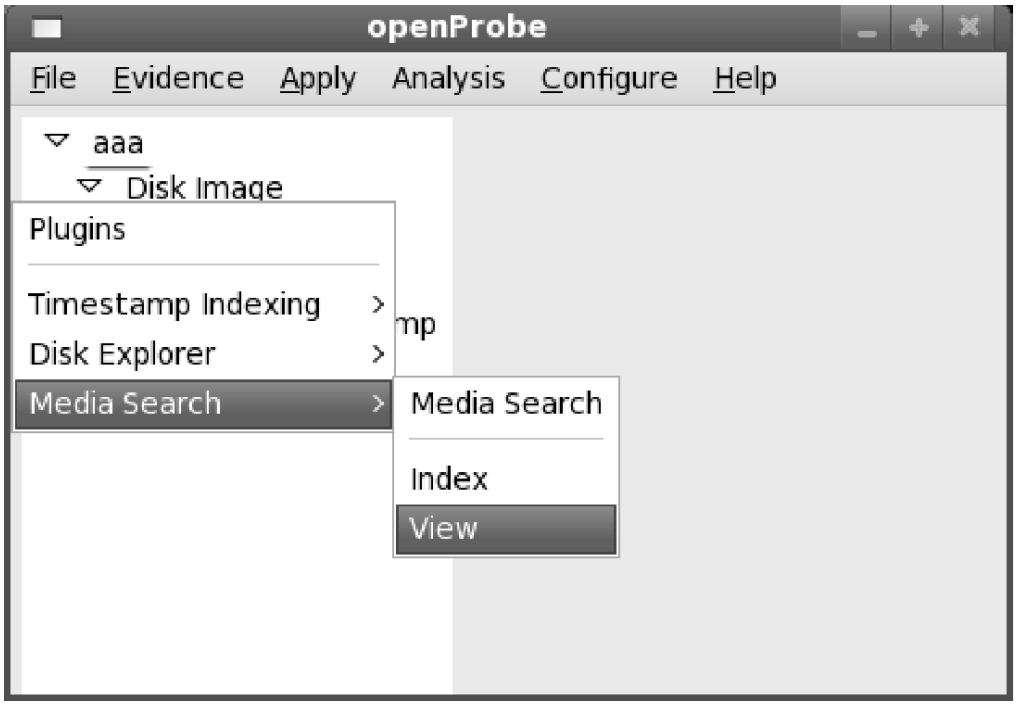


Figure 15: Applying a plug-in to an evidence source

Plug-ins may be configured by the platform end-user to behave in a certain manner. To accommodate this requirement the platform supplies the functionality which allows the user to customize various settings associated with each plug-in that may be applied to a specific case.

Figure 16 illustrates the plug-in configuration facility provided by the platform. Each configurable plug-in receives a panel on which it can render the controls that it uses to capture configuration settings from the user. This allows each plug-in the freedom to capture whichever data is needed to operate successfully.

7.6.3 Plug-in configuration

Plug-ins may be configured by the platform end-user to behave in a certain manner. To accommodate this requirement the platform supplies the functionality which allows the user to customize various settings associated with each plug-in that may be applied to a specific case.

Figure 16 illustrates the plug-in configuration facility provided by the platform. Each configurable plug-in receives a panel on which it can render the controls that it uses to capture configuration settings from the user. This allows each plug-in the freedom to capture whichever data it requires to operate successfully.

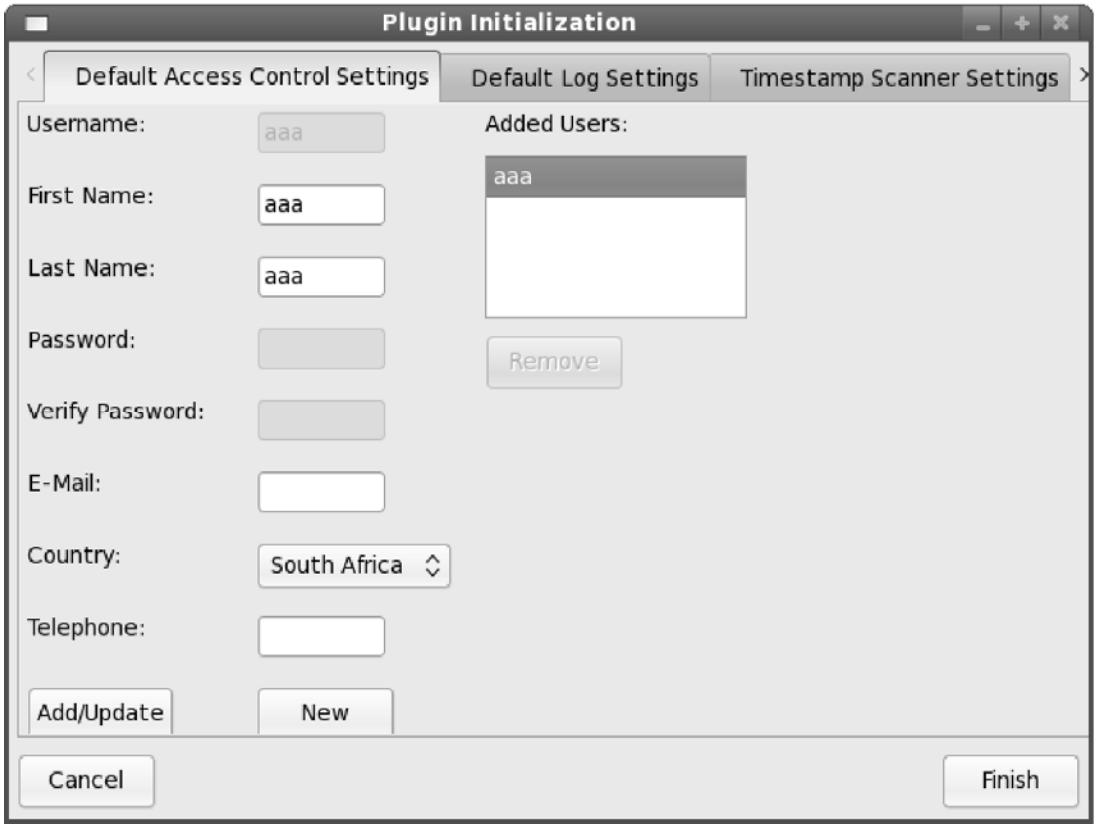


Figure 16: The plug-in configuration facility

7.6.4 Plug-ins

Various plug-ins were developed in an attempt to showcase some of the features provided by the platform. The plug-ins distributed with the platform may seem to be relatively primitive, to say the least. The simplicity of the developed plug-ins is not necessarily a weakness, as they serve as programming examples due to the fact that their inner workings are relatively simple to understand. The following subsections discuss the plug-ins, developed as part of this dissertation, and distributed with the platform, namely the TimestampTool, DiskExplorer and the SearchTool.

TimestampTool

The purpose of the timestamp tool is to create a searchable index of file creation, access and modification times. Indexed searches have the advantage over live searches, due to the fact that fast database queries are performed when performing indexed searches, as opposed to slow disk search operations performed with live searches. A further disadvantage of live searches is that they require that the entire evidence source is searched every single time a search query is executed.

The timestamp tool displays a grid containing the following indexed file information for each file found on the file system:

- The name of the file;
- Creation time;
- Access time;
- Modification time.

The timestamp tool has to create an index before any timestamp information can be displayed. The index creation process may be time-consuming due to the fact that the content of the entire disk image needs to be processed, which may potentially take up to several hours to complete. To address this issue, it was decided that indexing can be performed as a background task.

This allows the platform user to perform other investigation tasks while an index is being created. Figure 17 displays the output produced by the timestamp tool plug-in. The plug-in displays the creation time, access time and modification of each file found on a disk image. This information may be of value to investigators when constructing time lines by using file timestamps.

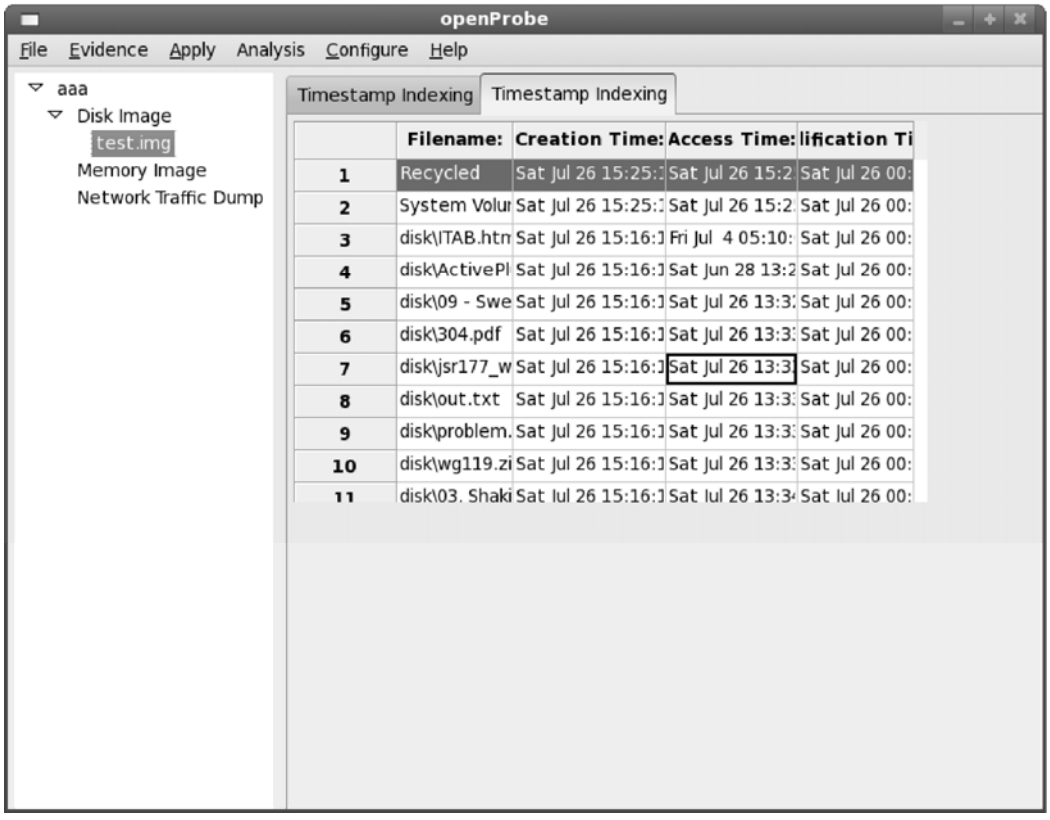


Figure 17: The TimestampTool plug-in

The disk explorer is an interactive plug-in that allows the user to browse the file structure of an acquired disk image. Unlike the timestamp tool, the disk explorer depends on user input to generate results. It displays a tree that depicts the file structure of the acquired disk image; directories are represented by tree nodes while files are represented by leaf nodes.

The plug-in also allows the user to view a file's associated attributes by simply clicking on the file of interest. Attribute-related information, such as file creation, modification and access time will be displayed at a designated area when the user selects a file. Figure 18 is a screenshot of the interface used to browse the file structure of a captured disk image.

DiskExplorer

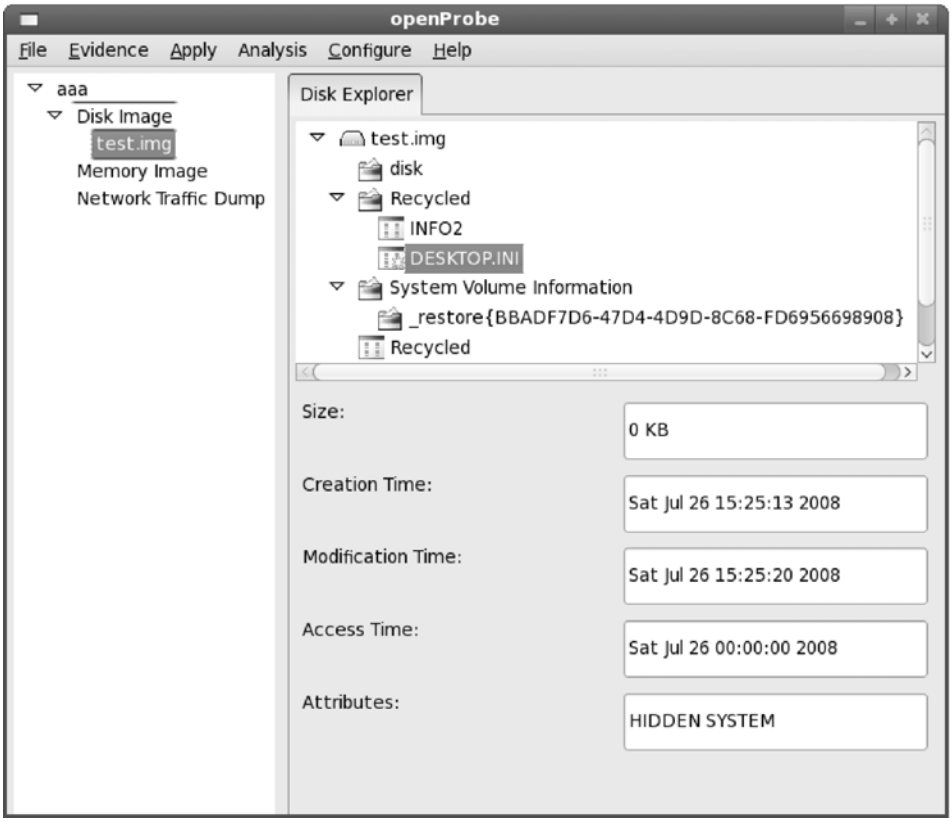


Figure 18: The DiskExplorer plug-in

The disk explorer plug-in allows the end-user to inspect the disk structure in an intuitive manner for the presence of specific files. Although the disk explorer plug-in lacks the functionality to actually extract files from the disk image, it can be argued that the functionality can simply be added to the plug-in due to its modular design.

SearchTool

The search tool plug-in allows users of the platform to search database tables for specific values. All matching results are displayed to allow further inspection. The search tool plug-in makes use of the search functionality supplied by the platform search plug-ins. Since the search functionality is available to all platform plug-ins, it actually is possible to re-use the search capability supplied by the search plug-ins.

The results obtained by the search tool plug-in are displayed exactly as they are obtained by the search plug-ins, which means that the information displayed is the same information that would be supplied to any of the platform plug-ins that makes use of the supplied search functionality. This would allow potentially any plug-in to possess powerful searching capabilities with minimum implementation effort. A screenshot of the supplied search functionality is presented in figure 19.

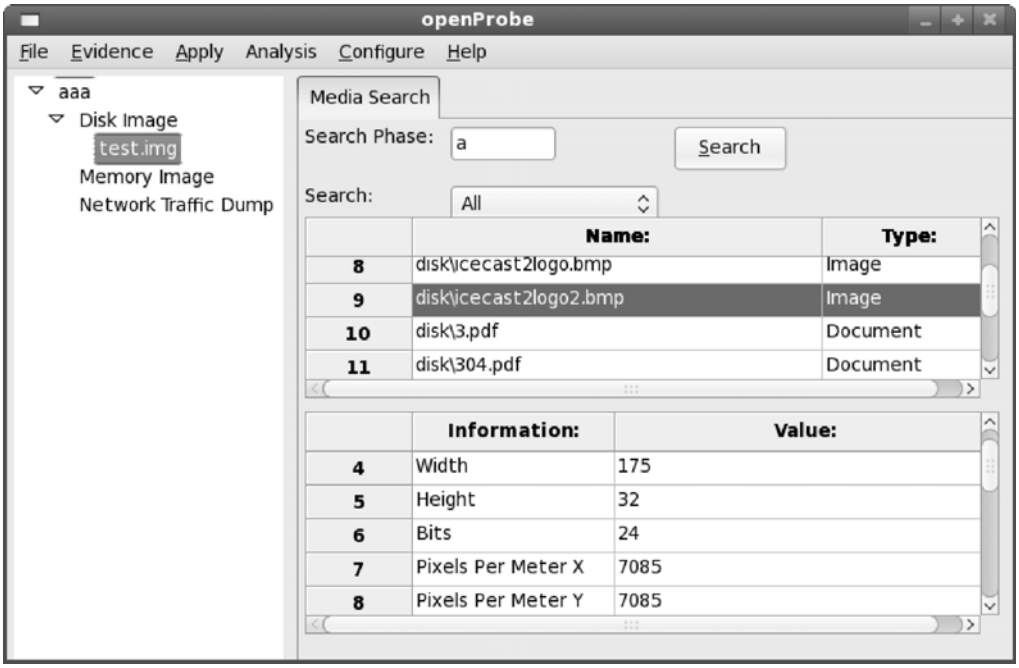


Figure 19: The SearchTool plug-in

7.7 Technical difficulties

Numerous technical difficulties were experienced while developing the platform. Although these did slow down the development of the platform to some extent, they did not manage to stop the development completely. Some of the difficulties experienced with the development of the platform are discussed in more detail in this section.

One of the major challenges faced in developing the prototype was to

merge various seemingly unrelated open-source projects into a single cohesive unit. Conceptually, the integration of various open-source technologies may seem to be a relatively simple task to perform. Unfortunately it was not as simple as it may seem at first.

This is due largely to the fact that open-source projects are generally developed for a specifically designed purpose, in a rather specific environment. The integration of open-source projects is therefore challenging due to the fact that software technologies are often taken from the environments that they were designed to work and function in, and are placed within a different context, with a seemingly different task to perform.

The integration of the FreeDOS FAT file system driver can be considered to be one such example. The FreeDOS operating system is an open-source clone of the DOS operating system. All services used by any of the drivers in the operating system should be supplied by the operating system kernel. The file system driver shipped with the FreeDOS operating system has been designed with this constraint in mind. Various popular functions typically associated with libc, such as memcmp and memcpy are supplied by the FreeDOS project.

