

2004

Software for efficient file elimination in computer forensics investigations

Chad Werner Davis
West Virginia University

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

Recommended Citation

Davis, Chad Werner, "Software for efficient file elimination in computer forensics investigations" (2004). *Graduate Theses, Dissertations, and Problem Reports*. 1423.
<https://researchrepository.wvu.edu/etd/1423>

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

**SOFTWARE FOR EFFICIENT FILE ELIMINATION
IN COMPUTER FORENSICS INVESTIGATIONS**

by

Chad Werner Davis

**Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of**

**Master of Science
in
Electrical Engineering**

Approved by

**Dr. Roy S. Nutter, Jr., Committee Chairperson
Dr. Kurishinkal J. Cleetus
Dr. Powsiri Klinkachorn**

Lane Department of Computer Science and Electrical Engineering

**Morgantown, West Virginia
2004**

**Keywords: Computer Forensics, Open Source Software, Hash Filtering,
NSRL, Hard Disk Analysis, MD5, Known File Elimination**

Copyright 2004 Chad Werner Davis

Abstract

SOFTWARE FOR EFFICIENT FILE ELIMINATION IN COMPUTER FORENSICS INVESTIGATIONS

by Chad Werner Davis

Computer forensics investigators, much more than with any other forensic discipline, must process an ever continuing increase of data. Fortunately, computer processing speed has kept pace and new processes are continuously being automated to sort through the voluminous amount of data. There exists an unfulfilled need for a simple, streamlined, standalone public tool for automating the computer forensics analysis process for files on a hard disk drive under investigation. A software tool has been developed to dramatically reduce the number of files that an investigator must individually examine. This tool utilizes the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) database to automatically identify files by comparing hash values of files on the hard drive under investigation to "known good" files (e.g., unaltered application files) and "known bad" files (e.g., exploits). This tool then provides a much smaller list of "unknown" files to be closely examined.

Dedication

This thesis is written in loving memory of Jack Reon Davis, my father who fully supported my education and personal growth, but was taken by cancer too soon. He encouraged me to do my best, as he did for me. Dad, I know you are with me, always. I hope I have made you proud.

Acknowledgements

I would like to begin by first thanking my family, especially my mother, Jeanie, and sister, Debbie, for guiding me and cheering me on throughout my education at West Virginia University. They have always been there for me, making sacrifices along the way to help me achieve my goals. I would also like to thank my girlfriend, Alisa Greathouse, for the support, inspiration, and understanding she has given to me while I finished my research and wrote this thesis.

I must also thank my committee members, Dr. Roy Nutter, Dr. Kurishinkal “Joe” Cleetus, and Dr. Powsiri Klinkachorn for their guidance and support throughout this research endeavor. In addition, special thanks are given to Mr. Douglas White for his insight into the National Institute of Standards and Technology’s National Software Reference Library project.

Above all, I would like to give thanks and praise to God. Through His almighty power and wisdom, He has given to me the direction and strength to overcome all obstacles in life. In His mysterious ways, He has set forth a path for each and every one of us.

Table of Contents

Abstract	ii
Dedication	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	viii
List of Tables	ix
List of Symbols / Nomenclature	x
Chapter 1	
1.1 Introduction to Forensic Science	1
1.2 Introduction to Computer Forensics	2
1.3 Justification for Research	5
1.4 Background: Other Published Work	7
1.5 Other Software Available	8
1.5.1 Known Goods	8
1.5.2 HashKeeper	8
1.5.3 EnCase	9
1.5.4 ILook Investigator	10
1.5.5 Unix-based Tools	10
1.6 Problem Statement	11
1.7 Organization of This Work	12
Chapter 2	
2.1 Foundations of Computer Forensics Investigations	14
2.2 Step 1: Acquire the Evidence	17
2.2.1 Handling the Evidence	19
2.2.1.1 Chain of Custody	19
2.2.1.2 Identifying the Evidence	20
2.2.1.3 Collecting the Evidence	22
2.2.1.4 Transporting the Evidence	22
2.2.1.5 Storing the Evidence	23
2.2.2 Creating a Forensic Backup	24
2.2.3 Documenting the Investigation	25
2.3 Step 2: Authenticate the Evidence	25
2.4 Step 3: Analysis	27
2.5 Preservation and Presentation in Court	30

