Master Thesis
Computer Science
Thesis no: MCS-2003:17
June 2003

Computer Forensic Text Analysis with Open Source Software

Christian Johansson

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Christian Johansson
E-mail: uncle@fukt.bth.se

University advisor:
Bengt Carlsson
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

A computer forensic investigation is not only dependent on correct and flawless analysis of the given data, but also of this analysis being conducted within a reasonable amount of time and effort. Given the exponential growth in size of hard disk drives coupled with the fact that many phases in a computer forensic investigation are still performed manually, the compromise between accuracy and completeness is becoming more and more of a problem. This paper concentrates on the text analysis process within computer forensics, focusing on the use of open source software. It discusses and examines the different techniques used in the possible future streamlining of said process.

Keywords: forensics, text analysis, natural language

Acknowledgments

The author would like to thank the following persons and institutions for the help and support they have given prior and during the writing of this thesis.

- Bengt Carlsson, my supervisor at Blekinge Institute of Technology, whose input and suggestions were invaluable during the completion of this thesis.
- Per Mellstrand, for proofreading the thesis and giving helpful advise.
- FUKT, the computer club at BTH, where I have spent the last five months investigating, coding, tearing my hair and finally writing this thesis.
- My former girlfriend, for showing a delightful flair for timing when breaking up with me at the beginning of this project, thus enabling me to sit at FUKT all through the nights without feeling guilty about it.

TABLE OF CONTENTS

1 INTRODUCTION

Forensics is the application of science in order to produce items of evidentiary value in regards to civil and criminal investigations. This is a complex process of preserving, documenting, extracting and analyzing various amounts of potentially evidentiary material. Since the product of these activities have the potential to crucially impact on peoples lives, e.g. by sending them to jail for 20 years, there lies a great amount of responsibility on those conducting these activities and of course on the scientific level of the activities themselves. Forensics has a long history, e.g. around the year 700 fingerprints were used in China to establish identity of clay figures and documents [1]. During the last some two hundred years the art of forensics has matured and developed significantly, now encompassing such diverse areas as entomology, ballistics and pathology, and is now regarded as standing on a solid scientific foundation. Although development is forever ongoing the understanding and acceptance of forensic techniques and methodology is solid throughout both the scientific and juridical communities.

Computer forensics is a subset of forensics, concentrating on the handling of evidential material relating to computers and other electronic devices such as cellular phones and Personal Digital Assistants. Just as with any other discipline within forensics, computer forensics has some basic and well-defined methodologies that remain consistent. And as with the other disciplines, these methodologies need to be coupled with flexibility and intuition in order to successfully handle potentially criminal material [2]. Compared with other disciplines within forensic science, computer forensics has been around for a relatively short amount of time. Partly because of this, computer forensics has yet to amass the broad acceptance other forensic disciplines enjoy. This will hopefully change in the not too distant future, as the ongoing research and overall interest in computer forensics has increased during the last few years.

Something that sets computer forensics apart from other forensic disciplines is the unsurpassed, and seemingly forever ongoing, increase of data that needs to be processed. In the eighties, when the first small hard disk drives appeared, a 5.25-inch hard disk held about five to ten Megabytes of data. Whereas today it is not uncommon for ordinary home computers to feature something like two hundred Gigabytes of data space. Applied to forensic pathology this is the equivalent of on average having two bodies to process twenty years ago, and today on average having about eighty thousand corpses to examine in each and every crime scene. Most forensic pathologists would pass out when faced with such a development, but this is reality in computer forensics. However, processor speed has increased in a similar way throughout the years, and herein lays the salvation. To put it simply, in order to keep up with the increasing amount of data, more and more processes have to be automated.

As part of this automation development process this paper focuses on computer forensic text analysis. There already exists some commercial software for text analysis and searching, e.g., the dtSearch program [3]. Such programs read through the selected directories and create indexes, which are later used for the actual searches. Creating these indexes can take some time, but once this is done searches can be done with breathtaking speed. When conducting its searches dtSearch utilizes different levels of text analysis, ranging from regular expression like Boolean searches to fuzzy and phonic analysis. Such theories form the groundwork for this paper, although angling the discussion towards a computer forensic perspective. This means that it expands the current implementations and ideas to encompass computer forensic specifics like the

handling of ambient data, physical level analysis and the applying of degrees of interest in order to differentiate among large amounts of data.

When deciding how to implement the theoretical discussions, there are a few considerations. The first question is whether to build upon existing software or to implement something from scratch. In this case it was a simple question to answer since there is already a widely used open source computer forensic package available, namely The Sleuth Kit [4]. Together with the graphical front-end package Autopsy, The Sleuth Kit presents an in many regards viable option to commercial computer forensic products like EnCase [5]. The Sleuth Kit constitutes a convenient framework for further development, as the source code is free to use and modify as long as you adhere the various licenses accompanying the software. Furthermore it has a large user base [6], implying it has been thoroughly tried and tested since its creation.

The reasons to concentrate on freely available and modifiable open source software are simply as follows. As mentioned earlier computer forensic software has the ability to make or break people's lives in a very profound way, therefore anything but publicly available source code should be inadequate. This is because such source code can be scrutinized and tested by the entire computer forensic community, or any one else for that matter, instead of being only accessible to a small group of developers within a single company. Consider for example who in their right mind would trust an encryption algorithm that has not been subjected to the full scrutiny of the research community with peer-review and publicly available papers? As time goes on this will only become more of an issue, as more and more processes within computer forensics need to be automated in order to keep pace with the forever-increasing amount of information at hand. Further argumentation on the subject of open source computer forensic software can be found in a paper by Brian Carrier, the author of The Sleuth Kit [7]. He argues that there need not be a conflict between commercial development of computer forensic tools and its open source counterpart. By keeping the core functions open source and freely available for peer-review they will be kept as updated and efficient as possible. In any case the bulk of most commercial implementation code does not pertain to these core functions, but on different ways to handle and present the extracted data. In other words, only some smaller parts of the implementations need to be open source, which would guarantee the vendors ample freedom in producing competitive and commercially viable computer forensic software. Which is exactly the situation with the encryption algorithms mentioned earlier; the actual core algorithms are well known, so it is the surrounding environments that separate the different software vendors. This satisfies both the need for trust in the core functions by the end users and the need for commercially viable endeavors by the companies financing the development.

2 THE SLEUTH KIT AND AUTOPSY

2.1 An overview of The Sleuth Kit and Autopsy

In the year 2000 Dan Farmer and Wietse Venema released The Coroners Toolkit [8], a suite of utilities for conducting computer forensic examinations. The suite quickly gained widespread popularity as it featured quality computer forensic tools and was distributed as open source. Among the various utilities for investigating and collecting data were the `grave-robber`, the main tool used for gathering data, and `lazarus`, used to reconstruct previously deleted files. Furthermore, The Coroners Toolkit included tools for copying files by inode number, listing file system inode information, copying address space of running processes and handling file time reports. Its instant success implied that it filled a void within the computer forensic community, and it showed that open source computer forensic tools could be a viable and rewarding endeavor, able to compete with commercial and closed source software.

Although being a worthy first foray into open source computer forensics, The Coroners Toolkit had its share of limitations [9]. First of all, The Coroners Toolkit could only analyze the same operating system as it was running on, e.g. if you wanted to analyze a BSD/OS file system, The Coroners Toolkit would have to be installed and used on a BSD/OS machine. Secondly, The Coroners Toolkit lacked support for a number of the by far most common file systems, such as NTFS (New Technology File System) and different versions of FAT (File Allocation Table). And last but not least, The Coroners Toolkit only handled the block and inode layers, which meant that an examiner would only have access to a vast array of inode numbers when trying to figure out the overall file layout, which obviously is somewhat awkward for most people. These shortcomings prompted the need for further development.