From an operating system perspective this design makes sense, as an operating system cannot possibly make use of precompiled libraries that were not designed specifically for the operating system under development. Unfortunately this design causes problems in situations in which any of the operating system code is used for purposes that it was not designed for.

The extraction of the FreeDOS FAT driver required a few modifications to allow it to make use of the standard functions supplied by libc. The modification removed the dependency that the driver had on the basis supplied by the FreeDOS operating system. Unfortunately other dependency problems exist that not only caused problems when compiling the FAT driver, but also caused the compilation failure of other open-source components.

The FAT file system driver is shipped with a non-standard version of types.h which clashes with the standard version of types.h distributed with popular compilers. The standard version of types.h contains type declarations of various commonly used structures. Unfortunately the types declared in the version shipped with the file system driver are not compatible with the types declared in the standard header file. If the types.h shipped with the driver is left as-is, the FAT file system driver compiles without any errors. The real problem is the effect that the non-standard types.h file has on other open-source components.

Other open-source components also make use of the types declared in types.h. The makefile used to compile the platform allows any piece of code to reference any header file included in the project. From a developer's perspective this is convenient, as all header files can be used without having to supply excessively long or unnatural header file path prefixes when referencing a header file in a different directory.

The side-effect of this configuration is that it is difficult for the compiler to

make a clear distinction between header files with the same name. The standard types.h and non-standard types.h would therefore be visible to any project that references the real types.h. The compiler would respond to the problem by complaining about the fact that various data types that have been declared in the standard types.h have already been declared in the non-standard version, and visa-versa.

The problem was solved by removing type declarations from the non-standard types.h that matched the declarations in the original types.h. In addition to this, reference to the original types.h was added to the non-standard version, which in effect means that the non-standard version is a subset of the standard version, with some additional modifications.

Some minor checks were included in the non-standard version in order to add additional types to the non-standard version if they are not supplied by the standard header file. This configuration allows the FAT driver to compile while solving issues experienced with the compilation of other open-source components.

Another technical challenge was to guarantee the portability of all platform code between Linux and Windows systems. The idea is conceptually simple: any piece of code that depends on only the platform should be able to compile and run in both Windows and Linux without requiring the developer to modify any code or make extensive use of pre-processor flags.

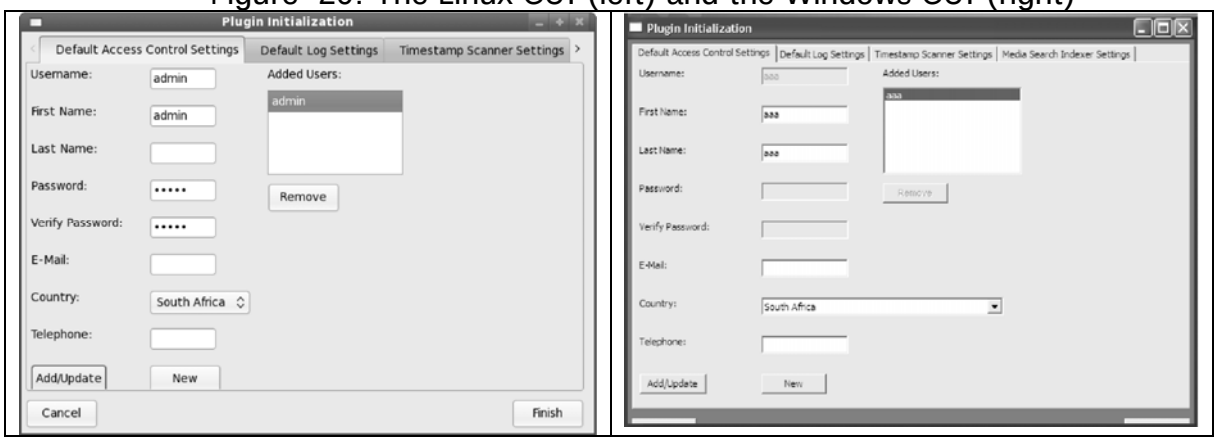
Unfortunately both Linux and Windows applications make use of APIs that are specific to the operating system in question. This means that the API available on one operating system may not necessarily be available on the other operating system. To overcome this problem, various base classes were developed that contain functional code equivalents for both Windows and Linux; the Linux-based code is compiled and used by Linux-based compilers, while the Windows-based code is compiled and used by Windows-based compilers. All code using the base classes are therefore (in theory) portable between Windows and Linux. The platform documentation provides more information on this topic.

Platform windowing capabilities are not supplied by the cross-platform base classes. The help of another framework was therefore required. wxWidgets [109] was used to supply cross-platform window management capabilities required by the platform's access layer. The use of the platform's base libraries, in conjunction with the windowing capabilities supplied by wxWidgets, makes the development of cross-platform plug-ins a reality. Minor interface inconsistencies may exist between Windows and Linux ports. Figure 6 illustrates the difference between standard platforms interfaces displayed on Linux, on the left, as opposed to Windows, on the right.

A quick comparison between the two interfaces (in figure 6) reveals that the spacing, look and size of the GUI components vary slightly between operating systems, causing the GUI layout to look different depending on which host operating system the platform executes. As can be seen from figure 6, the Cancel and Finish buttons are semi-hidden on the Windows-based application

screen, while they are perfectly visible on the Linux-based screen. Plug-ins should therefore be tested in both Windows and Linux to ensure that layout issues do not have a negative impact on the usability experienced with the implemented plug-in.

Figure 20: The Linux GUI (left) and the Windows GUI (right)



Another technical problem that wreaked havoc on the stability of the system was the issue of 64-bit portability. As described previously, the sizes of primitive types may differentiate between 32-bit and 64-bit systems, causing the size of structured types to vary between 32-bit and 64-bit systems. Data structures containing primitive types are often loaded from or written to data files. The differentiation in structure size may ultimately cause the same structure used on a 64-bit machine not to load data from the same structure written on 32-bit machines.

This type of problem is very difficult to detect and diagnose, as the compiler will not give any indication of the possible existence of porting issues, without explicitly telling the compiler to perform portability testing. The problem was solved by instructing the configuration application, which generates the makefile used to build the platform, to check the sizes of various primitive types used by the system. Pre-processor flags in header files allow the compiler to use the correct primitive sizes for the platform in question according to the values detected by the configuration script, to allow 32-bit and 64-bit structures to be compatible.

The size of the platform binary generated by Windows-based compilers is by default relatively large. The Windows-based version of g++ generated a binary of 23Mb, which places quite a burden on the storage requirements of the platform. The size of the executable is caused by the fact that the wxWidgets library is statically linked in an attempt to minimize the use of external libraries that need to be installed by the end-user.

As previously indicated, this decision is of value in an investigation setting,

as it allows an investigator to be able to make use of a single executable without any DLL dependencies. An investigator can therefore simply run the application on a computer without having to install any required runtimes.

To improve the size-related problem, the strip [38] command was used to discard unnecessary symbols from the executable, which dramatically reduces its size to an acceptable level. To decrease the size of the executable even further, it is compressed using the well-known executable file packer called UPX [77].

UPX makes it possible to reduce the size of an executable file by compressing data contained therein. A small loader program is attached to the compressed data to decompress and load the executable at runtime. The use of UPX therefore decreases the amount of disk space required by the platform.

The application of the strip command in conjunction with UPX resulted in decreasing the size of a 23 MB executable to a mere 1.9 MB. The fact that the executable is relatively small and has no dependencies on custom DLL files, other than the DLL files distributed with Windows, makes the platform an extremely attractive option for acquisition tool development, as acquisition tools have to have a relatively small disk footprint with very few external dependencies.

The test was also performed on a 64-bit Linux system (Fedora 9). An 10.4MB executable was generated as a result of the compilation process. The application of the strip command and UPX reduced the size of the executable to a mere 524KB. Please be advised that it would be unlikely to reproduce the results on other Linux systems and distributions due to different software configurations.

7.8 Platform limitations

The platform was designed with flexibility in mind. Users should be able to add functionality to the platform as required, while making minimal changes to the existing platform code base. The platform supplies an environment in which developers can create forensics-related plug-ins. The platform can therefore be described as an environment that aids the rapid development of cross-platform forensics plug-ins. This implies that the platform is only concerned with the provision of services to plug-ins using the platform, and not to the direct provision of services to the end-user.

From an end-user's perspective the functionality provided by the platform may seem to be rather limited and primitive. This is due largely to the fact that the purpose of the platform is only to provide services that are necessary to allow the plug-ins developed for the platform to work as required. A few, rather simple plug-ins have been developed and distributed with the platform. The purpose of these plug-ins is not to supply useful functionality to end-users, but rather to serve as examples for platform developers that are interested in developing similar plug-ins.

It might be argued that the flexibility of the platform can also be considered as one of its main weaknesses, due to the fact that it does not enforce or implement any forensics-specific methodology. Although this may

seem to be a weakness, it can be argued that functionality can be developed through the creation of plug-ins that can aid or enforce the use of specific forensic methodologies.

As described by the platform architecture, the development of plug-ins should be done by the developer on the platform access layer level. Due to the fact that a developer is allowed to choose the level of abstraction that is required for the development of a forensic prototype, it can be argued that the use of methodologies can be bypassed since he/she may simply choose to work on a lower level layer on which the functionality is not available.

Fortunately the source code of the platform is available to anyone to modify and improve as required. If specific functionality, such as the enforcement of methodologies, is required on every single platform layer, it can simply be added by a skilled developer. It should be noted that the purpose of the platform is not to supply every single feature known to the world of forensics, but rather functionality that is likely to be needed by prototype developers. Functionality that is not supplied by the platform can simply be added as required.

Another platform limitation is the fact that it uses a static design. The addition of new plug-ins to the platform basically requires the platform to be re-compiled to contain the new functionality.

The inclusion of file system drivers that were not developed for forensic purposes in the platform's array of file system drivers remains a moot point which is open for debate. The inclusion of open-source file system drivers allows for the quick addition of new file system support. The use of stable existing drivers allows the platform to take advantage of the investment made by other developers when they created the drivers. Well-known drivers also tend to be well documented and their strengths and weaknesses are well known. This means the behaviour of the existing drivers are likely to be well defined and work as they were designed to.

Unfortunately very few commonly used file system drivers have been designed for use within a forensic context. Features such as error reporting, recovery and data reconstruction would therefore not be supported directly by such drivers. This implies that the use of open-source drivers will only be of value in situations in which the file system image under investigation has not been damaged to an extent to which drastic measures are needed to access the corrupt data.

Like every other design decision, this decision has its advantages and disadvantages. If the design decision proves itself to be an extremely limiting factor that has a negative impact on the usability of the platform, a redesign will have to be performed to improve problem areas in the design.

Java and the .NET platform are currently extremely popular language choices among developers. Although the platform was developed in C++, it would in theory be possible to introduce technology that would make it possible for applications written in Java or one of the .NET languages to interact with the

platform. The creation of such technology would allow the inclusion of a larger developer community, which would in turn allow the platform to be positioned for wider use.

Unfortunately the development of the binding technology has not been undertaken as yet, since the development of the platform currently has higher priority than the extension of the platform to allow its use in other programming languages. If a need exists in the future for such a binding technology, a development effort may be undertaken to develop the language bindings required to allow other programming languages to interact with the C++ code in the platform.

7.9 Conclusion

This chapter has described the prototype that was developed according to the architecture and design specified in previous chapters of this dissertation. A technology discussion was conducted to describe the technologies used by the platform, as well as the amount of effort that went into the development of the current version of the platform. The platform documentation and package distribution method were also discussed, to allow the reader to understand the various options that exist to obtain copies of the platform.

Various platform features were discussed in this chapter. Each of the feature descriptions was accompanied by a simple screenshot to improve the reader's perception regarding the functionality that was implemented. Features such as the case wizard, evidence management, plug-in management as well as various existing plug-ins were discussed in an attempt to explain the current functionality provided by the platform.

Various implementation issues were discussed in detail. Among the issues addressed was the 32-bit to 64-bit porting problem, the Windows-Linux compatibility issue, and the problems experienced with the merger of various unrelated open-source projects into a single cohesive unit.

The prototype limitations were also discussed in an attempt to document the current known weaknesses of the platform. Possible design flaws, such as allowing the platform to be too configurable, and the inclusion of open-source code that was not designed to be used in a forensic context, were discussed in great detail to allow the reader to understand why various design decisions were made, as well as the possible impact that such decisions may have on the platform.

The development of the platform was an extremely time consuming exercise that delivered a product with basic functionality needed to aid the rapid development of forensic research prototypes. The platform can be of great value in the development of forensic prototypes, due to the fact that the support structure provided by plug-ins developed using the platform is relatively complete and allows the user to develop code with advanced functionality, with relatively little effort. Future effort will be devoted to the development of plug-ins to allow platform end-users to experience the same level of power that users of

commercial forensic software currently enjoy.

8 A Live-System Forensic Evidence Acquisition Tool

8.1 Introduction

Chapters 8, 9 and 10 serve as proof-of-concept chapters that report on research that was conducted using the platform. This is an attempt to prove that the platform can in actual fact be used to produce research of relatively high quality, requiring less time and utilizing fewer resources. This chapter describes the development of a prototype that utilizes the functionality provided by the developed platform prototype to perform live evidence acquisitions.

The trade-offs between dead and live forensic analysis are well known [15, 63]. The technique known as “dead analysis” is used to remove digital evidence from an environment in which it can be modified inadvertently before analysis is performed. The removal of digital evidence from such environments may be linked with information system downtime, which may cause financial and other non-monetary losses to innocent parties.

Live forensic investigations have been proposed as an alternative [15, 63] to dead analysis. Live forensic investigation techniques allow the investigator to perform evidence analysis while the information system in question is kept running. Unfortunately this technique also has limitations due to the possible presence of intermediaries, such as rootkits, which may posses the capability to modify any data that the investigator observes.

The purpose of a rootkit is to act as an information filter [15]; rootkits may intercept system requests in an attempt to control the flow of information in a computer system. The fact that digital evidence information may be filtered in such a manner makes live forensic analysis particularly unappealing [3]. Even if a rootkit is not present, the mere fact that an untrusted piece of code, in the form of a normal operating system service, has been asked to retrieve the forensic data casts some doubt on the validity of the data.

In any operating system, services exist on various layers; any rootkit may in principle, hijack one or more services on any of these layers. In particular, a rootkit hides its presence by modifying several system services. It typically modifies the list of processes displayed to users to exclude itself from the list, it may remove the names of its own files from file lists, and so on. In order to do this effectively it has to operate at a fairly high level; if it knows what information has been requested, it is easier to remove traces of itself.

Given these facts about rootkits, the reliability of digital evidence retrieved from a low level may be better than that retrieved at a higher level. Obtaining information from a lower level not only bypasses rootkits that have been installed on a higher layer, but also shortens the chain of services that are used to answer a given query. If fewer services are involved, the probability that one of them has been modified is lower than in the case of a longer chain of services. However, if data is retrieved from a lower level, it is necessary to reconstruct the

higher-level information structures ideally using code that is known to be reliable.

This chapter describes the development of a prototype that accesses information from a disk on low level during a live analysis. Its own code is then used to reconstruct the logical files that exist on top of the low level information on the disk.

Apart from demonstrating that a tool such as this can be constructed, and arguing the merits of the approach, there is a third goal of this chapter: the prototype tool was built using the platform that was developed and documented in previous chapters of this dissertation. This platform is intended to allow fast prototyping of forensic tools, including tools that emerge from academic research and once-off tools needed for specific investigations.

The platform ensures that as much code as possible is reused, thereby increasing the reliability of evidence collected and its potential admissibility in the case of legal proceedings. The platform also potentially enables the investigator to utilize other tools built using the platform, thereby increasing the range of collection possibilities and analysis that may be performed based on collected data. As the first tool built on this platform the prototype is also intended to examine the ease with which such a new tool may be constructed on the platform.

The remainder of this chapter is structured as follows. Section 8.2 elaborates on the practice of live evidence acquisition. Section 8.3, 8.3.1 and 8.3.2 describe the implementation of the prototype as well as the results that were obtained. Section 8.4 discusses the conclusion that was reached after this prototype had been developed.

8.2 Live evidence acquisition requirements

Digital evidence needs to be collected in a manner that ensures that the accidental modification of forensic evidence does not occur. Walker [99] cautions that a single file timestamp found to be later than the date of acquisition may cause digital evidence to be declared inadmissible in court. It is therefore extremely important to ensure that the state of the source of acquisition remains intact when an acquisition is performed.

Any file accessed on a logical partition, which is mounted in standard read/write mode, may have its associated attributes (such as access time) modified by the operating system when an action is performed. Although the ability of the operating system to update the access time of files as they are accessed may be useful for system administrators, it is less desirable when digital evidence acquisitions are performed. The use of standard file access routines, supplied by the operating system, should therefore be avoided when acquiring evidence to ensure that the state of the evidence in question is not changed accidentally.

In addition, according to Casey [22], the use of standard operating system copy routines should be avoided when live acquisitions are performed due to the fact that root kits may be present on the system in question. Live

acquisition software should therefore have the capability to perform low-level file access without the help of the operating system. Casey [22] also states that files should be accessed in read-only mode to preserve the integrity of the files and the metadata describing them. The reasoning behind this is that any file system mounted in read/write mode will implicitly modify the file system state when a file access is performed, thus compromising the investigation.

Another, more technical, requirement for an acquisition tool is the need for the static compilation and storage of binaries that will be used to perform acquisitions. According to Adelstein [3] an investigator should never trust any binaries stored on the system in question; the investigator should rather use statically compiled binaries that do not require the use of external libraries. These binaries should be stored on CD-ROM to ensure that they cannot be altered.

8.3 Development of the prototype

This section considers the development of the prototype intended to be used during a live analysis, based on the Reco platform. The prototype will rely on the Reco platform's physical layer to supply access to the file system images as well as the interpretation layer to supply file-access routines that may be used to access files in read-only mode (see figure 21).