Chapter 3		
3.1	Introduction to Cryptographic Hash Functions	32
3.2	Properties, Goals and Classifications of Hash Functions	35
3.3	Hash Function Structure	37
3.4	Overview of Popular Hash Functions	39
3.5	The MD5 Hash Function	40
3.5.1	Terminology and Notation	42
3.5.2	MD5 Algorithm Description	43
3.5.3	How Secure is the MD5 Algorithm?	47
3.5.4	Why Choose MD5 Over Other Standard Hashing Algorithms?	50
Chapter 4		
4.1	The Need for a National Software Reference Library	52
4.2	Established Criteria by Law Enforcement	53
4.3	Construction of the National Software Reference Library	54
4.4	The NSRL Reference Data Set (RDS)	56
4.5	Uses of the RDS	58
4.6	Specifications and File Formats of the RDS	61
4.6.1	RDS Data Elements	61
4.6.2	Logical Record Structure of the RDS	64
4.6.3	Physical Record Structure of the RDS	67
4.7	Current Version of the RDS	68
4.8	Effectiveness of the RDS Hash Sets	70
Chapter 5		
5.1	Software Design Methodology	71
5.2	Preparing the NSRL RDS Data	73
5.3	Logging the Investigation	77
5.4	Searching for Files	79
5.5	Hashing Files	80
5.5.1	A Note Regarding Zero-Byte Files	80
5.6	Comparing Hashes with the RDS	81
5.7	Investigative Analysis Views	82
5.7.1	Hex Editor View	83
5.7.2	File Information View	84
5.7.3	File Preview View	85
Chapter 6		
6.1	Success of Research Work	87
6.2	Implications for Computer Forensics Investigators	88
6.3	Software Application Testing	89
6.3.1	Efficiency Tests	90
6.3.2	Effectiveness Tests	90
6.4	Recommendations for Future Work	95
6.5	Final Conclusions	96

Bibliography	97
Appendices	
Appendix A: MD5 Hashing Algorithm Reference Implementation	100
global.h	101
md5.h	102
md5.c	103
mddriver.c	109
Appendix B: Project Source Code	114
mainform.frm	114
hexedit.bas	119
md5file.bas	121
filesearch.bas	122
dbconnect.bas	124

List of Figures

Figure 1: Number of Incidents Reported to the CERT Coordination Center, 1988-2003	4
Figure 2: Damgard/Merkle Iterative Structure for Hash Functions	38
Figure 3: MD5 as a Block-chained Digest Algorithm	41
Figure 4: NSRL RDS Logical Record Relationships	64
Figure 5: Software Design Methodology	71
Figure 6: Sample Execution of the Software Application	72
Figure 7: Sample NSRLFile Table	75
Figure 8: Sample NSRLMfg Table	75
Figure 9: Sample NSRLOS Table	76
Figure 10: Sample NSRLProd Table	76
Figure 11: Relationships Created Between Data Tables	77
Figure 12: Hex Editor View	83
Figure 13: File Information View	85
Figure 14: File Preview Capability	86
Figure 15: Disguised Image File as an Executable File	91
Figure 16: Disguised Executable File as an Image File	92
Figure 17: Legitimate Version of Symantec's Norton Ghost 2002 Utility	93
Figure 18: Cracked Version of Symantec's Norton Ghost 2002 Utility	93
Figure 19: Steganography Within Known Image Files	94

List of Tables

Table 1: Number of Incidents Reported to the CERT Coordination Center, 1988-2003	4
Table 2: Amount of Files Typically Installed by Operating Systems and Applications	6
Table 3: Summary of Selected Hash Functions Based on MD4	51
Table 4: Use of the RDS in Examining Graphics Files	60
Table 5: Data Elements of the NIST NSRL Reference Data Set	63
Table 6: File Record	65
Table 7: Manufacturer Record	65
Table 8: Operating System Record	66
Table 9: Product Record	66
Table 10: RDS Version Record	67
Table 11: Example FILE Data	67
Table 12: Example MANUFACTURER Data	68
Table 13: Example OPERATING SYSTEM Data	68
Table 14: Example PRODUCT Data	68
Table 15: Example RDS VERSION Data	68
Table 16: Current Version of the NSRL RDS	69
Table 17: Effectiveness of the NSRL RDS	70
Table 18: File Attribute Values, Descriptions, and Associated Source Code	85