The problem concerning only handling inode and block layers, was fixed when Brian Carrier released TCTUTILS [10]. TCTUTILS was an extension of The Coroners Toolkit and featured a collection of tools making mapping of different layers of the file system possible. This made it possible for examiners to list file and directory names, paving the way for more efficient and easily performed investigations. The two remaining problems were addressed when Carrier released The @stake Sleuth Kit (TASK) with the assistance of @stake [11]. TASK is a collection of computer forensic command line tools that merges, expands and builds upon the functionality of The Coroners Toolkit and TCTUTILS. Today development of this tool is independent from @stake and other commercial or academic organizations, which has prompted Carrier to change the name of the software package to The Sleuth Kit.

Although The Sleuth Kit is a, in the authors opinion, formidable computer forensic tool, it is still a command line based utility. In practice this means that, although fast and efficient for smaller investigations, The Sleuth Kit is somewhat cumbersome to use when handling larger sets of data. It also means that it can be an awkward task to handle the level of documentation prowess required when dealing with actual live cases. This is were Autopsy enters the scene [12]. This package is a graphical user interface to The Sleuth Kit, designed and developed by Brian Carrier in response to the just mentioned shortcomings of The Sleuth Kit. Actually, Autopsy is not only a simple graphical interface for The Sleuth Kit, it also adds further functionality like the ability to handle timelines and file type sorting, and for easing large scale case management. In addition to this, Autopsy also provides an easy-to-use interface for the basic Unix

commands `grep` and `strings`, by not only parsing program input and output, but also by automatically mapping the found segments to the files that allocated them along with their individual status.

All in all, The Sleuth Kit/Autopsy package of today is an outstanding computer forensic tool, developed with dedication and thorough knowledge and understanding about the actual needs of a computer forensic investigator. Along with the latest developments [13] this package is already rivaling the technical capabilities and ease-of-use of such high-end commercial packages as EnCase. However, as this paper concentrates on text analysis, it will refrain from further discussions regarding the overall software of The Sleuth Kit and Autopsy, and simply conclude that this package was chosen as a platform for the implementation of the coming discussions on text analysis. The Sleuth Kit and Autopsy are available at [14] and the author strongly recommends any interested readers to try them out. This paper continues with a short review of the `grep` and `strings` commands, coupled with a discussion on regular expressions, before proceeding with more specific text analyze phases in the coming chapter.

2.2 The `grep` and `strings` commands

The `grep` command is an integrated part of most Unix-like systems and is basically used for searching files for the occurrence of some input pattern. In its most basic form of use, `grep` simply searches the input file or files for the exact matches to the input and prints out the lines containing the given input. What makes `grep` so much more than a simple search tool is its ability to handle regular expressions [15]. Regular expressions are used by many Unix-like systems for text handling and parsing within their commands and utilities; not only `grep`, but also `sed`, `awk`, `vi` and `procmail` to name but a few. The programming language Perl also features one of the more extensive implementations of regular expressions. A comprehensive and thorough review of all features of regular expressions is far beyond the scope of this paper and interested readers are encouraged to seek enlightenment elsewhere. Although `grep` is a magnificent utility in its own right, as a standalone tool it can be rather cumbersome when dealing with computer forensic evidence, as there might be no actual file system to examine, just massive chunks of data imaged from a suspects hard disk drive. This means that an examiner manually has to perform the mapping of finds to actual files and folders, which can be rather time-consuming and error-prone. As mentioned before this is done automatically when using Autopsy, which saves time and reduces the possibility for error when handling the evidence.

The `strings` command is a tool mainly useful for examining the content of non-text files, e.g., executable files, music files or picture files. When used, `strings` prints the printable character sequences it finds throughout the file. By default the minimum length of these strings are 4 characters, but using input flags when calling `strings` can change that. There is not much more to comment on this utility, a simple but sometimes very useful and handy little tool. Autopsy uses it automatically in conjunction with `grep` when dealing with non-text files.

3 COMPUTER FORENSIC TEXT ANALYSIS

3.1 An overview of computer forensics text analysis

Traditional data search technologies are based upon looking for some exact match of the data being searched for and although this behavior is reasonably fast and effective as a standalone process, it is somewhat insufficient in a computer forensic domain. This problem can in some respects be alleviated by the skillful use of regular expressions, as discussed in chapter two, but to form a conclusive solution further consideration has to be done. This chapter will discuss both the techniques that can be covered within the realm of regular expressions and those who need further implementation.

3.2 Boolean style searching

As mentioned in the previous chapter, regular expressions can in principle handle more or less all Boolean searches, although the use of a Boolean search engine can to a great extent simplify the performance of such searching. Boolean searches include the AND, OR and NOT parameters, which are taken from classic logic theory [21]. Among literature dealing with Boolean searching within the text analysis domain, one sometimes comes across such concepts as "phrase searching" and "proximity searching". Phrase searching is simply done by grouping words together into sentences and using these, preferably by using AND searching. While proximity searching tries to locate occurrences of a word within certain distance from a second word, this kind of search is henceforth labeled WITHIN searching. These four search types (AND, OR, NOT and WITHIN) form the basis of the Boolean search engine, and by utilizing their capabilities more abstract searches like "narrow", "broaden" and "exclude" can be performed. Among text analyzing literature one can find several references to different kinds of Boolean search types, but they can mostly all be traced back to the functionality of these four core functions. Besides supplying the possibility of rather complex searching in their on right, a functional Boolean engine is important when implementing further functionality such as natural language systems, as these transparently make extensive use of such search methods.

3.3 Thesaurus searching

Consider, for example, searching for a word like "delicious". This word, like most other words, has an array of synonyms and words that are more or less closely related in their meaning. The basic idea of thesaurus searching is to automatically include all these synonyms and related words into the search, thus expanding the chance for a successful and relevant find. Obviously this is done by using a thesaurus database of some kind. Before the actual search starts, the search program accesses this database and retrieves the relevant words and includes these words into the search. This is not a very difficult thing to implement, only access a database and receive a few extra words to add into the search. The grimness of it all would only hit when trying to actually assemble the database itself into any degree of usefulness, seeing the immense wealth of words in the English language. Luckily this has already been done.

WordNet [16] [17] is one such database, developed and maintained by the Cognitive Science Laboratory [18] at Princeton University [19]. To cite the homepage of this project:

> "WordNet is an online lexical reference system whose design is inspired by current psycholinguistic theories of human lexical memory. "

Not only is it a great online resource for all would-be writers, it is also available for download to be incorporated in projects such as this. Using such a database it is not difficult to incorporate the power of thesaurus searching into The Sleuth Kit. With the implementation of WordNet availability and returning to the opening example of "delicious", a thesaurus enabled search might thus include such words as delectable, luscious, pleasant-tasting, scrumptious, toothsome and yummy.

Considering the computer forensic environment there can be many cases featuring similar characteristics, and not all of these might be included in a general thesaurus database covering an entire language. An example of this might be when dealing with cases including illegal substances such as anabolic steroids or drugs. By including a user-defined thesaurus in addition to the general one, specific word relating to such topics can be included, thus keeping pace with the forever increasing number of different substances to recognize or different street vocabulary and linguistics not usually found in a more general thesaurus. Other examples might include adding an array of different explosives, like C4, trimonite, amatex, cyclotol, dynamite and so on, whenever conducting a search for the word bomb. Given a generally accepted standard for interfacing such user-defined thesauruses, computer forensics related databases could be distributed among law officials and forensic personnel in order for all to keep up to date.