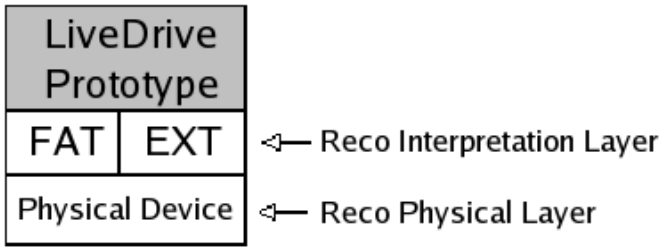


Figure 21: The prototype's dependence on the Reco Platform

At the time of writing, the Reco platform supports the following file systems: FAT12, FAT16, FAT 32, EXT2 and EXT3 which were sufficient for the development of this prototype. The prototype was developed in a manner that will allow the source code to compile and run on both platforms. The following two subsections discuss the sections of code that are platform-specific to both Linux and Windows. Although the lower-level implementation details may be different on each platform, the overall higher-level algorithm remains the same.

8.3.1 The Linux-based prototype

A device that may contain a file system in Linux is known as a “block device” [72]. These block devices may be opened just like any other file in the Linux environment, with the exception that administrative privileges are required. Thus it is possible to open the block device and read information from it; this process is similar to reading information from an acquired hard drive image. The

process of accessing a file located on a mounted logical partition is now discussed in more detail.

The partition on which a file of interest is located needs to be identified and opened in read-only mode. The platform should then be instructed to use the opened file as the target of analysis. The easiest way to determine which block device represents the logical partition in question is by inspecting the content of the /etc/mtab [8] file. This file contains information about mounted partition types, their mount points, as well as the location of the block devices containing the partitions.

Unfortunately this technique has some disadvantages: administrator (root) privileges are required when a device is opened as a file and access to the /etc/mtab file changes the state of the file system. A way in which to access the /etc/mtab file without altering its access time would be to firstly open the device on which the file is located with the Reco platform, access the file in read-only mode, and then close the device again. This method allows access to the file without compromising the integrity of the file system, but prior knowledge of the block device that maps to the mounted root partition would be required for this technique to work.

8.3.2 The Windows-based prototype

A similar method is now discussed for the Windows environment. A logical device is opened as a file and the Reco platform is then instructed to mount the open file as the forensics target in question.

According to Microsoft [70], a logical device can be opened as a file using the Windows CreateFile() API call with the filename ".N" where N denotes the drive letter representing the logical partition in question. Unfortunately administrator privileges are needed to perform this operation. The next question is how to determine which logical drives are mounted.

According to Microsoft [71], the GetLogicalDrives() API call can be used to provide this piece of information. The call returns a bitmap representing the drive letters of the mounted logical partitions. Using this information in combination with the CreateFile() method, it should be possible to acquire access to a logical partition in the same way as discussed in the previous section.

The partition containing the file of interest is located and opened. The Reco platform is then instructed to use the opened file as the source of analysis, after which the files stored on the partition are accessible to applications using the framework.

8.4 Implementation

A prototype was developed in C++ that uses the methods discussed above. A graphical user-interface was developed using the cross-platform wxWidgets [109] framework. The prototype was developed in very little time (about two days worth of programming), and when considered apart from the Reco framework, contains relatively few lines of code.

As noted, the prototype source files were designed to compile in both Windows and Linux without requiring any special makefile or changes to the project source code. Platform-specific sections of code were marked for compilation using pre-processor flags specific to the operating system in question. A combination of the two approaches allowed for the development of code that works in both Windows and Linux, without any major compatibility issues.

8.5 Results

The system was tested on Windows XP and Linux Fedora Core 4. Executables were generated that statically linked to the Reco platform library.

The results obtained on both operating systems were similar: regular files could be accessed without modifying the files themselves or metadata describing those files. A mounted logical partition could be opened by the prototype, the directories in the partition in question could be browsed and files could be copied to another partition to allow forensic examiners to inspect their content. Figure 22 shows a screenshot of the prototype in action.

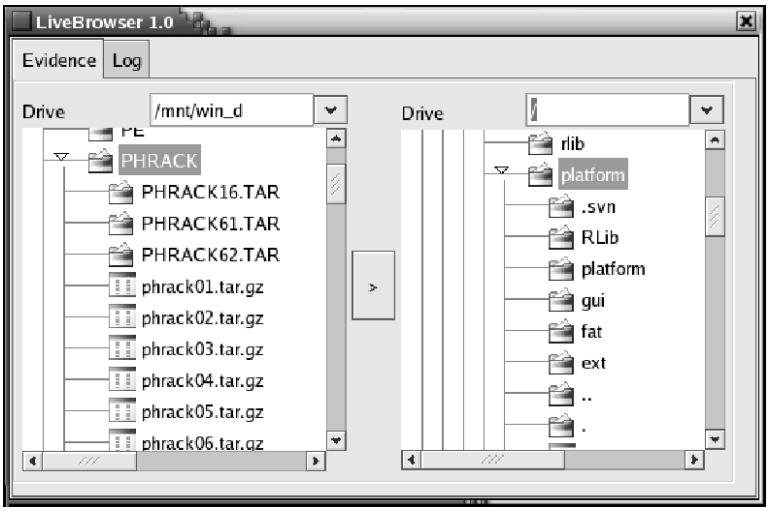


Figure 22: Screenshot of the prototype.

An access-time comparison was conducted to illustrate the difference in access time currently experienced by the file system drivers used by the prototype. An application was developed that created images of different sizes on the logical partitions targeted by the file system drivers. The created files were then read and the time it took to read each consecutive file was captured for each distinct logical partition. Figure 23 illustrates the results obtained.

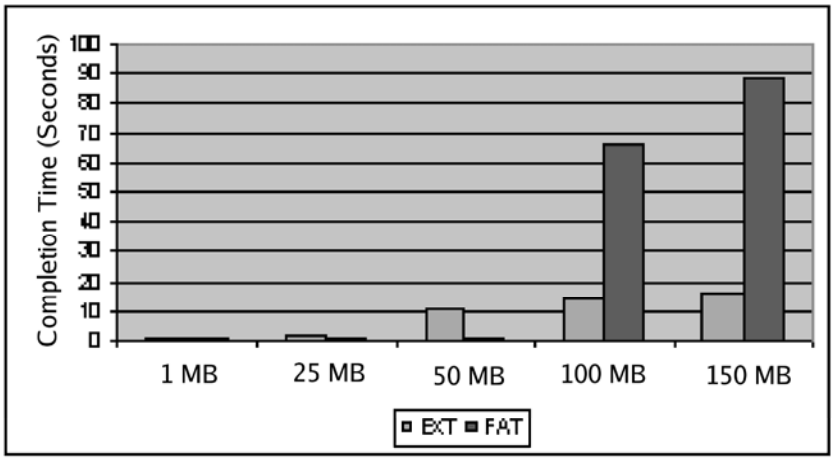


Figure 23: A comparison between the Reco file system drivers.

The graph indicates the efficiency of the chosen driver implementations used by the Reco platform. The EXT file system driver showed signs of linear increase in access time as the size of the file in question increased. This result was expected due to the fact that more work is performed when more data is accessed.

The FAT driver yielded less desirable results. Instead of indicating a linear increase as expected, signs of exponential increase were evident with an increase in the amount of data accessed. This is unfortunate as it indicates that the Reco platform is not currently able to supply fast access to large files stored in a FAT partition.

Because the higher-level operating system layers were bypassed when the acquisitions were performed, it can be assumed that the results obtained would be immune to most rootkits in circulation. Although most rootkits may be bypassed using the method described in this chapter, it is by no means a comprehensive way to neutralize the effect of every type of rootkit that may exist. This is due largely to the limited involvement that the operating system has in the control of the logical devices connected to the system.

When access is required to a logical device, the prototype sends a request to the underlying operating system that asks permission to open a logical drive as a file. When data needs to be read from the logical drive, a read request is sent to the operating system to perform the task. A sophisticated kernel-level rootkit which contains the same file processing capabilities as the Reco platform, could in theory return carefully crafted blocks of code that were maliciously engineered to hide traces of data or inject falsified information. Although this type of rootkit would be rare due to its immense complexity, it would still be possible for the average malicious programmer to develop such a system by using enabling tools like the Reco platform and others like it.

8.6 Conclusion

This chapter has described work done against the backdrop of the two types of digital forensic evidence acquisition techniques, namely live analysis and dead analysis techniques. Dead acquisition occurs when the system in question is taken offline before an acquisition is performed. Live acquisition occurs when a system is kept online while a digital acquisition is performed. Because the reliability of results obtained using the live acquisition method is lower than that obtained using dead acquisition, live acquisition should only be used when dead acquisition is impossible due to influencing factors such as loss of income or endangering lives if the system should be taken offline.

The Reco platform was used to develop a prototype for both Windows and Linux to illustrate the usefulness of the platform in prototyping scenarios. With only a few lines of code, the researcher was able to construct a system that had the capability to access files stored on FAT12/16/36 and EXT2/3 partitions, without modifying the files in question or the metadata describing those files.

The method used to supply access to files stored in a logical partition may be criticized due to the fact that administrator privileges are required by the user of the prototype. Although this is a valid issue, it should be considered in a live acquisition context. Another valid criticism is that the method used to obtain access to the file system information is by no means absolute; the low-level data access mechanisms may still be bypassed by sophisticated kernel-level rootkits, which makes the use of the specified technique in live forensics less desirable.

The development of the prototype proved that the Reco platform could indeed help researchers to develop forensic prototypes in shorter amounts of time, while requiring a lesser degree of in-depth understanding of operating systems. What is remarkable is that the Reco platform proved to be extremely useful in the development of this prototype, even in the platform's early stages of development. If a project in its early stages of development could be useful to this extent, imagine the exciting capabilities that it would have at a later point in time when development has progressed to a point where it can compete with its commercial counterparts.

9 Using Timestamp Relationships as a Forensic Evidence Source

9.1 Introduction

Digital evidence is not well perceived by the human senses [101]. Crucial pieces of digital evidence may simply be missed due to the fact that examiners do not fully comprehend how seemingly useless pieces of data can be converted to evidence of high value. This situation may be very problematic for digital investigators as it may create an incomplete picture of digital crimes under inspection [21]. It is therefore extremely important to examine all evidence, no matter how insignificant it may seem.

If an investigation team can understand an intruder's modus operandi, it may be possible to determine various attributes describing the intruder, such as skill level, knowledge and location [18]. Security mechanisms such as log files are usually used to determine the actions of the intruder. Unfortunately it is possible that active security systems on the compromised system may be configured incorrectly or disabled completely [92]. In such circumstances investigators will have to turn to alternative sources of digital evidence.

File timestamps may serve as a worthy alternative, as timestamp information may be viewed as a simplistic log of events as they occurred. Although file timestamp information may be considered one-dimensional in the sense that it only records the time of the very last action that was performed on a file, it may still be a valuable source of evidence when few alternatives remain. Unfortunately the processing of file timestamp information may be complicated by the sheer volume of available timestamps that need to be processed.

The overabundance of digital evidence that needs to be processed in small amounts of time could be described as an audit reduction problem [28]. The audit reduction problem describes the situation in which the presence of too much information obscures the focus point of investigations. Audit reduction is prevalent in digital evidence analysis due to the masses of files that need to be inspected, spurred on by increasingly massive storage capacities of modern storage devices.

File timestamps analysis is an excellent example of the audit reduction problem: modern hard drives have a storage capacity that may be anywhere in the range of gigabytes to terabytes; a very large number of files may be found on these devices, with each file having different timestamp information associated with it. Although most of the file timestamps would be irrelevant to a case, a few may be the key to its successful resolution. If these timestamps are simply overlooked, an incorrect conclusion could potentially be reached which may have dire consequences for the accused as well as for the investigation team.

This chapter discusses the use of timestamps as a supplement or alternative to log files, when log files are not available. The information

deficiency problem is discussed, which describes the situation in which not enough information is available to allow investigators to get a clear picture of significant forensic events. This highlights possible problems that may be experienced with alternative evidence sources.

The concept of synergy applied to digital data is proposed as a solution to the information deficiency problem. This principle should allow investigators to use various insignificant evidence sources to generate abstract forms of information that is considered to be of forensic value.

The chapter is structured as follows. Section 9.2 discusses the importance of file timestamps. Section 9.3 focuses on file timestamps related to incident phases. Section 9.4 introduces the information deficiency problem and section 9.5 discusses a possible solution to the problem. Section 9.6 discusses the development of a prototype using the Reco platform, section 9.7 presents the results obtained and section 9.8 discusses the prototype flaws. The chapter closes with section 9.10 which presents a conclusion.

9.2 File timestamps as a source of evidence

Attackers may try to delete or alter log files in an attempt to cover their tracks; fortunately pieces of information may remain due to a lack of skills or access rights [92]. As an example, consider the use of well-known UNIX commands such as cat and grep. An attacker may use these two commands to remove identifying information from a system log file. A clever attacker may even change the log file's modification date after the alteration, so as not to arouse any suspicion from the system administrator. With the system log files compromised, investigators will have to find an alternative source of evidence as compromised evidence sources may not be credible in a court of law.

Fortunately there exists a less obvious source of digital evidence - file timestamps. Consider the example just mentioned: the attacker used a combination of well-known tools such as the cat and grep commands to remove identifying information from the system log file. Very few attackers would actually reset the file access timestamp that was created when the shell command was executed. Even if they did manage to modify the file access times, they would have used a tool to do so. This means that although the commands used by the attacker do not have valid timestamps associated with them, a valid timestamp would be left somewhere on the system by the attacker, unless the command was executed from a read-only medium.

From this discussion it should be obvious that only extremely skilled attackers would be able to access a system without leaving a single trace; less skilled attackers are bound to leave small pieces of evidence behind that may ultimately be used to identify them.

Popular file systems such as FAT, NTFS and EXT store file timestamps to keep a record of:

- The file creation time;

- Last time the file was accessed;
- The last time the file was modified.

These timestamps are updated by the underlying operating system when appropriate, but skilfully written applications also have the ability to manipulate timestamps as they require. Applications have different approaches concerning the management of timestamps. As an example, consider two well-known UNIX commands, namely cp and tar. When a file is copied using the cp command, the resulting creation and modification timestamps of the destination file indicate the time that the cp command was executed. This is not the case with the tar command. When a compressed archive is created, the relevant files, along with their timestamps, are stored in a compressed archive. It should therefore be noted that some applications will posses timestamp modification capabilities which may have a negative effect on the timestamp analysis process. This topic is discussed further in section 9.8.

9.3 Timestamps and incident stages

Three digital evidence stages were identified in chapter 5 which classify evidence according to its temporal relationship to a digital incident. The identified stages are as follows:

- Pre-incident;
- Incident;
- Post-incident.

Chapter 5 presents a complete discussion on this topic. The information supplied by timestamps is very limited in the sense that a timestamp only records the last time a specific activity took place. To simplify this discussion, it is assumed that a file has only a single timestamp associated with it. Although this is not the case in reality, the principle remains the same for all timestamp-based information.

The most accurate timestamp from an evidence timeline classification point of view is the timestamp recorded in the pre-incident stage, as a timestamp with a time earlier than the incident means that the file in question was used before the incident occurred. This means that the file may have executed an action on files involved with the incident, but it could only have done so up until the point that it was last loaded in memory.

Timestamps captured in the incident stage indicate that the files in question were used during the incident stage, but they could also have been used during the pre-incident stage. The situation gets worse in the post-incident stage: files with a timestamp in this stage may have had actions performed on them during any one of the phases. An information deficiency problem therefore exists with regard to timestamps and the incident stages, in particular the post-incident stage.

For analysis purposes it will have to be assumed that evidence had actions performed on it during every stage prior to its current incident stage. A solution to the information deficiency problem may be to introduce additional evidence sources in an attempt to build timelines that indicate upper and lower bound incident stages in which actions were performed on the object in question.

9.4 Applications and file timestamp relationships

In order for a timestamp to change, an action is required. The action will have to be triggered by an application or device driver resident in memory at the time of change. For this discussion it is assumed that three types of timestamps exist, namely the creation, modification and access timestamps, and that the operating system alone can modify file timestamp values. The value of the timestamp is not important in this discussion, as its meaning is largely dependent on the application that triggered the event.

What should be considered important is the fact that an executable code that triggered the event should have been active in physical memory prior to triggering the event. This means that the file in question should have been loaded into memory, thus modifying its file access timestamp. An executable that accesses or modifies a file should therefore have a file access timestamp which is earlier than the file in question's timestamp (create, modify and access timestamp depending on the action performed). The following function can be defined to determine if an application's create, access or modify time has been edited:

$$touched(f) = ceil\ (\ create(f), access(f), modify(f)\ )$$

Using the defined function, the following condition should therefore hold:

$$access(executable) = touched(file)$$

Unfortunately, due the information deficiency problem identified previously, a piece of executable code may be loaded again in the future which means that the stated condition will not hold anymore, as the access time of the executable code will have changed. The following situation may therefore exist:

$$access\ (executable) = touched(file)\ or\ access\ (executable) = touched(file)$$

This basically means that it would not be possible to pinpoint the application responsible for the modification of a file, as insufficient information exists. The executable access timestamp cannot be used to rule out the application associated with it, as the application may have been resident in memory for some time before the modification of a file's timestamps. It can

therefore be concluded that application/file timestamp relationships are of very little forensic significance on their own; some additional form of information is required to help rule out executables that could not have modified the file in question.

The executable access timestamp cannot be used to help rule out the application associated with it as the application may have been resident in memory for some time before it triggered the modification of a file's timestamps. If it was possible to prove that the application in question was removed from memory some time after its file access timestamp indicated, it may be enough to rule out the application as the trigger source.