List of Symbols / Nomenclature

NIST	National Institute of Standards and Technology
NSRL	National Software Reference Library
RDS	Reference Data Set
CRC-32	Cyclic Redundancy Check
MD4	Message Digest version 4
MD5	Message Digest version 5
SHA	Secure Hash Algorithm
SHA-1	Secure Hash Algorithm revision 1
MDC	Modification Detection Codes
MAC	Message Authentication Codes
OWHF	One-Way Hash Function
CRHF	Collision Resistant Hash Function

Chapter 1

1.1 Introduction to Forensic Science

When most people hear the term forensics, they immediately relate to one of the many dramas currently on television that explicitly deal with the adventures of police forensic technicians such as *CSI: Crime Scene Investigation*. For example, an episode may depict police investigators taking numerous photographs while gathering fingerprints and blood and hair samples at the scene of a murder on Fremont Street in Las Vegas. A crime has transpired and the duty of the law enforcement agency is to collect evidence, identify a suspect, and assemble a solid case against the alleged perpetrator. These television dramas have become popular due to America's growing fascination in the field of forensics.

Forensics is the application of science and technology in a civil or criminal investigation to preserve, extract, analyze, and document various items with the aim of producing potentially evidentiary material. "Some crime-history experts place the origins of forensic science in early writings on forensic medicine; a Chinese work titled *The Washing Away of Sins*, published in about 1250, described ways to distinguish between accidental death and murder" [1]. For many common crimes, the methodologies and challenges to solving mysteries are familiar—a time-tested process as old as the legal system itself. The criminal investigator must first ascertain a motive, means, and opportunity for an alleged perpetrator of a crime before the case can be tried in court to obtain a conviction. Throughout the last two centuries, the field of forensics has developed upon its solid scientific foundation and expanded significantly to encompass

many diverse areas including pathology, fingerprint identification, document analysis, ballistics, and even analysis of computer evidence.

1.2 Introduction to Computer Forensics

Rapidly changing technology and expansion in communications and information exchange within corporations and even our own homes has made our world smaller. “America is substantially more invested in information processing and management than manufacturing goods, and this has affected our professional and personal lives” [2]. The market for computer technology is driven by the demand for new and enhanced features. Consequently, functionality and cost, not security, are chief considerations in its design. Computer usage has become ubiquitous and commonplace, and misuse of instant messaging and email communications, file downloading, online banking, and other mundane Internet technologies now pose potential criminal threats to a computer system.

As America has shifted from the production of manufacturing goods to reliance upon the accurate function of information processing systems, modern criminals have also to a large extent made the transition into the cyber world. More and more, these criminals are both utilizing and targeting computer systems. Electronic trails, such as evidence left by criminals who manipulate data, have replaced paper trails. A suspect’s notebook or diary of yesterday may today take the form of a file existing on a floppy disk or hard drive. Crimes involving violence and theft are not impervious to the effects of the information age. The world is gravely cognizant of terrorist attacks in a physical sense such as the plane hijackings which occurred on 9/11. However, we must also

recognize that an equally serious and costly terrorist attack could come from the Internet in the form of a denial of service attack, email bomb, or computer virus.

Computer forensics, one of the newest subsets under forensic investigation, “involves the preservation, identification, extraction, documentation, and interpretation of computer data” [3]. Crimes related to technology may seem novel, but their character remains analogous to other crimes. The motivations of criminals remain consistent; however, their methods do change in relation to advancing technology. New tools are constantly emerging to aid criminals in the commission of their crimes on the Internet. Such utilities steal credit card numbers and other personal information, crack passwords, and deny access to web servers.

In response, the requirements of law enforcement agencies and the regulatory environment are continually evolving. The techniques used by investigators to examine these crimes also change. New security tools are continually being developed to counter the threats posed by modern criminals.

Figure 1 and Table 1 below depict the dramatic rise in the number of incidents reported to the CERT Coordination Center, a major reporting center for Internet security problems, within the past fifteen years [4].