When discussing a standard for such a user-defined database, it is the author's opinion that the interface used by WordNet should be considered. The reason for this is that WordNet is already used to a great extent within search software, e.g. dtSearch incorporates WordNet in order to handle its thesaurus searches, and it must thus have proven to be a reliable piece of software [3]. Furthermore research and development within the WordNet project is ongoing and thriving, thus providing confidence in the continuing validity of the system. And of course, let us not forget one of the more important features of WordNet, it is free to use and modify as long as you adhere the license accompanying it.

3.4 Stemming

Stemming is the task of normalizing a certain word into its original status, the root or stem. This is done by removing all affixes contained in the word, e.g. "considering" becomes "consid" when the "-ering" part is removed. The reason for doing this is simply that a search using stemmed words will be more likely to turn up relevant hits. Using "considering" as the search word would miss both "considered" and "considers", unless you stemmed the word first. It might seem strange to stem "considering" into "consid" instead of "consider", but the reader should bear in mind that stemming is exclusively done in order to improve performance and the stemmed words might not always be linguistically pleasing to behold.

There exists a variety of different algorithms for stemming words. However, many of them can be traced back to the Porter Stemming Algorithm, devised by Martin Porter in the late seventies and early eighties [20]. The algorithm used in the

implementation derived from this paper features the Porter Stemming Algorithm. In addition to just conflating the search words before each search, it is also possible to stem all words within the process of building the index. This reduces the size and complexity of the index, making for more efficient and faster searches, and will be discussed further in the section concerning indexing.

3.5 Natural language

Natural language processing has proven to be somewhat more complicated that as first anticipated when research in this area began in the late fifties. Although experiencing large difficulties in producing systems that can handle general language understanding, natural language processing has had some success within certain, well defined, areas. Two such areas are Information Retrieval and Text Categorization, which both are relevant to computer forensic text analysis.

Most modern Information Retrieval systems use a vector-space model in order to find what they are looking for [21]. They treat every list of words as a vector with *n*-dimensions, where *n* is the number of unique words in the search data. The presence of a word in a certain list is indicated by the value 1, and the absence of a word is indicated by the value 0. Both search input and the search data is treated in this same way, and therefore performing a search is simply a matter of comparing these vectors. The nice thing about the vector-space model is that it allows for ranking of the search hits based on their closeness to the search input. In order to make these searches more relevant it is common to apply weights to each term, based on, for example, how often they appear within the total search data or their closeness to other highly relevant words. This is a rather basic way of analyzing text, as more or less all information is gathered from the words themselves, and the syntax and semantics of the sentences that they belong to are mostly discarded. However, this is often more than enough to capture the essence of a text document, and thus correctly evaluate its likeness to a set of search words.

Text Categorization is a related subject, and one which natural language processing has proven rather successful in. What Text Categorization does is to examine if a certain text belongs to a fixed topic category or not, hence the name. The success of Text Categorization is due the fact that the categories are fixed and the system can thus be tuned towards these respective interests. A prominent example of such application is the numerous online news services that collect and provide subscribers with information about user-defined topics. The use of such text analysis methods are only limited by the imagination of the user, e.g. surveillance agencies could, and probably already do, make use of such systems while supervising online communication.

The areas where natural language processing has been successful during the last few years all share two common treats; they focus on a single domain and a single task. Neither of these excludes the possibility for successful implementation of such systems within a computer forensic domain. Both Information Retrieval and Text Categorization are examples of such areas, and both are applicable in the computer forensic field.

3.6 Fuzzy searching and Phonic recognition

One very common task within computer forensic investigations is to analyze residual data pertaining to communication between people. This data can for example be e-mail messages, IRC session logs or ICQ data files. Due to the fact that many people seem to have a somewhat slack attitude towards linguistic correctness when composing these messages, they are rather likely to contain misspelled words, which may in turn make it more difficult to analyze the data using conventional methods. This is especially true since it is nearly always the significant words, as opposed to noise words as "or" and "he", that people have a hard time spelling, consider for example "nitroglycerine". A fuzzy search engine helps to alleviate this problem as it provides the ability to find words that are slightly misspelled, or even extremely and utterly wrong, all depending on the chosen level of fuzziness. These systems are usually based around a fuzzy expert system, which takes a first input and gives it a degree of likeness in relation to a second input. The range for these degrees usually lies between 0.1 and 1.0, but is often converted into the range of 1 to 10 within the graphical interface for the convenience of the human user. A likeness of 2 indicates a very low resemblance, and a likeness of 9 is given to an only slightly misspelled word. Based on this it is easy to filter the output at the desired level. Fuzzy searching works both ways, in that a misspelled search word will find a correct word in the search data.

Phonic recognition is somewhat similar. Two words are said to share a phonic resemblance when they sound alike but differ in spelling. This is usually implemented in a similar way as the fuzzy system; in fact the implemented expert system can often be used, although with another set of rules. When used in conjunction with fuzzy searching these two techniques form yet another set of tools with great use within most text analysis processes.

3.7 Indexing the search data

The concept of index based searches is as follows: Instead of accessing the entire amount of search data every time a search is being performed, a list or index of all significant words is constructed beforehand and it is this index that is then accessed in every search. This will obviously introduce a great performance boost since the amount of unique words is finite while the data to be searched is limited only by the size of the storage space at hand. For example, there is no need to access the entire hard disk drive during every search since all significant metadata is compiled into the index where it is easily and efficiently accessible. This metadata contains information about locations, frequency and so on about all significant words; exactly what it contains is dependent on such things as performance considerations and intended use of the system.

What constitutes a significant word is often based on a user defined list containing all words regarded as line noise, e.g. "if", "and", "or" and so on. In most cases there is simply no need to know the exact location of the two hundred thousand unique instances of the word "be". Since a substantial part of most text masses are composed of such line noise, there lies a great performance benefit in filtering them out. And since the file defining this noise is user controlled, it is easily modified should the need to examine all instances of "have" ever arise.

The idea of using index-based searches within computerized search engines is not a new one; libraries, scientific databases and web-based search engines to name but a few examples all use index tables in order to improve performance and overall

efficiency. After all, no-one expects Google [22] to rummage through the entire Internet every time someone presses the search button!

Stemming can be used when constructing the index to further enhance the performance by assuming that words with the same underlying stem share similar meaning, thus further cutting the size of the index. However, this process is somewhat error prone, and the actual performance benefits gained are to some extent forfeit by the errors introduced into the performed searches [23]. The level of acceptance of such errors is dependant on the situation at hand and under certain circumstances index stemming is desired and should be available. An alternative use of stemming is to refrain from stemming the actual index, and only stem the search words, thus widening the range of hits during searches. This latter behavior can especially be desirable in the beginning of an examination, as a method of gathering broader insights into the overall status of the data mass, prior to more fine-grained searches.

The main benefits of using an index-based system for analyzing data can be summed up into one word; speed. Imagine having two hundred Gigabytes worth of data to analyze, and every time the examiner presses the search button the system has to access every single part of that data. Even with today's fastest hard disk drives, this would be enough to infuriate the most patient forensic examiner. Obviously all the data has to be accessed at least once when creating the index, but when this process is complete any following searches can be executed with significant performance benefits compared to conventional search methods. The larger the size of data to be analyzed the larger the relative performance benefit of using index-based searching compared to not using indexing. The following chapter will among other things present conclusive evidence in favor of this allegation.