As an example, consider the diagram illustrating the executable access timestamps in the different incident stages (see figure 24). Various evidence artefacts have been organized according to file creation timestamp dates. Executables 1, 2 and 3 could individually have created files A,B,C,D and E. It is therefore not possible to rule out any executables from the equation Thus application/file timestamp relationships are not of forensic significance on their own.

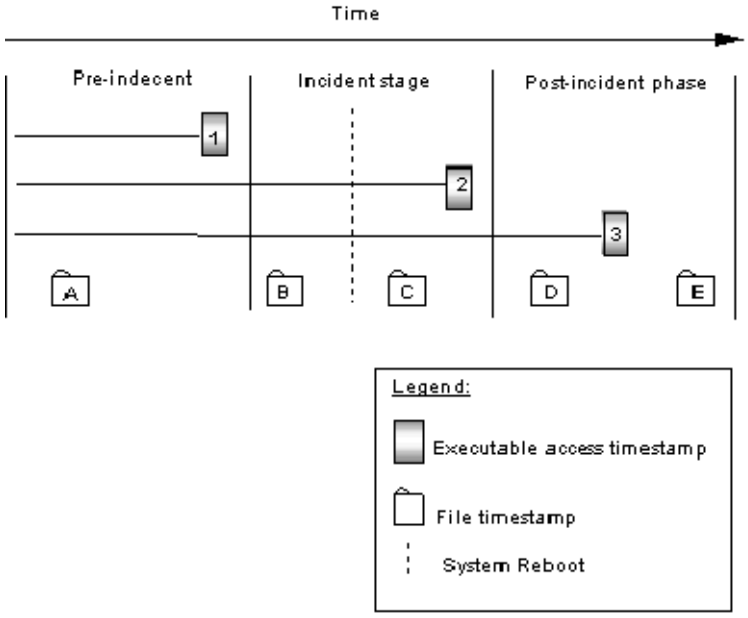


Figure 24: Files organized according to timestamp information

Imagine the intruder managed to reboot the system in question during the incident phase. Knowledge of this event may help to place an upper-bound on the last possible time that executable 1 could have had an effect on the file timestamps of the listed artefacts. File access information informs us that executable 1 was last executed during the pre-incident phase; a system log file (collaborating evidence) shows us that the system went offline during the incident phase. Executable 1 was not loaded again after the system went back

online after the reboot. It can therefore be concluded that executable 1 did not have an effect on the timestamps of the listed artefacts after the reboot.

With enough collaborative evidence at hand it may be possible to narrow down substantially the list of executables which may have been responsible for triggering an event that modified timestamp information. The above example relied on the knowledge that the system had been rebooted. Normally such information will be gathered from a system log file, but in the absence of credible log files, investigators may once again need to turn to file access timestamps as an indicator of system events.

When a system boots, various executables are loaded as services. These executables are usually only loaded once and stay loaded until a system halts or reboots. By looking at the access timestamps of these services it may actually be possible to determine when the system booted. This method is discussed further in section 9.6.

9.5 Solution to the information deficiency problem

Synergy describes the situation in which the whole is greater than the sum of the parts [105]. Although various events may be seen as insignificant on their own, their importance may increase when their collective importance is realized, that is when a state of synergy is achieved.

Consider again the example in figure 24: each of the events that caused changes in timestamp information associated with the files is of very little forensic value when considered on their own. Event the timestamp that indicated that the system in question performed a boot operation would seem relatively useless on its own, as it does not convey any useful information other than that a system boot took place. The real value in the timestamp information lies in the fact that it indicates that certain events took place. On a higher level, these events may be related to one another in order to create an abstract view of the events as they occurred.

The example in the previous section illustrates that it may be possible to extract useful information from data that is seemingly useless when viewed on its own. A file's access timestamp may have very little importance on its own; its importance is directly related to the importance of the event that it represents. The synergy principle that focuses on the creation of abstract evidence information from insignificant pieces of data, may therefore be formulated as follows:

Event data is generated when an significant digital event occurs. Although the generated event data is of little value when viewed independently, collectively event data can produce information that can help investigators to deduce relationships between events to produce abstract views of the evidence at hand.

Investigators usually have lots of complex questions to answer in a short amount of time [18]; the possibility therefore exists that evidence may be overlooked as investigators focus their attention on evidence that seems more important in an attempt to save valuable time. However, identifying the relationships that may exist between seemingly unimportant pieces of digital evidence may be an extremely tedious task to perform.

As Adelstein [63] points out, it is not feasible for investigators to manually analyse storage devices with storage capacities in excess of gigabytes as there is just too much data to process. Without some form of automated processing the benefit obtained as a result of time invested by investigators would be minimal due to the sheer volumes of data that needs to be processed.

9.6 Prototyping

A prototype was created based on application/timestamp relationships discussed previously, in an attempt to illustrate the defined principle in action. The prototype was developed to extract information from a Linux-based EXT2/3 file system storing ordinary files, applications and system logs. The prototype was built under the assumption that the file timestamps had not been tampered with. It was also assumed that the executable access time indicated the last time the application was loaded by the operating system. File creation timestamps were ignored as it is assumed that file access and modification times will always be later than a file's creation time.

Casey [21] proposed a certainty scale that may be used to determine the level of trust that can be placed in the information deduced by the investigators by examining the forensic evidence at hand. Evidence that appears highly questionable will have a low certainty level associated with it while evidence that can be correlated with other captured evidence sources will receive a higher certainty rating.

Casey's certainty scale can be used in addition to the synergy principle to increase the level of trust placed on extracted information; evidence which can be correlated with other sources of information may yield a higher degree of certainty. Relating Casey's work and the synergy principle to timestamp information, it can be assumed that timestamp information that is correlated with timestamp information from the same disk image will have a lesser degree of certainty than timestamp information that may be related to some other form of evidence, such as system logs. The prototype in this case was built with the purpose of identifying the last possible time that an application could have been loaded in memory, known as the “last possible execution time”.

This was done in an attempt to determine which files could have been modified by the application in question. The last possible execution time is determined in one of two ways: by correlating an application's access timestamp with system log entries, or by correlating an application's access timestamp with the access timestamps of system applications and/or files that are accessed on system boot or shutdown events.

The first method would obviously be the better choice for the correlation of evidence as it contains a rich source of system-related history information. To determine the last possible time an application could have been in memory is simple: use the application in question's access timestamp and search for the earliest system halt or reboot event that occurred thereafter in the log file. The time specified in the log for the halt or reboot event would therefore serve as the last possible execution time as the executable was not accessed again after that specific point in time.

The second method may serve as an alternative to log files in situations when it has become evident that the system log files have been tampered with or in environments where no log files exist. When an operating system boots or halts, it loads various system applications and accesses stored settings, thus changing their accessed timestamps. The timestamps viewed on their own are insignificant, but when used to determine when a system was turned on or off, they may be of great value to forensic investigators.

As an example, consider the sequence of events that occurs when a standard Linux system boots. The first process created by the kernel executes the /sbin/init application. When the /sbin/init application starts, it reads the /etc/inittab file for further instructions. By simply checking the accessed timestamps of either one of the two files, it would be possible to determine the last time that a system booted. It can be argued that the information is also obtainable from alternative sources (such as the /proc/uptime file), but in situations where the alternative is damaged or simply does not exist, timestamps will have to suffice.

Calculating the last possible execution time by means of the second method is similar to that used in the first method: determine an application's accessed timestamp information and determine the last time a system booted or halted by looking at the applications and files associated with the system boot or halt operations.

The prototype described in this chapter reads disk images to produce XML files containing timestamp information. These XML files are then converted to scatter charts to improve the way timestamp information is perceived by the human senses. The prototype depends on the Reco Platform, developed as part of this dissertation, as well as the well-known library, JFreeChart [78]. The design is illustrated in figure 25.

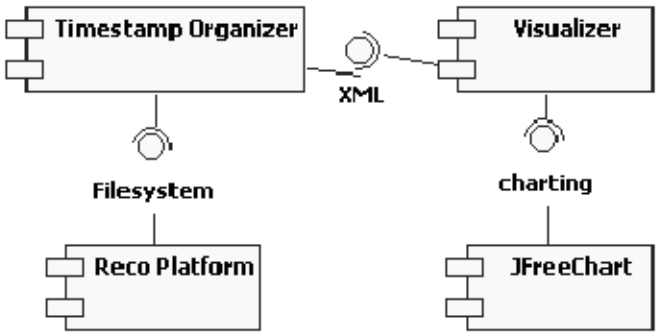


Figure 25: The prototype design

The Reco Platform supplies low-level EXT2/3 support to the system while the JFreechart library supplies the graphing functionality required by the application. The prototype source code has been released under the GNU GPL license and is available on Sourceforge [56]. The next section discusses in more detail the results that were obtained using the developed prototype.

9.7 Results

The prototype was tested using Linux (Fedora Core 4). A disk image was made and last possible execution times were computed for each application using both methods described previously to produce separate XML files. A scatter chart was constructed using each detected file's modification and access times as coordinate values.

The selected application's last possible execution time was plotted as horizontal and vertical lines to indicate the reach (in terms of what the application could have modified) of the application in question. Figure 26 illustrates the scatter chart as well as the horizontal and vertical lines indicating the maximum reach of the application in question.

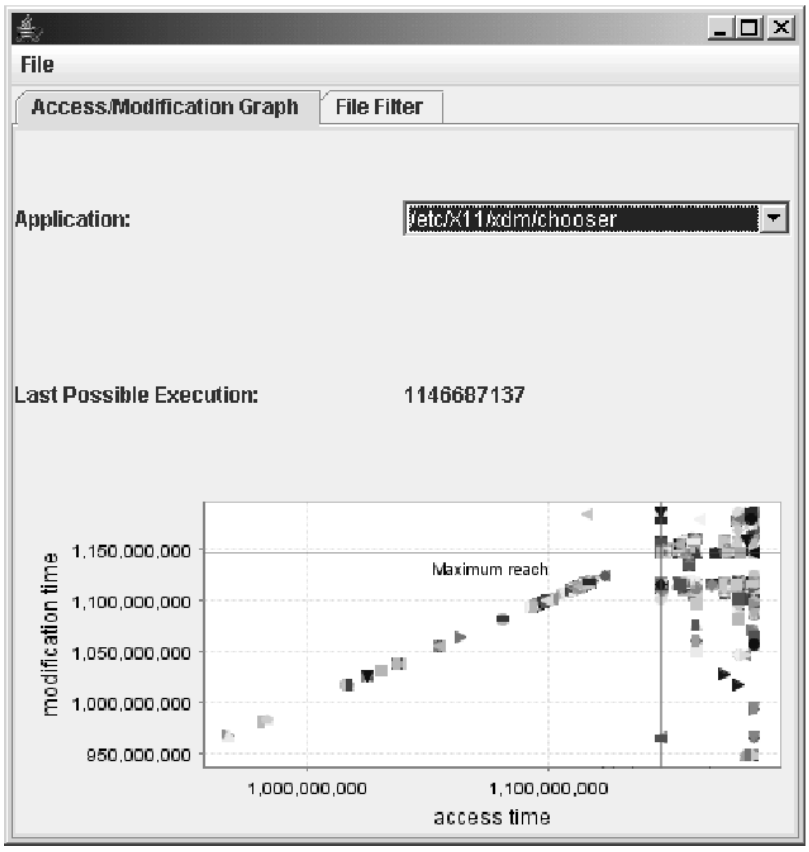


Figure 26: A screenshot of the prototype

The user is able to select an application of interest from a dropdown control populated with a list of applications. Upon selection, the application's last possible execution time is computed and plotted on the scatter chart. The last possible execution time, access time and modification times are represented by integer values; the integer value is a timestamp that describes the number of seconds that have elapsed since January 1,1970 (which means that the values could easily be manipulated using a function such as ctime) when the event in question occurred.

In the example (figure 26) the last possible execution time for the application */etc/X11/xdm/chooser* was calculated to be 1146687137 seconds since 1 January 1970. Translated to human-understandable terms, the last possible execution time for the application in question is Tuesday, May 2, 2006 at 23:58:57. The application cannot be responsible for any file access or modification operations performed after the last possible execution time, represented by the horizontal and vertical lines on the graph. Any files outside of the horizontal and vertical lines will therefore have been accessed or modified by other applications.

By simply looking at the chart generated, it is possible to visually detect which files could have been modified by the application in question. Due to the

sheer magnitude of the number of files that are stored on a disk drive, a file filter functionality has been added to the prototype to search for files with timestamps conforming to specific criteria. Determining the names of the files that could have been modified by the application in question was as simple as submitting a filter query that contained the last possible execution time of the application in question.

A comparison between the two techniques used to determine an application's last possible execution time yielded the results that were expected: since system log files contain detailed history information, more accurate last possible execution times could be calculated, thus leading to more accurate results. File access timestamps contain only the last time the file was accessed and can therefore be compared to a log file containing entries which date back to the last time a system in question was booted. This implies that the second method could work with the same efficiency as the first method in a scenario where a system rarely goes offline. However, the second method would be very inaccurate for systems that go offline frequently.

9.8 Criticisms

As discussed in section 9.2, some applications have the ability to modify timestamps. The work in this chapter assumed that timestamps are modified by the operating system only, and did not take into account that some applications may manipulate the proposed analysis method by changing file timestamps to render the method invalid. In reality, the interpreted meaning of a timestamp is therefore largely dependent on the way in which the application responsible for the creation or modification of a file manages timestamp information.

It was also assumed that the application will be stored on a writeable medium; its timestamp information will therefore be updated each time the application is loaded into memory. This may not necessarily be the case as it is possible in the UNIX environment to mount file systems in read-only mode. This means that an application's file access time will not change, thus rendering the method described in this chapter useless.

Another concern is that an application may have accessed or modified a suspicious file prior to its own last possible time of execution; if the suspicious file was accessed or modified again some time later (presumably after the application in question's last possible time of execution), the timestamp may be labelled as being out of reach of the application in question. Technically this is true as the file was last modified by another application, but this situation may not always be desirable.

A way to overcome this problem is to divide application timestamps into the various incident stages discussed in section 9.3. Only applications with access timestamps falling in the incident and post-incident phase will have to be considered for inspection, as it can be assumed that applications with last possible execution times falling in the pre-incident stage were not involved with the incident in question.

9.9 Future Work

A complex application would typically touch various files while it is executing. A typical scenario would be where the application in question first accesses its configuration files and then data files. By describing an application's actions formally, it may be possible to create a profile that accurately describes an application's file access characteristics.

Another topic that requires attention is the inspection of the file access of an operating system's boot process. When an operating system performs the boot process, various files are accessed. Different operating systems access different files, which creates the possibility that the file access operations performed by an operating system could potentially be used as a fingerprint to help operating system identification in circumstances in which conventional methods are not deemed appropriate.

The prototype could potentially be improved by adding the concept of a termination signature. The termination signature describes the characteristics of an application when it terminates, in other words what actions it takes just before it terminates. If such a signature can be incorporated into the concepts described in this chapter, more accurate results may be obtained. Future studies may be devoted to the identification and definition of such signatures.

9.10 Conclusion

This chapter discussed how timestamps could be used to rule out files that could not have been modified by distinct applications, based on an application's calculated last possible execution time. The synergy principle was introduced, which claims that insignificant pieces of event data may collectively be of significant forensic importance.

A prototype was constructed based on this principle, using timestamps as a source of insignificant evidence. The prototype calculated the last possible execution times of various applications and depicted this information visually in a manner that can easily be understood by the investigator. It has become evident that a great need exists for ways in which digital evidence can be visualized in order to allow investigators to easily understand it. The prototype helped to visualize abstract digital data which is not well perceived by the human senses, to help investigators to easily understand the produced data, as well as its importance.

Unfortunately the method used by the prototype is not absolute in the sense that it cannot successfully be applied to all environments under all conditions. The technique can therefore not be considered as an absolute solution to the problem discussed in this chapter, due to various factors that may have a negative impact on the accuracy of the produced results. Nevertheless, it would still be possible to use the described technique in certain environments in which few sources of digital evidence are present.

10 Development of an NTFS File System Driver

10.1 Introduction

The developed forensic platform was designed to allow the use of existing open-source file system drivers in an attempt to decrease the amount of effort required to add support for new file systems. To verify that the use of open-source drivers are acceptable, a custom file system driver will be developed in an attempt to determine if it would be possible to get higher quality information using the developed driver code. If the custom driver produces information which is of higher quality, it can be argued that the use of open-source drivers within the platform is not recommended as the quality of the information produced by the open-source drivers are of low quality. However, if the quality of the information produced by the custom driver is equal or less than the quality of the information produced by the open-source drivers, the use of open-source drivers within the platform would be acceptable.

The ideal approach would be to perform a comparison between an open-source driver and a custom driver that both add support for the same file system. This method would allow the comparison of two items that are exactly alike. Unfortunately such an approach may be considered a waste of resources due to the fact that existing functionality would be re-implemented in an attempt to determine if the re-implementation can add additional value to results obtained through its use.

It was therefore decided to add support for a file system that is not supported by the platform in an attempt to determine whether or not the results obtained through the development of custom drivers could be of higher value than the results obtained through the use of existing open-source drivers.

This chapter is devoted to the development of an NTFS file system driver for the forensic platform. Concepts surrounding the NTFS file system, such as the master file table and alternate data stream are discussed to allow the reader to understand the inner workings of the NTFS file system. Once a good foundation has been established, the development process that was followed to produce an NTFS driver for the platform is discussed in detail.

The rest of this chapter is structured as follows. Section 10.2 discusses some basic NTFS background information, such as NTFS features, logical cluster numbers and virtual cluster numbers. Section 10.3 discusses the theoretical limitations of the NTFS to allow the reader to understand its capabilities. Section 10.4 conducts a technical discussion on the inner workings of the NTFS file system. The work discussed is then applied in section 10.5 which describes the construction of a file system driver capable of reading NTFS volumes. Section 10.6 discusses the results obtained and section 10.7 concludes this chapter.