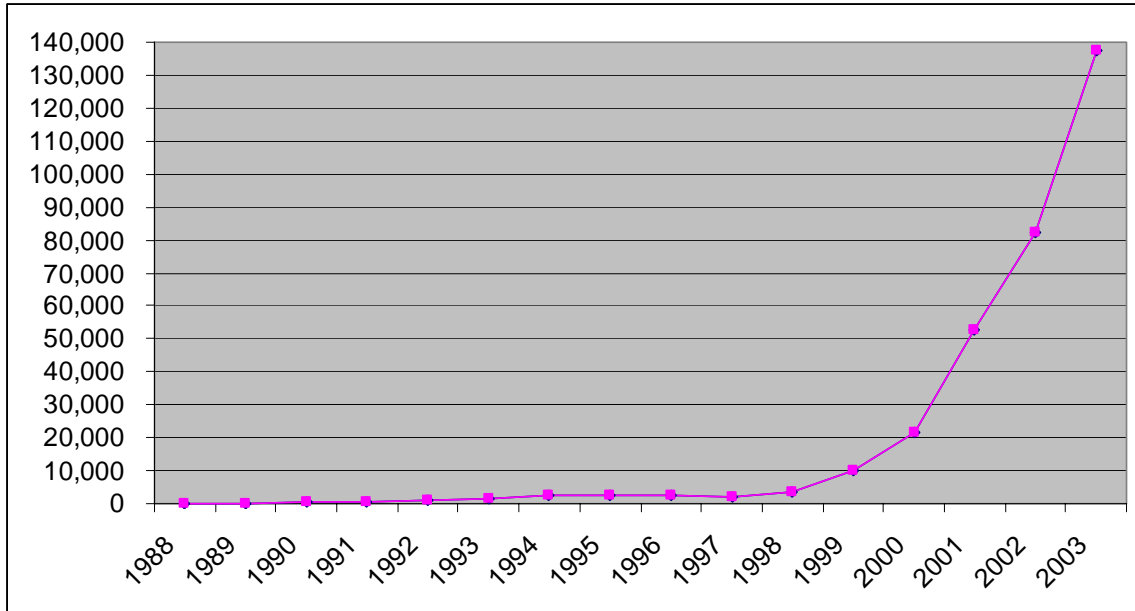


Figure 1: Number of Incidents Reported to the CERT Coordination Center, 1988-2003
(Source: CERT Coordination Center)

Year	1988	1989	1990	1991	1992	1993	1994	1995
Incidents	6	132	252	406	773	1,334	2,340	2,412

Year	1996	1997	1998	1999	2000	2001	2002	2003
Incidents	2,573	2,134	3,734	9,859	21,756	52,658	82,094	137,529

Table 1: Number of Incidents Reported to the CERT Coordination Center, 1988-2003
(Source: CERT Coordination Center)

The field of computer forensics is very demanding. Digital evidence is extremely volatile and special precautions must be taken to preserve its integrity. Explaining the technical aspects of an investigation may prove challenging in a court of law, since many potential jurors are unfamiliar not only with computer forensics, but also with computers themselves.

Computer forensics specialists must couple their own flexibility and creativity when encountering the uncommon with attention to detail in following precise, established methodologies and procedures. Legal precedent offers considerable direction in court cases within common-law countries such as the United States. However, the dynamic nature of computer crime frequently involves untested issues and cyber lawyers must deal with relatively more ambiguity than do many of their fellow legal counterparts.

In the example of the murder on Fremont Street in Las Vegas, the crime scene would be photographed, investigators would search for evidence, and they would acquire various samples such as blood and hair. In computer forensics investigations, evidence is gathered in a similar fashion; however, it is frequently desired that the entire system be recreated in the courtroom. Of course, no one would demand that the prosecution recreate all of Fremont Street at a trial proceeding, but in a computer crime case, that is often the expectation.

Computer forensics investigation is not a suitable field for the complacent. However, it is an exciting career for those highly motivated individuals who desire steady challenge and self-development through their own flexibility and continuous learning.

1.3 Justification for Research

Computer forensics investigators, much more than with any other forensic discipline, must process an ever continuing increase of data. The personal computer revolution in the early 1980s ushered in the introduction of the first hard disk drives.

These 5.25-inch hard disk drives held approximately five to ten Megabytes of data—or roughly 2,500 to 5,000 double-spaced pages of information. At the time, any size over ten Megabytes of storage was viewed as too large for a "personal" computer. By contrast, it is commonplace for today's ordinary home computer systems to feature hard disk drives ranging anywhere between twenty and two hundred Gigabytes of data space—the equivalent of 10,240,000 to 102,400,000 double-spaced pages of information.