4 THE IMPLEMENTATION

4.1 Overview of the implementation

This chapter starts by explaining the construction of the index and thereafter continues to examine the use of the so far implemented aspects of the previously discussed analysis methods. Diagrams displaying actual program run characteristics will be presented in order to either support or discard ideas and suggestions made throughout this paper.

The computer used to perform the following tests was a Pentium 4, 2.80 Gigahertz machine with 512 MB of RAM. The exact same machine was used to boot both the Windows XP operating system and the FreeBSD 4.6 operating system, in order to get a consistent hardware environment for the tests. The dtSearch program was used in the Windows XP environment in order to produce comparison data. All further proof of concept implementations within this project was done in ANSI C and tested on the FreeBSD platform. It should be clear from the context in which they are found if a diagram pertains to dtSearch runs or originate in project derived implementations.

4.2 Indexing the data

As stated in earlier chapters the main reason for using index based searches it to speed up the process of multiple searches by removing the need to access the entire search mass in every unique search. The actual construction of the index file takes approximately the same time as one single conventional search, seeing that this is the only time the entire mass of data needs to be accessed. During this construction the metadata of every significant word within the data is recorded and assembled into the index files for future access.

The core size of the resulting index file in respect to the original text mass decreases as the original text mass grows bigger and bigger. This is simply a consequence of the fact that all languages feature a finite amount of words, and as the search data expands more and more words found will be duplicates, and will thus already be present in the index files. This however, is not the entire truth as the overall size of the resulting index files is largely dependant on the amount of metadata selected to represent every instance of a given word, at the very least this metadata needs to contain information about the location of the word. A shrewd representation of this data is needed in order to prevent the index file growing too large relative the size of the original data mass. A diagram comprising a series of tests using the dtSearch program is presented below in Figure 1 to illustrate the above discussion.

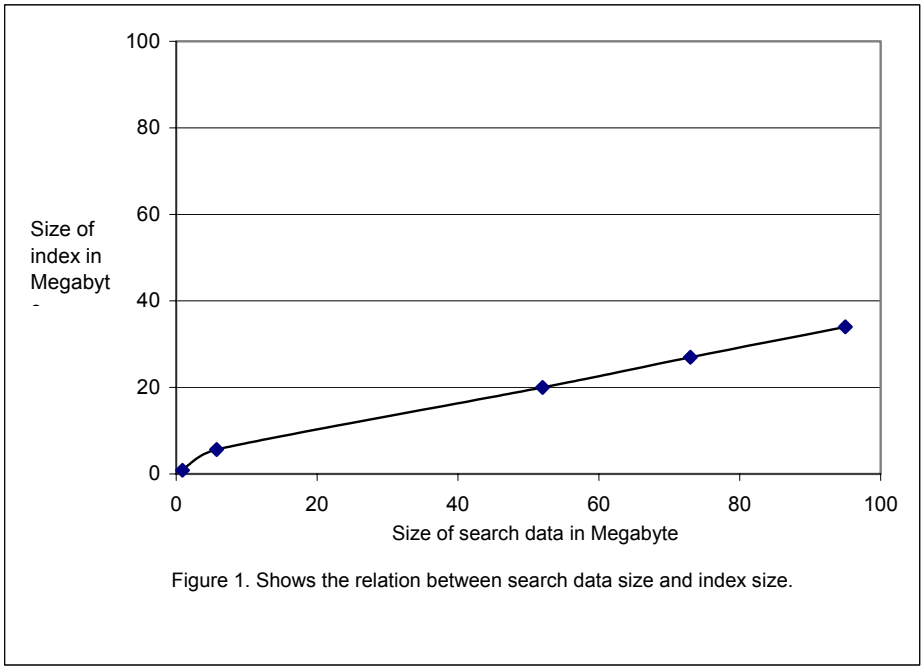

Figure 1. Shows the relation between search data size and index size.

In Figure 1 the X-axis represents the size of the original data mass in Megabytes, while the Y-axis show the size of the resulting index file compilation, also displayed in Megabytes. As is clearly visible the data/index file size ratio follows a more or less linear dependence, and thus the *absolute* amount of storage size gained increases with the size of the original data mass, while its *relative* counterpart does not. The reason for the linear development is that every found instance of a word is accompanied by certain metadata, increasing the size of the index files in a linear fashion. This in turn, implies that the *relative* time saving during multiple searches increases along with the original data mass. One might be forgiven for thinking that the relative time saving will be like the relative size saving, i.e. linear. However, remembering the previous discussions pertaining to index searching, the observant reader realizes that the core size of the index files does *not* follow this linear dependence. To put it simply; the larger the size of the original data mass, the better of an idea it is to index it. Figure 2 shows the data graph of a selection of test runs using the project derived implementation, and illustrates this point in a very obvious manner.

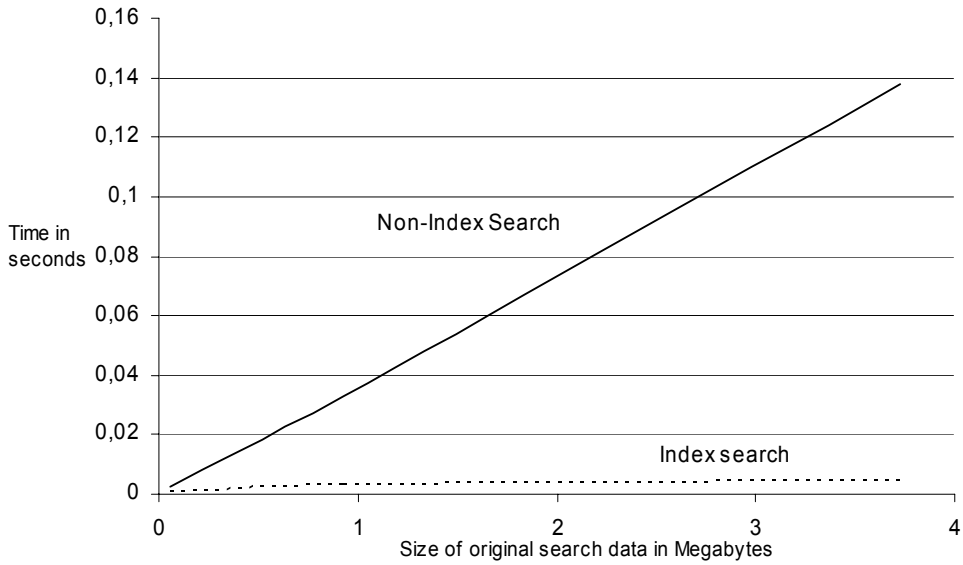

Figure 2. Compares the speed of Non-Index search vs. Index search.

Figure 2 clearly shows the enormous advantages of using index based searches. The X-axis represents the size of the original data mass, while the Y-axis shows the time it takes to rummage through the said data. The non-index search is obviously linear since doubling the amount of search data will double the time it takes to examine it, whereas the index based search will alleviate the need to search through endless amounts of word duplicates. In fact, the time it takes for the index based search to complete levels out as the unique-word count reaches that of the total amount of available words in the given language, thus *adding no further search time when adding arbitrary amounts of further search data*. Obviously the metadata part of the index files will continue to expand in a linear fashion while adding further search data, but this does not affect the time it takes to search the core index files.