10.2 NTFS background information

NTFS was developed by Microsoft as an alternative to the widely used FAT file system. This was due mainly to the fact that the FAT file system was not suited to the needs of business and corporate users [57]. An analysis of the needs of the business and corporate users resulted in the development of NTFS.

Some of the most notable NTFS features include [57, 73]:

- Better reliability thought the use of transactions;
- Built-in security, access control and disk quotas;
- Support for very large partition sizes;
- More efficient storage of files by NTFS compared to the method used by FAT;
- Support for long file names are (up to 255 characters) while maintaining support for legacy DOS 8.3 filenames;
- Built-in compression and encryption support;
- Distributed tracking of OLE objects.

To understand the rest of this chapter, two key concepts need to be understood by the reader, namely those of logical cluster numbers and virtual cluster numbers. A cluster can be seen as the basic storage unit in the NTFS file system [88, 102]. Each cluster consists of a fixed amount of sectors and each sector consists of a fixed amount of bytes. Since a cluster may be seen as a basic storage unit, it can be assumed that every piece of data stored in NTFS will be stored in one or more clusters. Each cluster in a NTFS volume is numbered sequentially, starting from 0. The sequence number assigned to a cluster is known as its logical cluster number or LCN.

A stream may be viewed as a collection of clusters addressed through their consecutive logical cluster numbers. Each cluster stored in a non-resident data stream is given a sequence number [88]. The assigned sequence number always starts at 0 for each stream and is referred to as a virtual cluster number or VCN. To access a cluster stored in a non-resident stream (using a VCN), the VCN has to be converted to an LCN. This is done through the help of data runs. The concept of data runs is beyond the scope of this chapter, but may be investigated by consulting Russon and Fledel [88] for an in-depth explanation. Figure 27 illustrates the relationship between disk offset, logical cluster numbers and virtual cluster numbers.

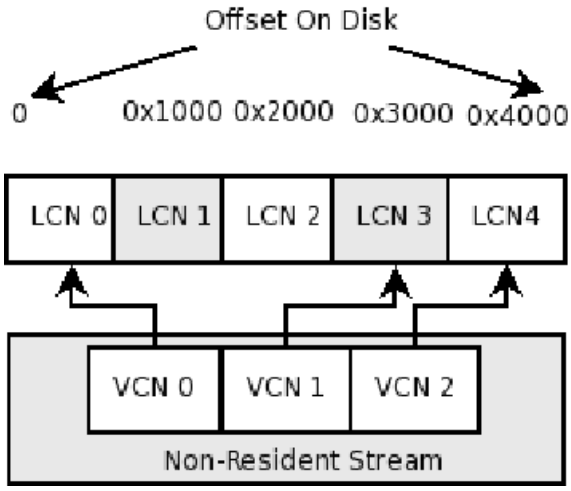


Figure 27: The relationship between disk offset, LCN's and VCN's

The figure can be summarized as follows: a cluster consists of a collection of bytes; each cluster is given a unique sequence number, known as the logical cluster number (LCN). Each non-resident stream contains a sequence of clusters numbered with virtual cluster numbers (VCN); each virtual cluster number refers to a logical cluster.

The background information to assist the reader to understand the basics of NTFS file systems has now been discussed. The rest of this chapter focuses on the development aspects surrounding the creation of an NTFS file system driver that can be used by the platform.

10.3 NTFS limitations

The theoretical size limit of an NTFS volume is clusters [73]. The theoretical limitation seems adequate for the current data storage requirement experienced by corporations. Unfortunately there exists a difference between practice and theory: the theoretical NTFS volume size depends largely on the implementation at hand. On Windows XP the maximum number of supported clusters per volume is a total of clusters [73].

It should therefore be noted that an NTFS volume can in theory be larger than the size limitation imposed by some implementations. The size of an NTFS volume is therefore dependent on the limitations imposed by the implementation as well as the chosen cluster size used to represent an elementary storage unit. A larger cluster size allows for the storage of more data.

NTFS supplies compression support through the use of the LZ77 algorithm [88]. The compression support is supplied by an NTFS driver implementation and should be transparent to the applications accessing the NTFS volume. This would allow applications to access compressed files with having to know that any of the accessed files are actually compressed. Compression is supported on NTFS volumes in which the cluster was set up to a maximum size of 4096 bytes [73]. Any volumes with a clusters size larger than 4096 bytes would therefore not be able to use the compression support provided by NTFS.

NTFS is a relatively advanced file system format that can be seen as a replacement for its predecessors, namely FAT16 and FAT32. As mentioned previously, a quick comparison between the three file systems was constructed to illustrate the differences that exist among them. Table 7 illustrates the differences between the various file system formats. By inspecting the table it should be clear that the capabilities of the NTFS file system far outweigh the capabilities of its FAT16 and FAT32 counterparts.

Table 7: A comparison between FAT16,FAT32 and NTFS

	FAT16	**FAT32**	**NTFS**
Maximum File Size	4GB – 1 byte	4GB – 1 byte	Theory: 16 exabites – 1KB Implementation: 16 terabytes - 64KB
Maximum Volume Size	4GB	32GB due to implementation.	Theory: 2^64 clusters – 1 cluster. Implementation: 2^32 – 1 cluster.
Files Per Volume	2^16	4,177,920	2^32 – 1 files
Maximum Files And Subfolders Within A Folder	512	65534	N/A

10.3.1 NTFS files and meta data

Everything in a NTFS volume can be seen as a file [88, 102]. From a user's perspective a file contains data whilst metadata contains data about data. Although a slight difference exists between file data and metadata outside of the NTFS context, both are treated the same by NTFS as they are both stored in exactly the same manner on disk [88].

From a software engineering perspective this simplifies the design of a read/write mechanism to access information stored on the file system, as the method of retrieval and modification would be the same for every item stored on the NTFS volume. Thus only one data retrieval mechanism is required to fetch file data and metadata. From this point on it is assumed that a file in the context of NTFS refers to any object (data or metadata) that is stored in an NTFS volume.

Every file in an NTFS volume is listed in a file called the master file table, or MFT [57, 73, 88]. The MFT may be seen as a structure similar to a relational database table due to the fact that it consists of rows of file records and columns of file attributes. Each file stored on the NTFS volume is listed in the database, even the MFT itself.

Due to the fact that every single file is listed in the MFT, it can be assumed that the MFT will consume a large amount of disk space. For performance reasons it would be a good idea to keep sections of the MFT as close together as possible to decrease the amount of time needed to access it. To achieve this effect the concept of MFT zones was introduced [88].

MFT zones help to decrease the fragmentation of the MFT by reserving a certain percentage of the disk space which may be used by the MFT. There are four different possibilities in regards to zone allocation, namely: 12.5%, 25%, 37.5% or 50% of disk space may be reserved for MFT purposes. The reserved space may only be used by non-MFT entries if there is no more disk space left in any other location except the reserved zones.

Meta data files

This section describes the concept of NTFS metadata files. Metadata is important from an NTFS perspective as it is used to describe the data stored by the file system. Metadata is stored along with data files in the master file table. The first 16 entries of the MFT are reserved for system metadata storage purposes [57, 73, 88] and cannot be used for any other designated purpose.

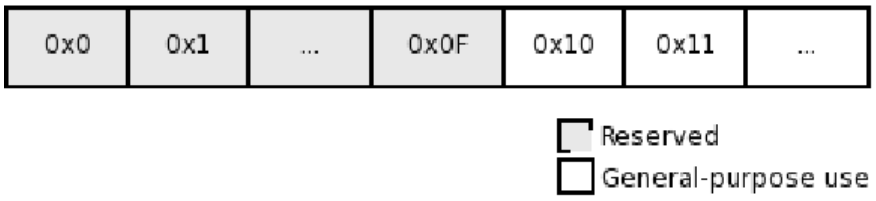


Figure 28: An illustration of the master file table

Figure 28 illustrates the current design with regard to the master file table: MFT entries 0 to 15 are reserved for system purposes, while any entry from position 16 onwards may be used to store general purpose files. Table 8 provides a summary of the descriptions for of the reserved files indicated in figure 28.

Table 8: A summary of the purpose of the reserved MFT entries

MFT Record:	File Name:	Purpose:
0	$MFT	Contains a record for each file stored in the NTFS volume.
1	$MFTMirror	A duplicate of the first four $MFT entries. Guarantees access to the disk when a single-sector failure occurs.
2	$LogFile	Contains a list of transaction steps that may be used by NTFS to recover from system failure.
3	$Volume	Contains standard information about the volume such as the volume name, serial number and creation time.
4	$AttrDef	Contains a description of various types of file meta data used by the NTFS volume.
5	.	The root folder which can be used as a starting point for directory traversals.
6	$Bitmap	Contains a bitmap of free/unused clusters.
7	$Boot	Contains the BIOS parameter block (BPB) and bootstrap loader.
8	$BadClus	Contains a list of bad clusters on the volume to ensure the exclusion of clusters that may cause data loss.
9	$Secure	Contains security descriptors for files.
10	$UpCase	Contains uppercase Unicode characters which may be used to convert lowercase letters to their Unicode uppercase equivalent.
11	$Extend	Used to store information regarding the extension of the NTFS volume.
12-15		Reserved for future use.

File attributes

Each NTFS file consists of a collection of file attributes which are used to describe the characteristics of the file in question. Each file may have one or more attributes associated with it. Table 9 gives a summary on the various NTFS attributes that may be associated with a file.

Table 9: NTFS file attribute summary

Attribute Type:	Purpose:
Standard Information	Contains basic information about a file, such as creation time, access time and modification time. Each file should have a Standard Information attribute.
Attribute List	Contains the locations of all attribute records that do not fit in the MFT record.
File Name	Contains the file's name. POSIX, Windows and DOS names are supported. A file will have at least one File Name attribute.
Data	A data stream associated with the file. A file may have more than one data stream.
Object ID	The unique ID of the object. This is used mainly for distributed object tracking purposes to enable OLE in distributed environments.
Reparse Point	Used to mount a disk drive at a specific location.
Index Root	Used by directories and for other purposes that may require the use of indexes.
Index Allocation	Used to store large non-resident indexes.
Bitmap	Used to indicate the usage of clusters involved with the storage of large non-resident indexes.
Volume Information	Contains the volume version. Only the $Volume file makes use of this attribute.
Volume Name	Contains the NTFS volume label. Only used by the $Volume file.
Volume Version	Appeared in Windows NT but was not used. Doesn't appear in Windows 2000 or Windows XP. This attribute can therefore be viewed as being obsolete.
Secure Descriptor, Symbolic Link, EA Information	Used to implement the HPFS extended attributes used by the information

	subsystems of OS/2 and OS/2 clients for Windows NT.

10.4 Accessing NTFS files

The purpose of the NTFS driver development exercise was to create a driver that adds NTFS support to the Reco platform. In order for a file system driver to be used by the platform, it needs to supply some basic functionality including the following:

- Opening a file on a disk volume and supplying read-only access to the file;
- Determining the size of a file;
- Getting and setting the file pointer position of an open file;
- Getting a list of files and subdirectories located in a specific directory.

It can therefore be concluded that the NTFS file system driver will have to supply file handling as well as directory handling functionality. For the purpose of writing a bare-bone driver to access information from a NTFS volume, the following MFT file entries have to be used:

- $MFT
- $BOOT
- .

The $MFT file is used to perform a lookup on the files stored in a NFTS volume. Before the $MFT file can be used, it needs to be located on the disk, for which the help of the $BOOT file is the only file on a NTFS volume with a fixed position. The $BOOT file will always be found at the first sector of a NTFS volume [57].

Accessing the $BOOT file is therefore a relatively simple process: open the disk volume that contains an NTFS partition, seek the first sector on the disk and read the file located at that sector. Table 10 summarizes the information that needs to be extracted from $BOOT in order to determine the position of the $MFT. Russon and Fledel[88] may be consulted for an in-depth discussion of the $BOOT file structure.

Table 10: A summary of useful information supplied by the $BOOT file

Offset:	Data Type:	Description:
0x03	char[4]	A NTFS volume will always contain the magic value "NTFS".
0x0B	unsigned short	Bytes per sector.
0x0D	unsigned char	Sectors per cluster.
0x30	unsigned long long	LCN where the $MFT is located.

It should be noted that the size of the specified data types is based on the standard data type sizes used by 32-bit C/C++ compilers. (See appendix 12).

The first step towards locating the $MFT is to verify that the disk image in question makes use of the NTFS file system to store data. This can easily be done by verifying that the magic values stored at position 0x03 match the string “NTFS”, since all NTFS volumes should have this magic value.

The second step is to actually determine the position of the $MFT on the disk. The following calculation may be used to determine the physical offset of the $MFT on a disk:

*$MFT location = ((bytes per sector * sector per cluster) * LCN)*

Once the $MFT has been located, the exploration of the NTFS partition may commence.

10.4.1 Directory traversals

NTFS makes use of a B-tree structure to store directory-related information [57, 88]. A B-tree is a balanced data storage structure that is largely used in relational database systems as a way of structuring data [57]. Because B-trees are balanced according to a specific key value, they allow fast searching when key values are queried. Applied to the concept of file systems, a B-Tree may be built up of file nodes, with each node represented by its file name as the key. This arrangement allows the execution of extremely fast directory query results and is therefore suited for use in a high-performance file system such as NTFS.

The traversal of a directory usually starts from the directory index root located at position 5 in the $MFT [73, 88]. The first step in performing a directory traversal is therefore to read the file with all its associated attributes located at position 5 in the $MFT. Each file may have various file attributes associated with it; for directory traversal purposes applied to the Reco platform there are four attributes that are needed to perform the traversal and extract the relevant directory information. These attributes are:

- $STANDARD_INFORMATION
- $FILENAME
- $INDEX_ROOT
- $INDEX_ALLOCATION (although this is not present when a very small directory structure exists).

Figure 29 depicts the information that needs to be processed by an NTFS driver in order to be used by the Reco platform. Each of the illustrated attributes has important information associated with it, indicated by the arrows. Note that the figure is a simplification of the storage structures used in an actual NTFS system; the reader will be able to find the declaration of the actual storage structures in appendix 13, if required.

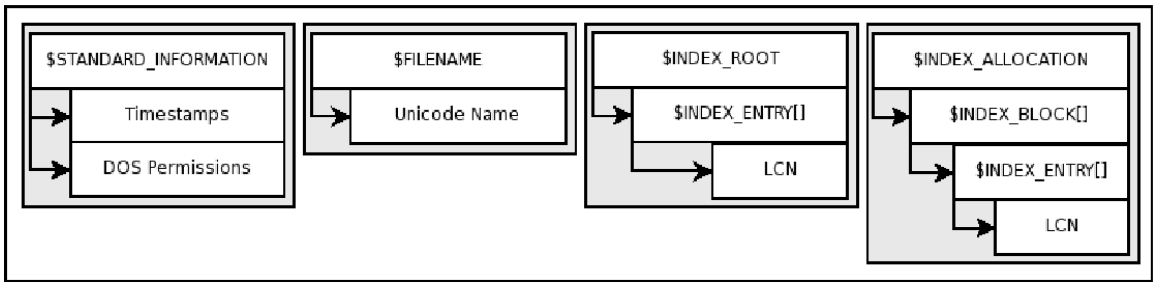


Figure 29: Information required by the NTFS driver

In order to determine which files are located in a certain directory, inspection of the $INDEX_ROOT attribute and the $INDEX_ALLOCATION attribute (if present) are required. The $INDEX_ROOT attribute contains information about files if the directory structure is small; larger directory structures are managed by a $INDEX_ALLOCATION attribute.

The $INDEX_ROOT attribute contains an array of $INDEX_ENTRY structures. The $INDEX_ENTRY structure may be seen as a leaf node of the B-tree containing a key and a stored value. The key may be the name of the file (although other key types are also allowed), while the stored value refers to the

LCN at which the file represented by the node may be found.

The $INDEX_ALLOCATION attribute also stores information about files contained in a directory; it differs from the $INDEX_ROOT attribute in the sense that it can store many more file entries and that an additional array is introduced for storage purposes. Each $INDEX_ALLOCATION attribute (which is always non-resident due to its size) contains an array of $INDEX_BLOCK structures. Each $INDEX_BLOCK structure points to an array of $INDEX_ENTRY objects. Each $INDEX_ENTRY object contains a key as well as an LCN which points to the file represented by the object.

To get a list of files located in a directory is therefore simple: process all the $INDEX_ENTRY entries in the array stored in the $INDEX_ROOT attribute. If the $INDEX_ALLOCATION object is present, all its $INDEX_ENTRIES should also be processed as well. The combination of the resulting file entries should supply a list of all the files listed in the directory in question. The name of the directory in question can be retrieved from the $FILENAME attribute and the relevant timestamp information from the $STANDARD_INFORMATION attribute.

10.4.2 Data streams

From an NTFS perspective, a data stream may be viewed as a sequence of bytes. Each NTFS file will have an unnamed data stream associated with it. NTFS has the capability to allow additional named data streams, known as alternate streams, to be associated with the unnamed data stream [73].

According to Broomfield [7], the NTFS alternate data stream functionality is poorly documented and public awareness of this feature is very low, even though it has been known to the security community [80, 110] for some time.

The following subsections discuss the way in which alternate data streams are used by Windows, as well as the manner in which they can be accessed.

Alternate Data Stream usage

The alternate data stream feature is used by Windows to store information concerning various documents [73]. To illustrate this functionality, a document was created in a format that does not natively support metadata. Metadata was then added to the document. The document source file was inspected in order to determine if the metadata was stored in the file in question.

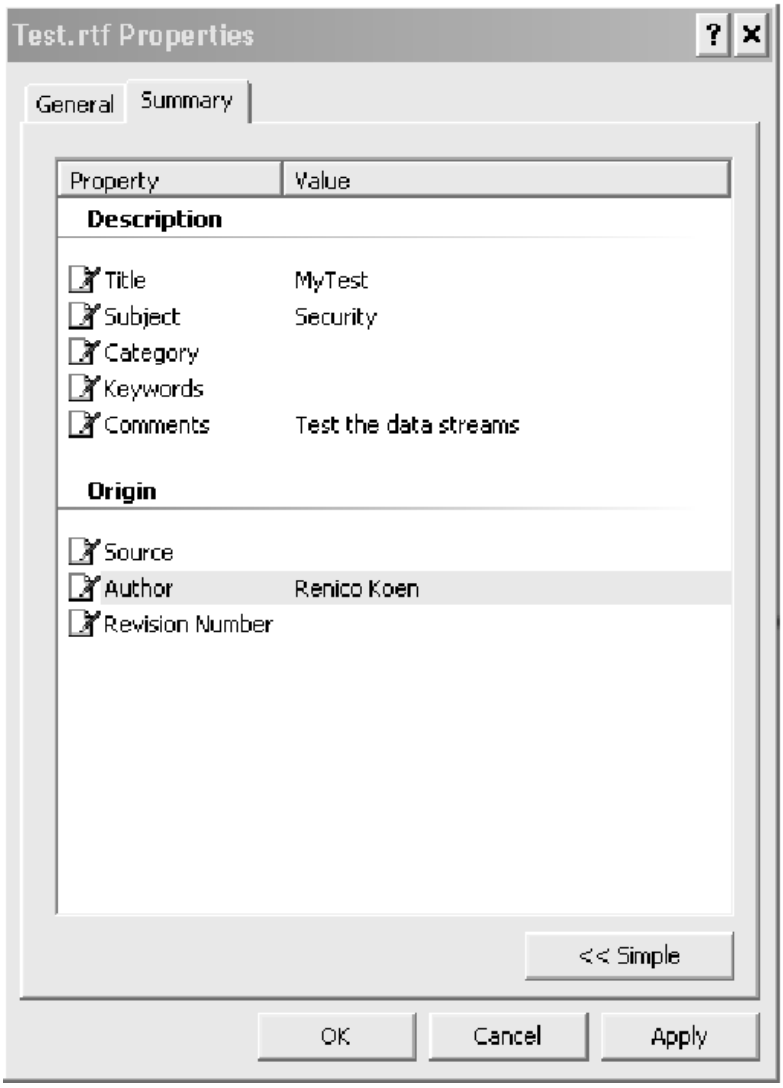


Figure 30: File Meta data that was used to describe the document

A file was created using Microsoft WordPad containing the string "Hello World". The file was saved in Rich Text Format (rtf). Summary information was supplied to the document by filling in the file summary information (which can be done by right-clicking of the document in question and selecting the Summary tab (see figure 30). The document was opened using a text editor after the summary information was entered, revealing the following code:

{\rtf1\ansi\ansicpg1252\deff0\deflang1033 {\fonttbl {\f0\fswiss\fcharset0 Arial; } } {\generator Msftedit 5.41.15.1507; }\viewkind4\uc1\pard\f0\fs20 Hello World\par }*

As can be seen from the code above, no summary information was

contained within the file. It can therefore be assumed that the summary information was stored somewhere else; most likely in an alternate data stream. The easiest way to verify that a file contains an alternate data stream is to copy the file to a file system that does not support the functionality. The operating system would thus be forced to throw away some information which would likely lead it to display a warning message.

The document file was copied from the NTFS partition to a FAT32 partition. The result was as expected: Windows displayed a warning message informing the user that the named data stream “Document Summary Info” would be lost if the copy process was to be executed (see figure 31).

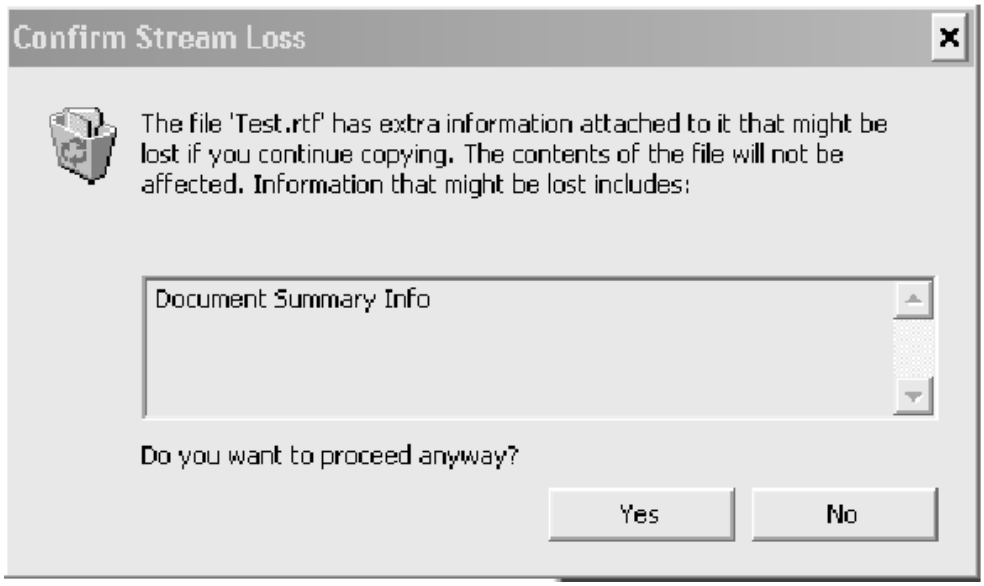


Figure 31: Warning presented by Windows due to possible data loss

It can therefore be concluded that the alternate data stream feature is in fact used by Windows to store file metadata in a non-obtrusive manner. This means that alternate data streams cannot simply be ignored as they are used by Windows for legitimate purposes.

Accessing the stream data

Applications may access the alternate data streams by referring to them by name. A data stream is represented by the $DATA attribute. The default data stream will be unnamed, while all named data streams will contain information about the default unnamed data stream. Figure 32 illustrates how the alternate data streams are stored on disk.

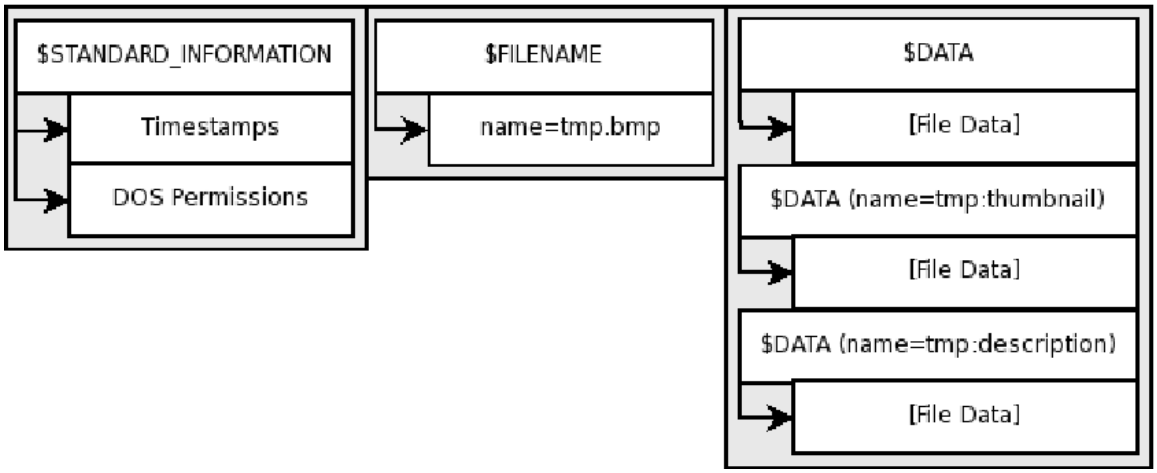


Figure 32: Various alternate data streams associated with a single file

Each file that contains data streams will have a $STANDARD_INFORMATION attribute, a $FILENAME attribute and one or more $DATA attributes (which may be resident or non-resident) associated with it. The use of alternate data streams may be useful in situations where extra data (such as descriptions) is associated with a file and managed as a single unit [73].

As an example consider figure 32. An image file named tmp.bmp may have a thumbnail image as well as a file description associated with it. The thumbnail and description could be stored as alternate data streams associated with tmp.bmp. The alternate data streams may be accessed by any application that knows of its existence as alternate data streams are not shown by standard directory listing commands, such as the dir command.

The process of accessing a data stream in a file can therefore be summarized as follows: locate the correct $DATA attribute used by the file in question and read data from the attribute's contents as required.

10.5 Development of the Reco NTFS driver

The previous section discussed the concepts surrounding the physical structure of the NTFS file system. This section focuses on the development of a file system driver that can be used by the Reco platform. In order to develop such a driver, some platform design information is required to allow the reader to understand how a new file system driver can be added to the platform.

A quick study of the platform's logical layer should be sufficient to allow the reader to understand the class structure involved in the development of file system drivers. The main components used by the logical layer of the platform consist of the following classes:

- DeviceEmulator and BlockDeviceEmulator;

- Filesystem;
- FSMounter;
- FilesystemFactory.

The DeviceEmulator class describes the interface that should be provided by the various device drivers used by the Reco platform. All the classes with the purpose of acting as a device driver should therefore inherit and implement the interface provided by the DeviceEmulator class.

The purpose of the BlockDeviceEmulator class is to supply access to acquired disk images. The BlockDeviceEmulator class implements the methods specified by the DeviceEmulator class, as well as additional methods which addresses the specific needs of block devices.

The purpose of the Filesystem class is to supply an interface that should be specified by all classes that acts as file system device drivers. The BlockDeviceEmulator class is used by the Filesystem class, and its successor classes, to access mounted disk images in an intuitive manner. Each file system object may have to be initialised differently. The initialisation process is likely to be used only when a file system driver is loaded. It would therefore make sense to split the initialisation code from the actual implemented driver code to conform to good object-oriented practices. The purpose of the FilesystemFactory is to supply an interface that should be used by all classes responsible for the initialisation of file system drivers. Each file system driver should have a factory class associated with it.

The FSMounter class keeps a list of all the mounted file systems as well as all the file system factory classes used to initialise the file systems. The purpose of the FSMounter class is to supply the developer with an easy way to mount and un-mount file system drivers. Because all the file system mount and un-mount procedures are handled by this class, it is important that the class knows about all the file systems and file system factories used by the Reco platform.

When a new file system driver and its corresponding driver have been developed, reference to it should be added to the FSMounter class to ensure that they are utilized by the Reco Platform. A simplified view of the class design is illustrated in figure 33.

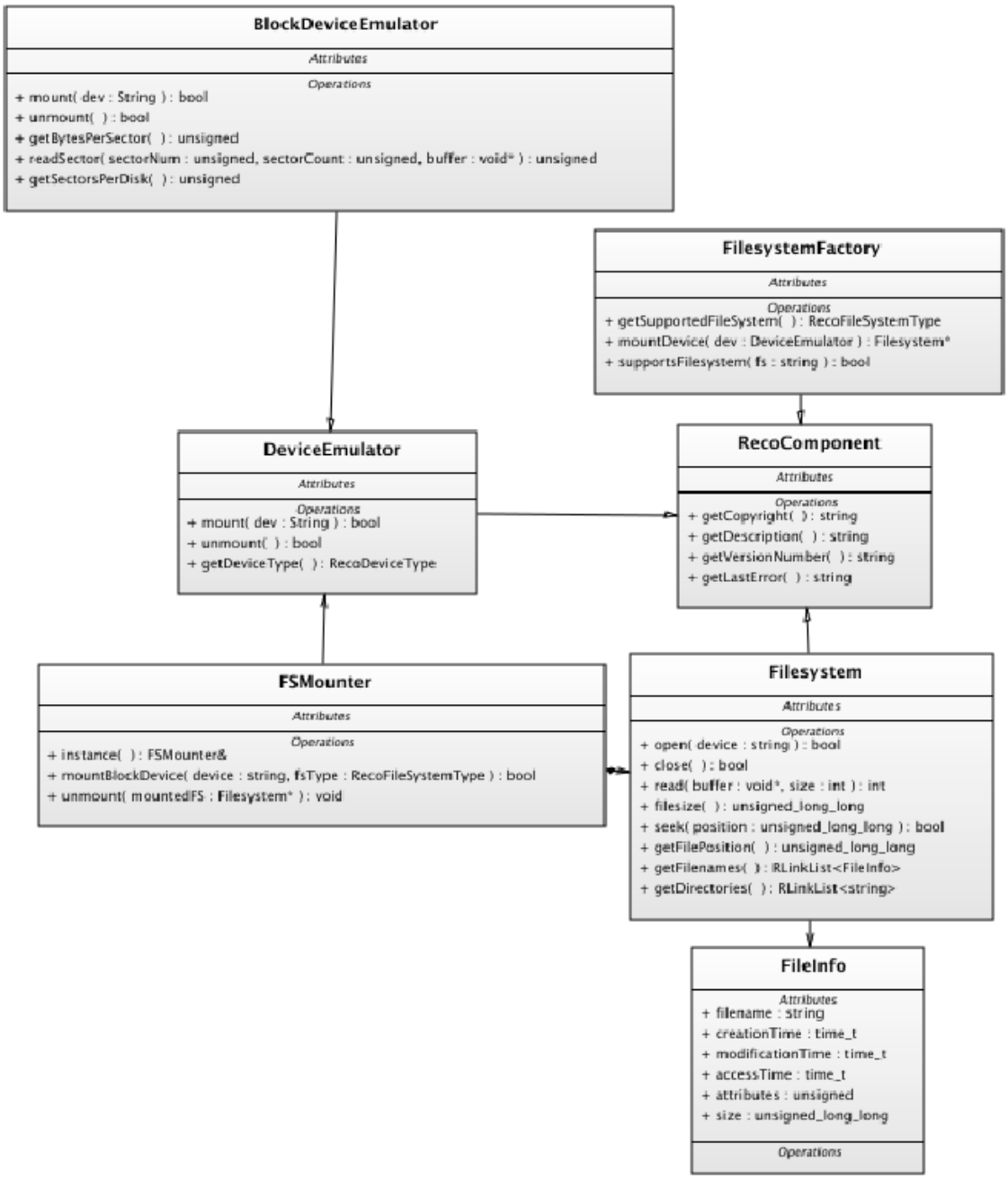


Figure 33: Classes used by the platform's logical layer

10.5.1 The Reco NTFS driver

The NTFS driver should conform to the specified platform design explained previously in order to be used by the Reco platform. The following sections discuss the development of the NTFS driver, paying special attention to the high-level implementation details for it to be compatible with the Reco platform.

A NTFS file may consist of one or more attributes. Every attribute is stored on disk using an ATTRIBUTE_HEADER structure (see appendix A). An NTFS file may consist of one or more attributes. Every attribute is stored on disk using an ATTRIBUTE_HEADER structure (see appendix A) and may contain metadata describing the file in question. All attributes have standard information associated with them, such as attribute name and length. From a design perspective all the information found in the ATTRIBUTE_HEADER may be stored in a base class, while any information specific to the attribute in question may be stored in a specialization of the base class. Figure 34 illustrates this design.

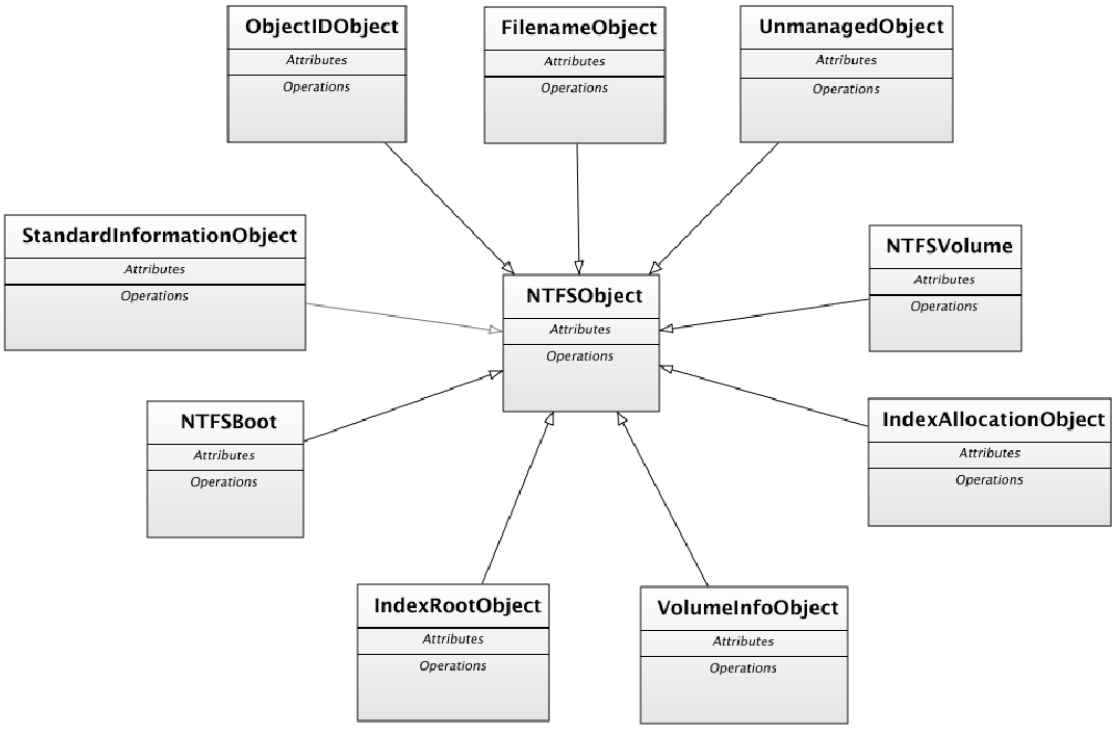


Figure 34: Class hierarchy used to implement the NTFS file system object

Upon inspection of the simplified class diagram it should be clear that the classes were designed to act as wrappers for the NTFS attributes specified in table 8. It should also be noted that not every attribute is supported in this release of the driver. All attributes not supported by the release are placed in an instance of the UnmanagedObject class. This allows all unused objects to still be accessible, even though they are not currently being used by the specified implementation.