Previously it was argued that the use of a noise file would introduce some performance benefits. The project derived implementation shows that this is indeed the case. The main benefits of using a noise file lies in the index construction process, and only to a smaller degree in the amount of time it takes to perform the subsequent searches. The resulting index files will shrink by a large margin resulting in a great save of storage space, which might often be a non-trivial issue when dealing with massive amounts of original search data. However, the core part of the index files will not shrink by a substantial amount, simply because the absolute majority of all words are generally not considered line noise, thus not generating any greater benefits in search time. But, as mentioned, the fact that a vast amount of storage space can be saved should encourage the use of noise files.

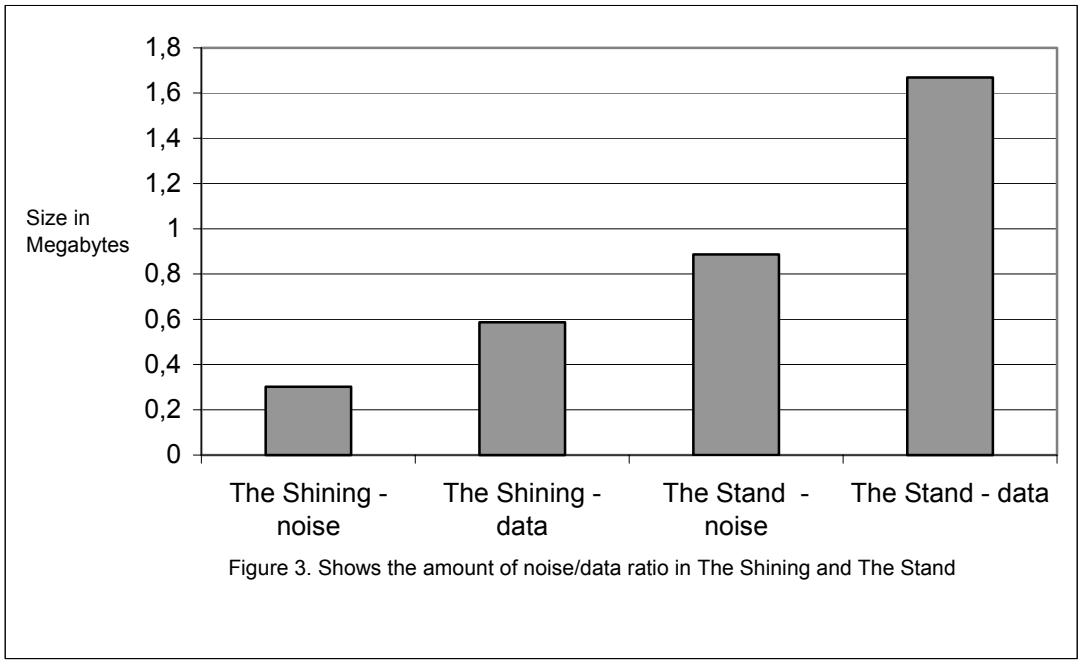

Figure 3. Shows the amount of noise/data ratio in The Shining and The Stand

Figure 3 shows the amount of line noise compared to significant data in two popular Stephen King books; The Shining and The Stand [24]. Although both these books contain a massive thirty-four percent line noise, this should not be taken as a dismissal of the literary capabilities of Mr. King, as this is a normal average in English fiction. Obviously the amount of noise filtered out is directly dependant on the content of the noise file, which is used when processing the search data. Keeping the noise file user defined is an absolute requirement, as this ensures that it can easily be modified to fit any situation that might arise. All in all, Figure 3 clearly illustrates the advantages of using a noise filter.

4.3 Thesaurus enabled searches

As mentioned in chapter three, the thesaurus of choice is the WordNet bundle. This package contains an entire application programming interface for using its many features within your own implementations. However, this proved to be somewhat cumbersome to incorporate in an orderly fashion into the current project derived implementation. The objective of this implementation is simple and clean; too quickly and efficiently produce synonyms and related words or expressions to any given search word. And thus a large portion of the code in the WordNet package was simply not needed.

The solution to this dilemma was to extract only the actual database files from the WordNet package, and implement a new and lightweight interface in order to access these files. This arrangement provides only the necessary features to accomplish the given task, thus reducing the required amount of code into a bare minimum. This produces a fast, reliable and time-efficient piece of software, while keeping it reasonably easy to maintain.

There should be three possible options when using the thesaurus search. First, and most obvious, is the option to not use it at all. Many times there might not be any use for this kind of search, and the possibility to ignore this behavior should always be

available. Secondly, the option to use automated thesaurus searches. This option automatically accesses the database, extracts the relevant data and then proceeds by inserting this data into the following search. This all happens without any interaction from the user, hence the inclusion of the word *automated* in this option. The third option is to allow for a user defined thesaurus search. The database is accessed, but instead of simply including all output into the search, the user is prompted to select the desired strings from the database output before inclusion into the following search.

The above discussion is applicable not only to the database files extracted from the WordNet package, but also to the user defined relevant-word databases introduced in chapter three. These special databases should by all means and purposes be incorporated in an identical fashion as the hitherto discussed WordNet database files. And using the lightweight interface ensures that this incorporation will be a streamlined process, since this implementation, unlike the WordNet package, has had this inclusion on the agenda all along.

4.4 Boolean searches

Implementing a Boolean search engine is somewhat like implementing a lightweight compiler. The different operators, e.g. AND or NOT, do not present any problem in themselves while implementing. However, the code needs to be able to handle all kinds of combinations of these single operators in conjunction with parenthesizes and actual data, which might require some care. Consider for example the difference of the following statements:

```
(donald AND goofy) OR mickey
donald AND (goofy OR mickey)
```

The speed of a Boolean search is heavily dependant on the nature of the search, for example an OR search can produce a result at the very instant it finds the first occurrence of one of its data components. A NOT search, on the other hand, has to examine the entire data mass in order to establish the fact that its negative data component is not present there. Before embarking on the task of producing massive regular-expression-like Boolean searches, one should be aware of the fact that Boolean searches are coupled with certain costs as far as time goes. The actual cost is largely dependant on the structure of the index files, since Boolean searches generally use metadata to a large extent in order to resolve the search. For example, consider the following statement:

```
donald w/10 duck
```

This statement searches for all occurrences of the string "donald" followed by the string "duck" no more than 10 words away. In order to resolve this search, the Boolean engine needs to be able to calculate, not only the actual presence of the two strings, but also their exact locations relative each other. Figure 4 gives some examples as to the time cost introduced by a sample of Boolean searches tested within the project derived implementations.

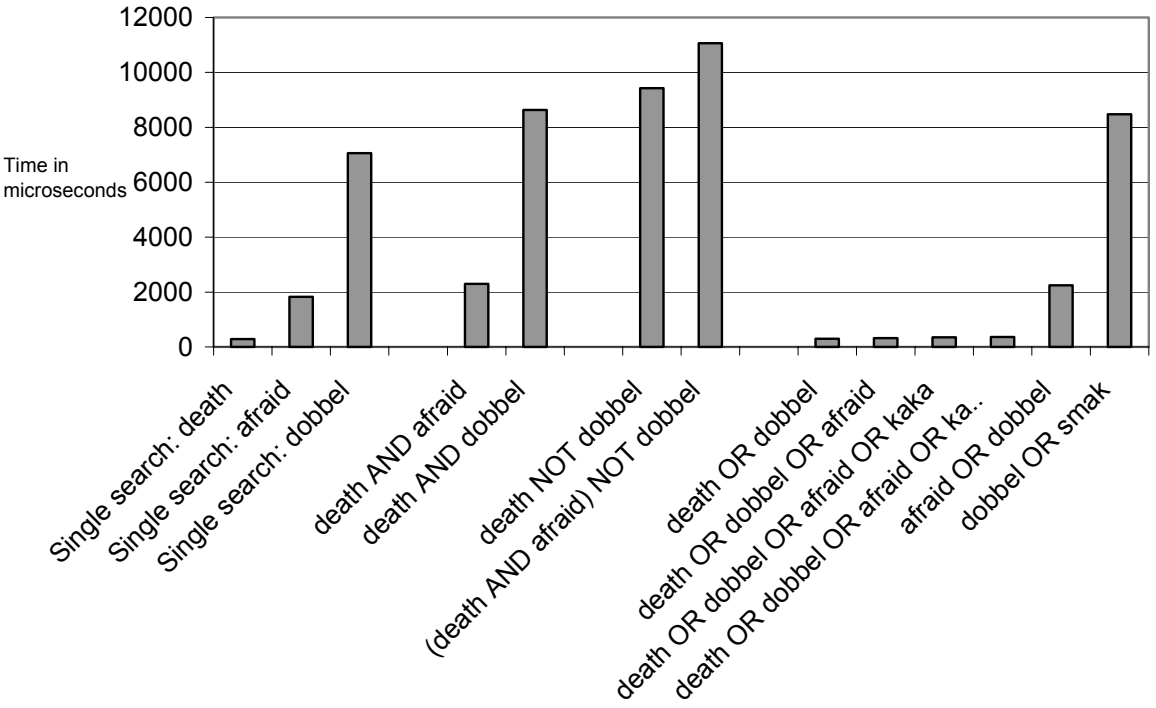

Figure 4. Shows the search time for different kinds of Boolean searches

The text mass used when constructing Figure 4 is the noise ridden The Shining by Stephen King. The Y-axis represents the search time in microseconds. The first three bars show single searches which halt when they find the first occurrence of the given word. As can be seen, the first occurrence of the string "death" is found rather early in the text, which should come as no surprise to fans of Mr. King. The first occurrence of the string "afraid" is found further into the text, and the string "dobbel" is not present at all within this book which thus illustrates the time it takes to traverse the entire text mass. The two following examples are AND searches which also halt when the search is true, and as can be seen, the behavior of these are as one might expect. The "death AND afraid" search halts after first finding the string "death" in the beginning of the text, and later on the string "afraid", thus taking only slightly longer to perform than the single search for "afraid". The "death AND dobbel" will return false, since "dobbel" is not present, and furthermore, it will once again have to traverse the entire text mass to establish this very fact. This is also the case with the following NOT searches, as they both need to make sure that any negative string is not present in the text mass. What then follows is an array of OR searches which halt when an occurrence of any of the input strings are found; as can be seen the first four halt on "death", the fifth on "afraid" and the last one returns false, since neither of the search strings are present in this fine piece of literature.

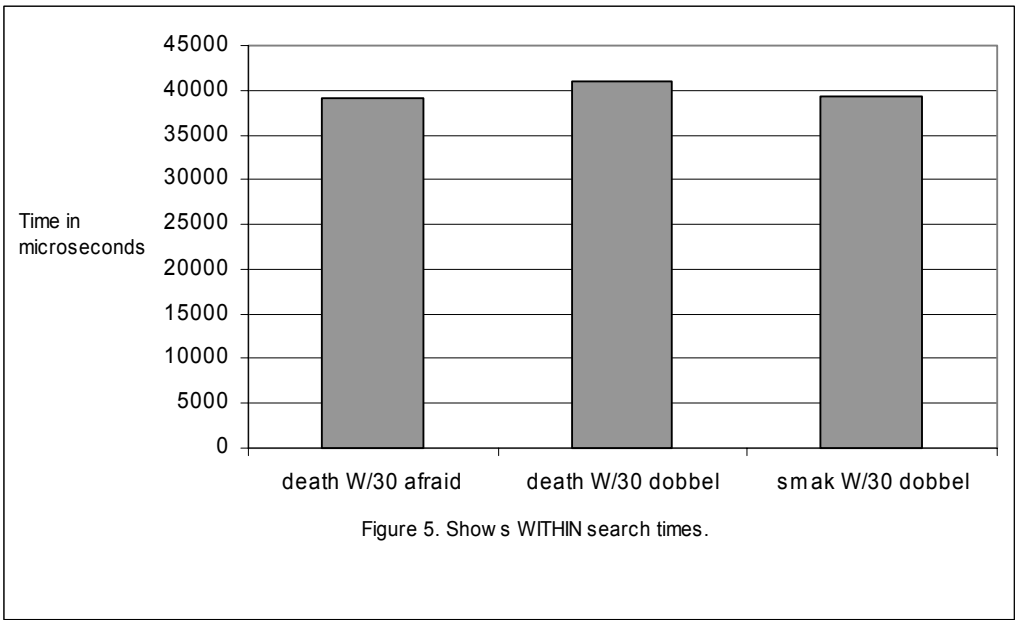

Figure 5. Shows WITHIN search times.

Figure 5 shows a selection of WITHIN searches, like the previously discussed "donald w/10 duck" example. The original text mass is once again The Shining by Stephen King, and the Y-axis yet again represent the amount of microseconds the search took to complete with the project derived implementation. As before, "death" and "afraid" are present in the text, while "dobbel" and "smak" is not. They all take approximately the same time to complete due to the simple fact that this specific implementation tries to find every occurrence of the search data, instead of halting at the first occurrence. This behavior is encouraged by the fact that this can be used within natural language searches in order to construct levels of relevance. As stated before, the problem with WITHIN searches is that they, to an even greater extent than other Boolean searches, make extensive use of metadata in order to resolve their search. The current project derived implementation lacks an efficient way to handle this, and future research is needed to ensure this problem is solved in a gratifying manner. However, the current implementation is still several times faster than the equivalent search performed with the dtSearch program, which noticeable hiccups on such searches.

4.5 Stemming

The notion that stemming is of great use in text analysis is not being questioned here; the question to ask is rather *what* to stem and *when* to stem. The answers to these questions are somewhat dependant on the intended use of the final system, but as this paper only concerns itself with computer forensic settings it will disregard use outside of this environment. This, however, does not mean that the results of these discussions are not valid in other domains as well.

Continuing this line of thought leads to the understanding that the index files themselves should not be stemmed. The reason to stem index files would be to further minimize their size, but the herein gained performance benefit does not in any way compensate the impending possibility of losing valuable data. So, to answer the question about *what* to stem, stemming should only be used when manipulating actual search data. This conclusion also implies the answer to the question about *when* to stem, i.e. stemming should be used in real-time within the actual search. This behavior

minimizes the risk of error, since the operator at hand will have the opportunity to manually filter the unabridged output data, while at the same time utilizing the benefits of stemming.

The stemming method of choice within the project derived implementation is of the Porter Stemming Algorithm derivation. A full definition of this algorithm can be viewed here [25]. As only the search input strings are being stemmed, there will only be an insignificant time penalty when enabling stemming. This follows, first of all, the fact that the actual stemming process is rather efficient and fast in itself, and secondly because the amount of words to stem will be kept at an absolute minimum which complies with the previously discussed tendency to introduce errors. The only occasion when stemming might display any kind of delay is when used in conjunction with natural language searching, where there are large quantities of search input to process. In this case it might be worthwhile to note that, just as one might expect, stemming follows a very linear pattern when confronted with growing amounts of input data. Figure 6 clearly illustrates this pattern.