Because an NTFS file contains various attributes, a composite class was created named NTFSFile that contains a collection of NTFS attributes associated

with a file. If any attribute of an NTFS file is required, its corresponding NTFSFile object should be queried as it contains the relevant attribute. The specialized NTFSObject classes contain relatively little application logic as the purpose of these classes is to act as wrappers for attribute data read from disk. The application logic responsible for the management of the NTFS objects and the interpretation of the stored image information are located in the NTFSVolume class.

The NTFSVolume class is responsible for the management of file information (and directory-related information as a directory is also seen as a file). All file-related queries are processed directly or indirectly by the NTFSVolume class. The class is also responsible for any disk or image access; all read requests are therefore handled by the NTFSVolume class.

The NTFSVolume class was designed to extract every NTFS file object from the $MTF table and convert them to objects stored in memory. In order to do this, application logic was developed to allow the class to locate the $MFT and extract the information according to the unofficial NTFS specification provided by Russon and Fledel [88].

When an NTFS volume is mounted, an NTFSVolume object will parse the $MFT table to extract a list of all the objects stored on the file system. All the objects are then kept in memory for later access. This technique has the advantage that it allows fast access to objects once all the objects have been loaded into memory, but loading all the objects stored on the file system may take a relatively long time to complete.

Another problem with this approach is memory usage - thousands of files may be stored on an NTFS volume. Storing representations of these files in memory may consume a considerable amount of resources. To minimize the amount of space used by the objects, it was decided that all data attributes should only be accessed once required. This means that only file metadata is stored in memory and not the data itself, thus lowering the memory usage to an acceptable range.

Another design issue that had to be addressed was the similarities between the information stored by an $INDEX_ROOT attribute and an

$INDEX_ALLOCATION attribute. Both have the capability to contain directory-related information. The main distinction between the attributes is that the $INDEX_ROOT attribute contains small directory structures (small enough to fit into an MFT entry which makes the entry resident). Larger directory structures are stored in an $INDEX_ALLOCATION attribute (which is non-resident and is therefore accessed from disk). Once the data has been extracted that contains the directory-related information (from the $MFT entry or from disk), the method used to extract the directory information is similar for both attributes.

A base class called DirectoryProcessor (see figure 35) was introduced, which contains the processing logic to extract directory information from processed file attribute information. The purpose of the IndexAllocationObject

and the IndexRootObject classes is therefore to extract the data containing the directory-related information and pass it to the DirectoryProcessor base class for processing.

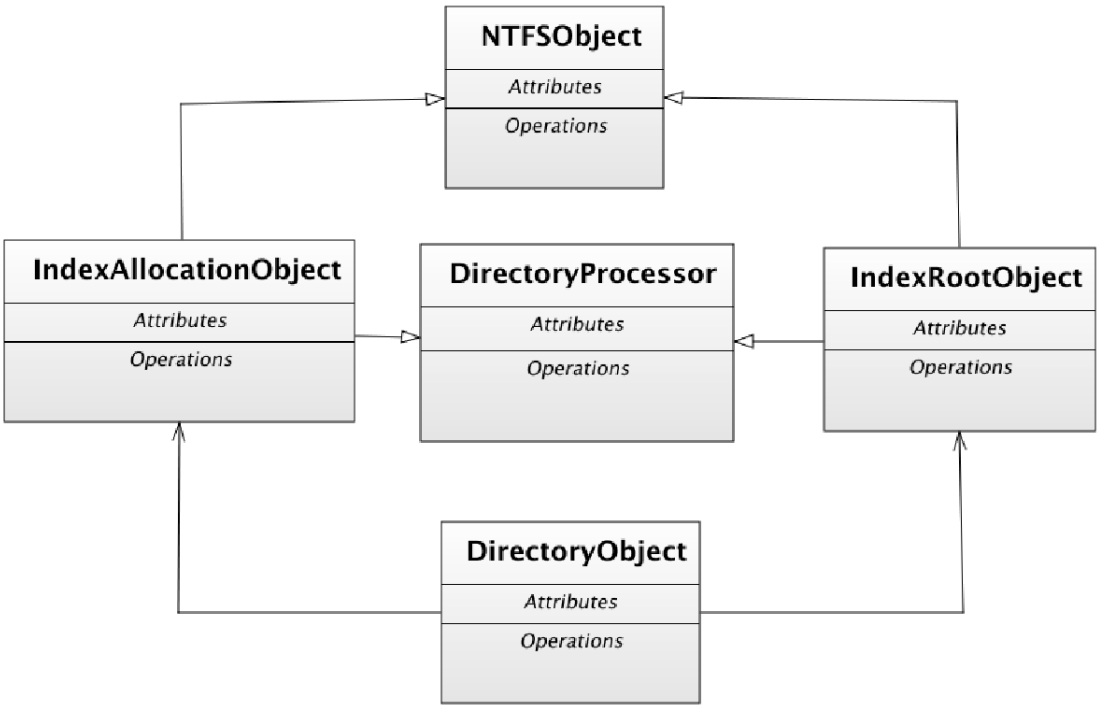


Figure 35: Design of the DirectoryProcessor class

The last aspect of the designed NTFS driver that needs discussion is the concept of reader objects. Ignoring forensic requirements for a moment, it can be assumed that normal file system use consists largely of file and directory access. To facilitate this need, two objects were introduced, namely the FileReader to allow access to a file's data attribute, and the DirectoryReader to supply access to data stored in an attribute containing directory information.

A directory's content is largely determined by the size of the directory in question; small directories may be resident, while larger directories will be non-resident. An $INDEX_ALLOCATION object does not have to be present for a directory entry, but an $INDEX_ROOT entry will always exist. This means that an $INDEX_ROOT attribute should always be scanned for entries when a directory scan request is made. If an $INDEX_ALLOCATION attribute is present, it should also be scanned, and the results should be appended to the results obtained from scanning the $INDEX_ROOT attribute. The purpose of the DirectoryReader object is to perform this described operation in an attempt to get an accurate listing of the files stored in the directory in question.

The FileReader object was designed to allow easy access to an object's

associated data attributes. The class allows the developer to access the content of a file without having to know how to manage resident and non-resident data runs. The FileReader object therefore makes it possible to access a file in the same manner as using an operating-system file access routine.

10.5.2 NTFS driver and the Reco Platform

The class structure described previously does not make use of the specified Reco interfaces and is therefore not Reco compatible. To use the classes in the Reco platform, two additional classes were introduced: the NTFSFactory and NTFSFileSystem as illustrated in figure 36

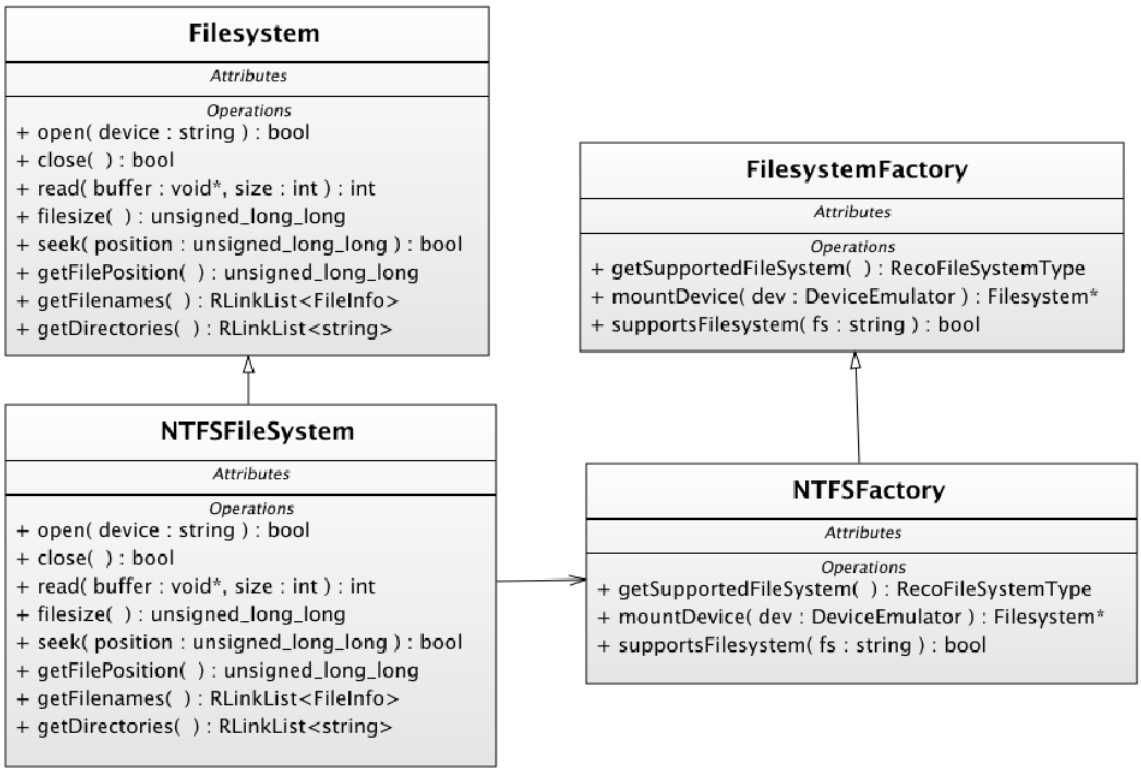


Figure 36: Making the driver platform-compatible

The purpose of the NTFSFactory class is to create an instance of the NTFSFileSystem object using a target NTFS image. The NTFSFileSystem object may be seen as an implementation of the Adapter design pattern [44] as it adapts the parameters and results sent and received and contains copies of the various NTFS objects discussed previously. These additional classes are used by the platform to interact with the developed NTFS driver classes.

Every file system driver that is to be used by the platform is therefore required to implement subclasses of the following classes:

- Filesystem;
- FilesystemFactory.

The implemented classes are used by the platform as a mechanism to interact with the driver object. Each file system factory object should be registered in the FSMounter object to enable the platform to know of the newly created driver's presence.

10.6 Results

This section discusses the results obtained from the NTFS driver which was developed and integrated into the platform. The entire development process took about one month to achieve. The developed NTFS driver has only enough functionality to read directory structures and access files. It therefore supports only the core functionality needed to extract file and directory information from an NTFS volume.

The NTFS driver was tested using various NTFS file system images. The driver was able to access files and displays the directory structures stored on the disk images. Unfortunately stability issues were experienced with the use of the driver. Some of the system images caused the driver to fail due to the use of additional NTFS functionality that was not implemented in this driver.

From a forensics point of view, the functionality provided by the custom driver is likely to be similar to the functionality provided by existing drivers. The only possible difference between a custom driver and an existing driver may be the fact that the former has the capability to detect and allow access to deleted files. It can be argued that this ability of the driver is not that unique as an open-source driver could have been modified to provide the same functionality.

Realistically speaking, it would not make sense to develop a custom driver without a good reason, based on the pure fact that it may take less time to integrate an existing, stable solution to the platform. Custom driver development should therefore be kept to a minimum, if possible.

10.7 Conclusion

This chapter discussed the development of a Reco-compatible file system driver that may be used to access NTFS disk images. A detailed discussion was conducted on the inner -workings of the NTFS file system, as well as the requirements imposed on a driver that should be Reco-compatible. The implementation of the driver was also discussed in great detail.

The purpose of the development of the NTFS file system driver was to determine whether or not the development of custom drivers has a significant impact on the value added to an investigation. Tests were performed using the custom driver that showed that the driver is capable of performing basic operations required by the platform.

Like any technology project, the development of the driver was developed under a strict time constraint, in this case a period of one month. Due to this steep time constraint only basic capabilities were implemented. It can be argued that every single information technology project will have a tight schedule to adhere to, which means that existing technology such as drivers should be implemented where possible.

If an open-source NTFS driver had been used instead of the custom-developed NTFS driver, the platform could have used a more stable, fully-featured driver that took less time to implement. It is therefore concluded that custom driver development should only be undertaken if no other feasible option exists.

11 Conclusion

11.1 Conclusion

This dissertation describes the development of a forensics platform that allows researchers to produce research prototypes requiring less time, while utilizing fewer resources. The platform supplies a set of rich libraries that may be used when developing research prototypes.

A platform prototype, named the Reco Platform, was created to allow the development of proof-of-concept research prototypes. The prototype was developed in C++ and is capable of compiling and running in both Windows and Linux. The prototype utilizes a layered architecture which, in essence, allows the prototype developer to choose the level of abstraction required when developing prototypes. The layered architecture therefore promotes flexibility as developers are not forced to incorporate certain designs that may be limiting or add little value to their prototypes.

Research prototypes were developed using the platform prototype, to prove that the platform supplies features to enhance the prototype development process. Two seemingly unrelated prototypes were developed using the platform, in an attempt to showcase the functionality provided by the platform, as well as the ease with which the prototypes were developed.

The first research prototype demonstrated a technique that can be used to access files on a live system without modifying any associated meta data. The prototype used the functionality supplied by the platform's interpretation layer to gain access to mounted disk partitions. Since disk access was performed using the read-only platform driver, no meta data modifications, such as file timestamps, occurred. The development of such a prototype would typically be a very complex process which may take months to achieve. However, the development of this prototype required relatively few lines of code and the implementation of the entire prototype took a single weekend to complete.

The second prototype was developed to calculate the last possible time at which an application could have been running, called the “last possible execution time”. This last possible execution time can be used to rule out applications that could have been used to modify various files located on a disk image. The last possible execution time may be valuable in situations in which a computer system was compromised. A system administrator may use the calculated last possible execution times for every application on the system, in order to determine which applications could have been targeted by the attacker(s). This prototype was developed once again with minimal effort and requiring relatively few lines of code.

Although the two demonstration research prototypes are considered to be simplistic, it can be argued that both of them contain elements that could be prevalent in other research prototypes as well. Since the platform managed to decrease the implementation complexity involved in the development of such

prototypes, it can be argued that the platform could be of use to prototype developers who require similar functionality. It can therefore be concluded that the platform can be used to increase the speed at which forensics prototypes are developed, while minimizing the implementation complexity and the amount of development time required.

11.2 Publications

The following publications were obtained as a result of the submission of various chapters in this dissertation to well-known conferences. The publications include:

R Koen and MS Olivier, “An Open-Source Forensics Platform”, Proceedings of the Southern African Telecommunication Networks and Applications Conference 2007 (SATNAC 2007), Sugar Beach Resort, Mauritius, September 2007.

R Koen and MS Olivier, “A Live-System Forensic Evidence Acquisition Tool”, Proceedings of the Fourth Annual IFIP WG 11.9 International Conference on Digital Forensics, Kyoto, Japan, January 2008.

R Koen and MS Olivier, “Using Timestamps Relationships As A Forensic Evidence Source”, Proceedings of the 7th Annual ISSA Conference, Johannesburg, South Africa, July 2008.

11.3 Future work

At the time of writing, the platform is still an academic proof-of-concept. This implies that it is not suitable for use in commercial or forensic environments. The platform will therefore have to undergo strenuous testing, and possibly some form of certification, in the future, in order to ensure its suitability for use in commercial and forensic environments.

The functionality supplied by the platform focuses largely on the provision of disk access routines to the prototype developer. Other equally important forms of evidence, such as network traces and memory dumps, are not supported by the platform as yet. Future versions of the platform will have to be extended to supply support for other forms of digital evidence.

Another problem with the prototype that requires some attention is the limited functionality supplied by its default plug-ins, compared to its commercial counterparts. The functionality provided by the existing plug-ins will need to be improved and new plug-ins will have to be developed, in order to compete with the vast feature sets supplied by commercial forensics tool kits.

References

[1] AccessData. Forensic toolkit sales and promotional summary. Available online: http://www.accessdata.com/media/enus/print/techdocs/Forensicas on July 2008.

[2] AccessData. *Forensic Toolkit User Guide*. USA, 2004.

[3] Adelstein, F. Live forensics: diagnosing your system without killing it first. *Commun. ACM 49*, 2 (2006), 63–66.

[4] Barkley, J. F., Cincotta, A. V., Ferraiolo, D. F., and Kuhn, D. R. Role-based access control (rbac): Features and motivations. In *11th Annual Computer Security Applications Proceedings* (1995).

[5] Bell, D. E., and LaPadula, L. J. Secure computer system: Unified exposition and multics interpretation. Tech. Rep. MTR-2997, The MITRE Corporation, Mar. 1976.

[6] Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering,20th Anniversary Edition*. Addison-Wesley, 1995.

[7] Broomfield, M. Ntfs alternate data streams: focused hacking. *Network Security 2006*, 8 (2006), 7–9.

[8] BrunoLinux.com. Fstab and mtab. Available online: http://www.brunolinux.com/02-The Terminal/Fstab and Mtab.html as on June 2007., 2007.

[9] Buchholz, F. P., and Spafford, E. H. On the role of file system metadata in digital forensics. *Digital Investigation 1*, 4 (2004), 298–309.

[10] Burmester, M., and Mulholland, J. The advent of trusted computing: implications for digital forensics. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), ACM, pp. 283–287.

[11] Card, R., Ts'o, T., and Tweedie, S. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (1994).

[12] Carrier, B. Autopsy forensic browser. Available online: http://www.sleuthkit.org/autopsy as on July 2008. 137

[13] Carrier, B. Open source digital forensics tools: The legal argument, @stake research report, 2002.

[14] Carrier, B. Defining digital forensic examination and analysis tools using abstraction layers. *International Journal of Digital Evidence 1* (2003).

[15] Carrier, B. D. Risks of live digital forensic analysis. *Commun. ACM 49*, 2 (2006), 56–61.

[16] Carrier, B. D., and Grand, J. A hardware-based memory acquisition procedure for digital investigations .