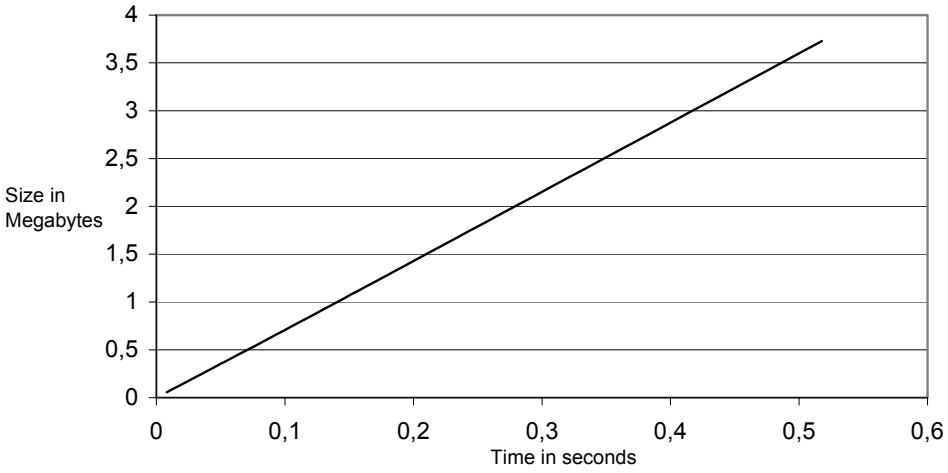

Figure 6. Shows the time/size ratio when stemming.

5 DISCUSSIONS AND FUTURE WORK

The storage capacity of modern computers is growing at an enormous rate, and although most people applaud this development, it obviously increases the workload for computer forensic examiners at a similar pace. As the disposable time for each case cannot be expected to increase, the only imaginable solution is to work faster! This insinuates raising the levels of automation involved in the forensic processes. No forensic examination is quite like any other, and the flexibility, intuition and pure knowledge of a human computer forensic expert cannot, in the present day, be rationalized away. This, however, does not mean that certain aspects of a computer forensic investigation cannot be automated in order to speed up the process. Recent years have seen a growing awareness concerning computer forensics as a legitimate science and a valid addition to the forensic arts as a whole.

This paper concentrates on the investigation of forensic text analysis techniques, and how one might streamline the process of examining large amounts of text. And as observant readers no doubt have already gathered from the previous chapters, there still remains a large amount of work in order to produce a working product. A large part of this work lies in actual implementation; getting the code together into one package, introducing a stable frame-work of error-checking, implementing at least a rudimentary level of user-friendliness and so forth. Another important part in the continuation of this project is to further research the theories of natural language processing which in itself is a very broad and extremely interesting subject. Chapter three discusses the computer forensic adaptations within natural language processing, and further research and implementation is needed in order to paint a conclusive picture of their validity. The positive impact of a fully functional computer forensic natural language processing system cannot be overestimated. This will enable investigators to discover patterns and shapes within large text masses, thus creating a way not only to analysis data *after* a crime is committed, but also to enable the possibility to preemptively discover possibly interesting activities.

As far as the indexing engine is concerned there is some further consideration to be done. The present implementation uses human-readable index files, i.e. it saves the output in ordinary text files. This was done in order to save implementation time and to produce easily viewed index files, which was of great practical value when performing the tests. A fully developed indexing engine, however, does not need this behavior. There will most definitely be a switch to binary format in the index files, as this result in a further minimization of the size of the index files and also helps in speeding up the process of searching.

Continuing the examination of the current indexing engine, a serious shortcoming presents itself; namely the lacking support of proper character encoding. As it is the engine only handles normal printable ASCII characters, which is fine as long as the text to be analyzed is written in English. Unfortunately however, people have been known to use other languages as well. One solution to this problem is to implement support for Unicode. Only then will the system be able to handle more or less all written languages in use today; including the these days highly topical language of Arabic, the linguistic pleasantries of Thai and the ever musical language of Swedish. However, since Unicode contains such an enormous wealth of possible characters, there exists the theoretical problem that this might create hopelessly large index files. Further research is needed in order to establish how much of a problem this is in real-life situations and how to counter the problem. Possible solutions include language filter systems that recognize different languages and automatically separate the internal

structure of the index files. Again, this is of little importance for the completion of the tests done in conjunction with this paper, but if there is to be any hope of this software ever transforming into a functioning application this is a situation that most certainly will have to be resolved.

The use of stemming in conjunction with indexing and searching has been researched for many years. However, its use in relation with computer forensic text analysis has to the best of the authors knowledge never been thoroughly investigated. Stemming does introduce a certain margin of error, a margin of error seemingly rather acceptable within most previous applications. However, within computer forensics the impact of these errors can be pivotal, and thus has to be circumvented while at the same time utilizing the positive properties of stemming. The reasoning throughout this paper shows that this is done by abandoning stemming-enabled indexing and only using stemming in conjunction with post-indexing activities like different kinds of searching.

Different kinds of Boolean searches are often required when dealing with the task of analyzing larger amounts of data. By utilizing the efficiency factor acquired through indexing, it is possible to perform these searches with great speed. There is quite some variation in the amount of time required for different Boolean searches to complete. Most notably the WITHIN search takes some time to resolve, as this specific search needs to access more metadata within the index files than the other search styles. The inclusion of Boolean style searching might help people inexperienced with the use of regular expressions to perform more advanced searches. Most of the Boolean searches can be done with regular expression when used with tools as `grep`, `sed` and `awk`, but the Boolean engine removes the need for the examiner to be fluent in the use of those programs. This is important as computer forensic examinations might not always be performed by computer experts, but rather law enforcement officers trained only in the use of specific forensic tools. Furthermore, the Boolean search system can be used by the natural language engine when constructing relational vectors in order to label different text with degrees of interest.

Generally the aim of natural language processing is to properly understand a given language, an endeavor that has proven more difficult than first expected when such research was started in the late fifties and early sixties. However, natural language processing has proven to be rather successful within specific and well defined domains [21], and computer forensics might be one such domain. The first step in computer forensic language processing is to, given a certain input text, be able to evaluate search data in order to find similarities and otherwise interesting sections. This is not overly difficult to accomplish in a basic manner and should actually not be confused with proper natural language processing. This first step is well understood and forms the theoretical border of this study. The second step introduces somewhat more of a problem, since this step involves giving up the luxury of the input text and having the natural language engine by itself categorizing text based on degrees of interest, and as might be expected, this is where one hits the brick wall. But given a projectile with sufficient momentum any brick wall will crumble.

The implementation of the fuzzy expert system used in the process of identifying misspelled words is not entirely complete, and needs further work to be brought up to the expected standards. This, however, should not be much of a hassle, as the underlying theory is well understood as discussed in chapter three. This is to some extent coupled with phonic text analysis in that it uses similar techniques to resolve the process. The fuzzy expert engine will be able to handle phonic analysis as well, albeit with a modified set of rules. The objective is to create an efficient fuzzy engine with an easily maintained and modifiable rule-set system.

Further file type support is also on the agenda, such as Outlook mail files, Word and PDF documents and different types of compressed files. It would indeed be quite a task to implement support for the entire plethora of imaginable and unimaginable file types, but the most common will most certainly be added in the near future with others to follow later on. Talking about different file types; there sometimes arises a need for exporting the search output into certain formats, like for example Excel charts or SQL-style databases, for further processing. Although this is not a prioritized addition at the moment, it will most likely be addressed later on in the development process.

All present code is implemented with a minimum of fuzz and with a basic command-line interface, which in turns helps in keeping the software fast and time-efficient. Presently the user-friendliness is not up to the standards required by a tool one expects anyone to use, but this should change as the tool matures further during the near future. User-friendliness was neither an objective nor a requirement for the completion of this paper.

There exist several reasons why open source is a preferred paradigm when developing software for computer forensic applications. The most important reason is one that holds true for most computer security related development, namely that if there is one thing that history has taught us, it is that proprietary and hidden security implementations nearly always fail [26]. Such technology instead needs to be properly peer-reviewed and thoroughly examined by the security community in order to stand the test of time [27]. Observe that this discussion only refers to the actual core implementations of the security critical system; the vast amount of code that handles user interfaces, file type support or other additional program features is where the possibilities of financial gain should lie. Some people feel that certain aspects of the solutions and algorithms used in computer forensics should be kept hidden in order to prevent the utilization of possible flaws in these solutions; they feel that if the functionality of a certain solution is known it can more easily be circumvented. This allegation, however, is null and void. Any algorithm or software solution pertaining to computer security whose entire success is dependant on the functionality of said algorithm being kept secret is flawed, and *will* fail.

6 CONCLUSIONS

The Sleuth Kit and Autopsy package, though undoubtedly an excellent computer forensic tool kit, lack certain key features when performing text analysis. When dealing with text analysis Autopsy acts as an interface for the `grep` and `strings` commands, and in addition to this automatically handles the mapping of finds to actual files and folders. Although the `grep` and `strings` commands, coupled with the might of regular expressions, form a powerful tool in their own right, they cannot hope to match the sophistication of high-end commercial search tools. One thing in particular that The Sleuth Kit lack is the ability to handle text analysis based indexing. This means that every time an examiner hits the search button, an entire new search will commence, which in turn can be rather time-consuming if the data at hand is some hundred Gigabytes in size. Seeing that multiple searches are more of a rule than an exception, this is a serious drawback that needs to be addressed. Furthermore, there is a need for expanding the search capabilities to handle such concepts as natural language searching and fuzzy and phonic searches.

Among the various conclusions to be drawn from this study the ones pertaining to indexing hold the most value. Proper indexing techniques are vital to the future success of automated computer forensic text analysis, as they introduce an enormous advantage through their speed and overall efficiency. As the average amount of data in every computer forensic investigation grows, the only hope to handle the situation seems to be through shrewd use of indexing. The tremendous speed advantage obtained with indexing compared with non-index based forensic examinations has been proven conclusively through this study.

One problem found concerning indexing is the actual size of the resulting index files; depending on their desired content they can grow into as much as half the size of the original data. When dealing with several hundreds of Gigabytes of original data, it is not hard to imagine the implied practical conundrum. Using binary index files helps alleviate this problem to a certain extent, along with the added benefit of even further speeding up the process of searching the index files. Another way to mitigate the situation is to apply a noise filter on the search data while indexing. This study shows that the size of the resulting index files can be reduced with more than thirty percent by noise-filtering the indexing process of a normal English text of any greater length. Applying noise filtering does not speed up the consequent searches by any greater margin, its main benefit lies in the reduction of needed storage capacity in order to facilitate the index files.

The use of stemming in the indexing phase does further minimize the size of the resulting index files, but the level of error introduced, slight as it might be, is found to be somewhat unacceptable within a computer forensic domain. By only enabling stemming during post-indexing analysis this problem is avoided, i.e. only the search input strings should ever be stemmed.

A competent Boolean engine helps when performing more advanced searches, because it alleviates the need for any greater knowledge in the use of the somewhat cumbersome regular expressions interface. Used in conjunction with indexing it enables sophisticated searches to be performed with blistering speed.

The inclusion of a fuzzy word engine enables the ability to search for misspelled words, which is of great importance within a computer forensic investigation. The reason for this is that many text masses of high relevance in such investigations, such

as e-mail messages and ICQ sessions, are highly prone to contain misspellings due to many peoples lax attitude towards linguistic correctness while composing such documents. A fuzzy spelling system helps the investigator circumvent this problem to a high extent. The ability to include thesaurus searches also helps in finding allusive text segments, as this incorporates synonyms and longer strings with similar meaning into the search.

The amount of code presently implemented is by no means staggering in any way, only about 1500 lines of code at the moment. This number will no doubt grow at an alarming rate when the work of inserting this code into The Sleuth Kit package commences. As it is the code is not ready for production level usage, among other things more robust error-checking and boundary area testing is needed. The main purpose of an implementation such as this is not to produce commercially viable software with all user-needs being catered for, but rather to implement the core functionality of the theory within the paper in order to test named theory. That being said, the seed for an efficient and ultimately usable piece of software is definitely present, as is indeed the need within the open source community for such an endeavor.

Following the discussions throughout this paper it has become the authors' firm conclusion that the only viable route for computer forensics is open source, extensive peer-reviewing and openly available research. The academic community has to embrace computer forensics to a greater extent for this scheme not to fail, seeing that commercial enterprises, with few exceptions, seem to have a hard time comprehending the importance of this line of thought. The last few years have seen computer forensics play an increasingly important part within the forensic community, and as our society becomes more and more saturated with digital and online devices and applications this will become even more apparent. Computer forensics will face a plethora of exciting challenges during the coming years, and the road to finding their corresponding solutions certainly seems to be an interesting one.

7 REFERENCES

[1] Inman, Rudin. *Principles and Practices of Criminalistics*, 2000.

[2] Warren G. Kruse, Jay Heiser. *Computer forensics: Incident Response Essentials*, 2001.

[3] http://www.dtsearch.com

[4] http://www.sleuthkit.org

[5] http://www.encase.com

[6] http://www.fox-it.com/survey/

[7] Brian Carrier. *Open Source Digital Forensics Tools: The legal Argument*
Available at:
http://www.atstake.com/research/reports/acrobat/atstake_opensource_forensics.pdf

[8] http://www.porcupine.org/forensics/tct.html

[9] http://www.sleuthkit.org/informer/sleuthkit-informer-1.html

[10] http://www.cerias.purdue.edu/homes/carrier/forensics/index.html\#tctutils

[11] http://www.atstake.com/

[12] http://www.sleuthkit.org/autopsy/index.php

[13] http://www.sleuthkit.org/informer/sleuthkit-informer-3.html

[14] http://www.sleuthkit.org/sleuthkit/index.php

[15] Jeffrey, E. F. Friedl. *Mastering Regular Expressions*, 2002.

[16] http://www.cogsci.princeton.edu/~wn/

[17] Christiane Fellbaum et al. *WordNet, An Electronic Lexical Database*, 1998.

[18] http://www.cogsci.princeton.edu/

[19] http://www.princeton.edu/

[20] Porter, M.F. *An algorithm for suffix stripping*, 1980

[21] Stuart Russel, Peter Norvig. *Artificial Intelligence: a Modern Approach*, 1995.

[22] http://www.google.com

[23] C.J. van Rijsbergen *Information Retrival, second edition*, 1979

[24] http://www.stephenking.com

[25] http://www.tartarus.org/~martin/PorterStemmer/def.txt

[26] http://www.microsoft.com

[27] http://www.openbsd.org