[17] Carrier, B. D., and Spafford, E. H. Automated digital evidence target definition using outlier analysis and existing evidence. In *DFRWS* (2005).

[18] Casey, E. Practical approaches to recovering encrypted digital evidence. *International Journal of Digital Evidence 1*, 3 (2002).

[19] Casey, E. Uncertainty, and loss in digital evidence. *International Journal of Digital Evidence 1*, 2 (2002).

[20] Casey, E. Network tra_c as a source of evidence: tool strengths, weaknesses, and future needs. *Digital Investigation 1*, 1 (2004), 28–43.

[21] Casey, E. Investigating sophisticated security breaches. *Commun. ACM 49*, 2 (2006), 48–55.

[22] Casey, E., and Stanley, A. Tool review - remote forensic preservation and examination tools. *Digital Investigation 1*, 4 (2004), 284–297.

[23] Chau, J. Digital certificates is their importance underestimated? *Computer Fraud and Security 2005*, 12 (2005), 14–16.

[24] Chen, K., Yu, L., and Schach, S. R. Measuring the maintainability of open-source software. *Empirical Software Engineering, 2005. 2005 International Symposium on* (2005).

[25] Christley, S., and Madey, G. Analysis of activity in the open source software development community. In *40th Annual Hawaii International Conference on System Sciences, 2007* (2007), pp. 1530–1605.

[26] Comino, S., Manenti, F. M., and Parisi, M. L. From planning to mature: On the success of open source projects. *Research Policy 36*, 10 (December 2007), 1575–1586.

[27] Coppin, B. *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., USA, 2004.

[28] Corey, V., Peterman, C., Shearin, S., Greenberg, M. S., and Bokkelen, J. V. Network forensics analysis. *IEEE Internet Computing 6*, 6 (2002), 60–66.

[29] Cowan, C. Software security for open-source systems, 2003.

[30] Craiger, P., Ponte, L., Whitcomb, C., Pollitt, M., and Eaglin, R. Master's degree in digital forensics. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences* (Washington, DC, USA, 2007), IEEE Computer Society, p. 264b.

[31] DFLabs. Ptk - an advanced free alternative sleuthkit interface. Available online: http://ptk.dflabs.com as on July 2008., 2008.

[32] DOMEX. Hashkeeper. Available online: http://www.usdoj.gov/ndic/domex/hashkeeper.htm as on August 2008.

[33] Eckstein, K. Forensics for advanced unix file systems. In *Proceedings of the 2004 IEEE workshop on information assurance* (2004), pp. 377–385.

[34] Ford, R. Open vs. closed: which source is more secure? *Queue 5*, 1 (2007), 32–38.

[35] Fowler, M. Is design dead? *Extreme programming examined* (2001), 3–17.

[36] Francia, G. A., and Clinton, K. Computer forensics laboratory and tools. *J. Comput. Small Coll. 20*, 6 (2005), 143–150.

[37] Free Software Foundation. Gpl-incompatible free software licenses. Available online: http://www.gnu.org/licenses/licenselist.html#GPLIncompatibleLicenses as on July

[38] Free Software Foundation. strip(1) - linux man page. Available online: http://linux.die.net/man/1/strip as on July 2008.

[39] Free Software Foundation. Open source initiative osi - the gpl:licensing. Available online: http://www.opensource.org/licenses/gpl-2.0.php as on July 2008., 1991.

[40] Free Software Foundation. Open source initiative osi - the lgpl: Licensing. Available online: http://www.opensource.org/licenses/lgpl-2.1.php as on July 2008., 1999.

[41] FreeDOS.org. The freedos project. Available online: http://freedos-32.sourceforge.net as on April 2007.

[42] Gacek, C., Lawrie, T., and Arief, B. The many meanings of open source, 2002.

[43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[44] Garber, L. Encase: A case study in computer-forensic technology. *IEEE Computer Society's Computer Magazine January 2001* (2001).

[45] Goth, G. Open source meets venture capital. In *IEEE Distributed Systems Online* (2005), vol. 6, IEEE Computer Socienty, pp. 2–6.

[46] Greiner, S., Boskovic, B., Brest, J., and Zumer, V. Security issues in information systems based on open-source technologies. In *EUROCON* (2003), pp. 12–15.

[47] Guidance Software. *Encase Forensic Edition User Manual, Version 4*. USA, 2004.

[48] Haggerty, J., Llewellyn-Jones, D., and Taylor, M. Forweb: file fingerprinting for automated network forensics investigations. In *e-Forensics '08: Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop* (ICST, Brussels, Belgium, Belgium, 2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–6.

[49] Horswell, J. *The Practice Of Crime Scene Investigation*. CRC Press, USA, 2004.

[50] Jansen, W. A., and Ayers, R. An overview and analysis of pda forensic tools. *Digital Investigation 2*, 2 (2005), 120–132.

[51] Kenneally, E. E. Gatekeeping out of the box: Open source software as a mechanism to assess reliability for digital evidence. *Iirginia Journal of Law and Technology 6*, 1 (2001).

[52] Kerr, F. C. Media analysis based on microsoft ntfs file ownership. *Forensic Science International* (2006), 44–48.

[53] Klber, R., and Galvo, M. Your computer forensic toolkit. *Revue /*

Journal Title Information systems security 10, 4 (2001), 49–60.

[54] Koch, R. *The 80/20 Principle. The secret of achieving more with less.* Nicholas Brealey Publishing, 1997.

[55] Koen, R. Reco platform homepage. Available online: http://sourceforge.net/projects/reco as on June 2007.

[56] Kozierok, C. M. Pcguide - ref - new technology file system (ntfs). Available online: http://www.pcguide.com/ref/hdd/file/ntfs as on November 2007.

[57] Kutchta, K. J. Computer forensics with the sleuth kit and the autopsy forensic browser. *The International Journal of Forensic Computer Science 1*, 1 (2006).

[58] Lally, S. J. What tests are acceptable for use in forensic evaluations? In *Professional Psychology: Research and Practice* (2003), vol. 35, pp. 491–498.

[59] Lawton, G. Open source security: Opportunity or oxymoron? *IEEE Computer 35*, 3 (2002), 18–21.

[60] Liebrock, L. Windows digital forensics toolkit: An analysis of digital forensics tools. Available online: http://infohost.nmt.edu/ cveitch/Documentation/ WindowsForensic CVeitch 103106.pdf as on July 2008., Nov. 2006.

[61] Loup Gailly, J., and Adler, M. zlib home site. Available online: www.zlib.org as on July 2008.

[62] Lyle, J. R. Nist cftt: Testing disk imaging tools. *International Journal of Digital Evidence 1*, 4 (2003).

[63] Manson, D., Carlin, A., Ramos, S., Gyger, A., Kaufman, M., and Treichelt, J. Is the open way a better way? digital forensics using open source tools. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences* (Washington, DC, USA, 2007), IEEE Computer Society, p. 266b.

[64] Mark Henley, R. K. Open source software: An introduction. *Computer Law and Security 24* (2008), 77–85.

[65] Massachusetts Institute of Technology. Open source initiative osi - the mit license:licensing. Available online: http://www.opensource.org/licenses/mit-license.php as on July 2008.

[66] McKemmish, R. What is forensic computing. In *Trends and Issues in Crime and Criminal Justice no. 118* (1999).

[67] McManus, D. J. A model of orginizational innovation: Build versus buy in the decision stage. *The International Journal of Applied Management and Technology 1*, 1 (2003).

[68] Mee, V., Tryfonas, T., and Sutherland, I. The windows registry as a forensic artefact: Illustrating evidence collection for internet usage. *Digital Investigation 3*, 3 (2006), 166–173.

[69] Mercuri, R. Challenges in forensic computing. *Commun. ACM 48*, 12 (2005), 17–21.

[70] Microsoft. CreateFile. Available online: http://msdn2.microsoft.com/en-us/library/aa363858.aspx as on June 2007.

[71] Microsoft. The file system. Available online: http://msdn2.microsoft.com/en-us/library/aa364972.aspx as on June 2007.

[72] Microsoft. GetLogicalDrives. Available online: http://msdn2.microsoft.com/en-us/library/aa364972.aspx as on June 2007.

[73] Microsoft. Working with file systems. Available online: http://technet.microsoft.com/en-us/library/bb457112.aspx as on November 2007.

[74] Mohay, G. Technical challenges and directions for digital forensics. In *SADFE '05: Proceedings of the First International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'05) on Systematic Approaches to Digital Forensic Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, p. 155.

[75] Moore, T. The economics of digital forensics. In *Fifth Workshop on the Economics of Information Security (June 26-28 2006* (2006).

[76] NSRL. National software reference library (nsrl) project web site. Available online: http://www.nsrl.nist.gov as on July 2008.

[77] Oberhumer, M. F., Molnr, L., and Reiser, J. F. Upx - the ultimate packer for executables. Available online: http://upx.sourceforge.net as on July 2008.

[78] Object Refinery Limited. Jfreechart. Available online: http://www.jfree.org/jfreechart as on July 2007.

[79] Palmer, G. A road map for digital forensic research. In *Report from the First Digital Forensic Resarch Workshop (DFRWS)* (2001).

[80] Parker, D. Windows ntfs alternate data streams. Available online: http://www.securityfocus.com/infocus/1822 as on November 2007., feb 2005.

[81] Petroni, J., Walters, A., Fraser, T., and Arbaugh, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3*, 4 (December 2006), 197–210.

[82] Pfleeger, C. P., and Pfleeger, S. L. *Security in computing, third edition*. Prentice Hall, USA, 2003.

[83] Raymond, E. S. *The Cathedral & the Bazaar (paperback)*. O'Reilly, January 2001.

[84] Redhat Inc. Redhat.com. Available online: www.redhat.com as on July 2008.

[85] Richard, G. G. Breaking the performance wall: The case for distributed digital forensics. In *Digital Forensic Research Workshop* (2004), DFRWS.

[86] Rogers, M. K., and Seigfried, K. The future of computer forensics: a needs analysis survey. *Computers & Security 23*, 1 (2004), 12–16.

[87] Rosenberg, J. Some misconceptions about lines of code. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics* (Washington, DC, USA, 1997), IEEE Computer Society, p. 137.

[88] Russon, R., and Fledel, Y. Ntfs documentation. Available online: http://data.linux-ntfs.org/ntfsdoc.html.gz as on November 2007.

[89] Salus, P. H. *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[90] Sansurooah, K. An overview and examination of digital pda devices under forensics toolkits. In *Proceedings of 5th Australian Digital Forensics Conference* (2007).

[91] Sowe, S. K., Stamelos, L., and Angelis, L. Understanding knowledge sharing activities in free/open source software projects: An empirical study. *Journal of Systems and Software 81*, 3 (2008), 431–446.

[92] Stallard, T., and Levitt, K. Automated analysis for digital forensic science: Semantic integrity checking. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference* (Washington, DC, USA, 2003), IEEE Computer Society, p. 160.

[93] Stephenson, P. The right tools for the job. *Digital Investigation 1*, 1 (2004), 24–27.

[94] Tamura, Y., and Yamanda, S. Software reliability assessment and optimal version-upgrade problem for open source software. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 1333–1338.

[95] Trolltech. Qt free edition license agreement. Available online: http://doc.trolltech.com/3.0/license.html as on July 2008.

[96] Ts'o, T. Home page of theodore ts'o. Available online: http://thunk.org/tytso as on July 2008.

[97] Ts'o, T. libext2fs. Available online: http://www.chrysocome.net/libext2fs as on July 2008.

[98] University of California, Berkeley. Open source initiative osi - the bsd license:licensing. Available online: http://www.opensource.org/licenses/bsd-license.php as on July 2008.

[99] Walker, C. Computer forensics: bringing the evidence to court. Available online: www.infosecwriters.com/text resources/pdf/Computer Forensics to Court.pdf as on April 2007.

[100] Wall, L. Artistic license. Available online: http://en.wikipedia.org/wiki/Artistic License as on July 2008.

[101] Wang, S.-J. Measures of retaining digital evidence to prosecute computer-based cyber-crimes. *Comput. Stand. Interfaces 29*, 2 (2007), 216–223.

[102] Wee, C. K. Analysis of hidden data in the ntfs file system. Available

online:
http://www.forensicfocus.com/downloads/ntfs-hidden-data-analysis.pdf as on June 2007., 2006.

[103] Wikipedia. Device file system. Available online: http://en.wikipedia.org/wiki/Block device#Block devices as on July 2007.

[104] Wikipedia. Freedos. Available online: http://en.wikipedia.org/wiki/Freedos as on July 2008.

[105] Wikipedia. Synergy. Available online: http://en.wikipedia.org/w/index.php?title=Synergy&oldid=146089977 as on July 2008.

[106] Wikipedia. Theodore ts'o. Available online: http://en.wikipedia.org/wiki/Theodore TsJuly 2008.

[107] Wilsdon, T., and Slay, J. Digital forensics: Exploring validation, verification and certification. In *SADFE '05: Proceedings of the First International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'05) on Systematic Approaches to Digital Forensic Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, p. 48.

[108] Wu, M.-W., and Lin, Y.-D. Open source software development: An overview. *Computer 34*, 6 (2001), 33–38.

[109] wxwidgets.org. wxwidgert cross-platform gui library. Available online: www.wxwidgets.org as on June 2007.

[110] Zadjmool, R. Hidden threat: Alternate data streams. Available online: http://technet.microsoft.com/en-us/library/bb457112.aspx as on November 2007, 2004.

Appendix

12 Elementary types

Table 11: Elementary type data sizes.

Elementary type:	Size in bytes:
char	1
unsigned char	1
unsigned short	2
unsigned long long	8

13 NTFS structures

```
struct FileBoot {
char jmp[3];
char systemID [8];
unsigned short bytesPerSector;
unsigned char sectorsPerCluster;
char unused_1[7];
unsigned char mediaDescriptor[1];
char unused_2[2];
unsigned short sectorsPerTrack;
unsigned short numberOfHeads;
char unused_3[8];
char unknown_1[4];
unsigned long long sectorsInVolume;
unsigned long long LCN_MFT;
unsigned long long LCN_MFT_Mirror;
char clustersPerMFTRecord;
char unused_4[3];
unsigned char clustersPerIndexRecord;
char unused_5[3];
unsigned long long volumeSerialNumber;
} __attribute__ ((packed));

struct Resident {
unsigned attributeLength;
unsigned short attributeOffset;
unsigned char indexedFlag;
char padding;
//die attribute name (2 * attributeLength vir Unicode).
//char name[512];
} __attribute__ ((packed));

struct NonResident {
unsigned long long startingVCN;
unsigned long long lastVCN;
unsigned short offsetToDataRuns;
unsigned short compressedSize;
char padding[4];
unsigned long long allocatedSize;
unsigned long long realSize;
unsigned long long streamSize;
//data run:
```

```
} __attribute__ ((packed));

//4.2
struct AttributeHeader {
unsigned attributeType;
unsigned length;
unsigned char nonResedentFlag;
unsigned char nameLength;
unsigned short offsetToName;
unsigned short flags;
unsigned short attributeID;

union {
Resident resident;
NonResident nonResident;
};
} __attribute__ ((packed));

struct StandardInformation {
unsigned long long creationTime;
unsigned long long modificationTime;
unsigned long long mftChangedTime;
unsigned long long accessTime;
unsigned DOSFilePermissions;
unsigned maximumVersions; //0 wys dat versioning disabled is.
unsigned versionNumber;
unsigned classID;
unsigned ownerID; //indien 0 is quotas disabled.
unsigned securityID;
unsigned long long quotaCharged;
unsigned long long updateSequenceNumber; //wys na $UsnJrnl. indien 0 is feature disabled.
} __attribute__ ((packed));

struct FileName {
unsigned long long parentDirectory; //wys na die base record van die parent dir
unsigned long long creationTime;
unsigned long long modificationTime;
unsigned long long mftUpdateTime;
unsigned long long accessTime;
unsigned long long allocatedSize;
unsigned long long realSize;
unsigned flags;
```

```
char unknown[4]; //used by preparse and EA
unsigned char filenameLength;
unsigned char fileNamespace;
char filename[512]; //unicode
} __attribute__ ((packed));

struct IndexHeader {
unsigned indexEntryOffset;
unsigned indexEntrySize;
unsigned indexEntryAllocatedSize;
unsigned char flags;
//padding
char padding[3];
} __attribute__ ((packed));

#define INDEX_ROOT_FLAG_LARGE_INDEX 0x01
//2.24
struct IndexRoot {
unsigned attributeType;
unsigned collationRule;
unsigned indexAllocationEntry;
unsigned char clustersPerIndexRecord;
char padding[3];
IndexHeader index;
//padding...
} __attribute__ ((packed));

struct IndexAllocation {
//die fields is net geldig as die laaste flag nie geset is nie:
unsigned long long fileReference;
unsigned short indexEntryLength;
unsigned short indexStreamLength;
unsigned char flags;

//padding??
char unused[3];
//die volgende veld is net geldig indien die laaste flag nie geset is nie:
union {
//stream

//die field is slegs geldig indien die sub-node flag geset is:
unsigned long long indexAllocationAttributeVCN;
};
} __attribute__ ((packed));
```

```
struct IndexBlock {
unsigned magic; //INDX is die magic value
unsigned short updateSequenceArray;
unsigned short updateSequenceArrayCount;
unsigned long long logSequenceNumber;
unsigned long long indexBlockVirtualClusterNumber;
IndexHeader index;
} __attribute__ ((packed));

//2.29
#define INDEX_ENTRY_SUB_NODE 0x01
#define INDEX_ENTRY_LAST_NODE 0x02
struct IndexEntry {
//unsigned long long fileMFTRecord;
unsigned fileMFTRecord;
unsigned unknown;
unsigned short length; //multiple of 8 bytes
unsigned short keyLength;
unsigned short flags;
char padding[2];

//attribute:
//union key {
FileName filename;
//} __attribute__ ((packed));
} __attribute__ ((packed));
```