“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” [5]. While such an undertaking would be infeasible in forensic pathology, it is the stark reality in computer forensics. Fortunately, computer processing speed has kept pace and new processes are continuously being automated to sort through the voluminous amount of data. Table 2 below shows the number of files typically installed by today's widely used operating systems and applications:

Operating System / Applications	Files Installed
Virgin Windows 98	4,266
Virgin Windows NT 4 Workstation	1,659
Virgin Windows 2000 Professional Edition	5,963
Virgin Windows ME	5,169
Windows 98 + Office 2000	23,464
Windows ME + Office 2000	24,124

Table 2: Amount of Files Typically Installed by Operating Systems and Applications
(Source: National Institute of Standards and Technology)

1.4 Background: Other Published Work

Computer forensics is a relatively new field, and the use of automated processes in the examination of files on a suspect hard drive is even newer. In fact, the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) Reference Data Set (RDS), an extensive database of known file information that can be utilized by these automated processes, was first distributed in 2001.

An exhaustive literature review only produced two relevant research papers. The first is a thesis, written in 2002 by Tye Brown Stallard at the University of California, Davis, which deals with automated text analysis within files to assist in computer forensics investigations [6]. In this paper, Stallard points out that “computer forensics analysts are swamped in evidence because of the large volume of data encountered, the dearth of trained investigators, and the lack of automated techniques to analyze computer crime data.” Stallard makes a short reference to the NIST NSRL, but does not go into depth on the topic, nor implement it within his project.

The second article is a final paper written by Simson L. Garfinkel at the Massachusetts Institute of Technology in 2003 [7]. This short paper proposes the notion for developing a framework by which a database of cryptographic hash values for files could be collected, searched, and replicated via a web interface. These files and corresponding hash values would be considered for evaluation under three categories: known good, known bad, and unknown. The known good files would exist as unmodified system files, applications, etc. The known bad files would exist as Trojan

horses, widely distributed pornography, or other malicious or illicit files. The known good and known bad files would not need examination by the forensic investigator, since they would be automatically disregarded and categorized as evidence. The unknown files would need close investigation by the computer forensics analyst.

1.5 Other Software Available

1.5.1 Known Goods

There are currently a few web services on the Internet that deal with databases of known files and their corresponding hash values. The first example of such a service is Known Goods, located online at <http://www.knowngoods.org> [8]. Brian Wotring of the Shmoo Group created known Goods in 2002 as a way for developers and end users to quickly determine whether or not a file has been modified since it was first installed from its distribution. The hash sets are available either for download or directly from the web service to anyone who wishes to use it free of charge. Unfortunately, the database currently only contains information for executables written for the Linux (i386), FreeBSD (i386), Mac OS X, Mac OS X Server, and Solaris operating systems. The service's website states that it is "not the authoritative source for checksums on files," but simply a tool that can be used to verify a questionable file by comparing its hash value with the one on record in the database of known good files.

1.5.2 HashKeeper

Another online service is HashKeeper, located on the Internet at <http://www.hashkeeper.org> [9]. This service was created in 1997 by the United States

Department of Justice National Drug Intelligence Center for use by any state, local or federal law enforcement entity. HashKeeper is distributed as a run-time Microsoft Access database application that implements the Message Digest 5 (MD5) file signature algorithm to establish hash sets for known files. The application compares those known hash values against the hash values of unknown files on a seized computer system. HashKeeper then categorizes hash values as authenticated (known good), authenticated and notable (known bad), or unauthenticated (unknown) and subsequently groups individual related hash values into hash sets. A forensics investigator can then determine with a degree of statistical certainty that files on a seized hard disk drive matching the database of known hash values do not need to be closely examined. Unfortunately, this online service has been down since May 2002 but their website promises a new and improved service in the near future.

1.5.3 EnCase

There are also integrated suites of computer forensics utilities available to investigators. An example of one of these suites is EnCase, written and distributed by Guidance Software [10]. This utility incorporates over fifteen tools such as disk imaging, deleted file recovery, and analysis of slack space on a disk drive. In addition, EnCase compares known file signatures, or hash values, with suspect files so that investigators can determine whether the alleged perpetrator has modified the data within files in order to hide evidence from detection. EnCase, used by thousands of law-enforcement agencies around the world, can also be purchased for use by educational institutions and

corporations. The drawback for using EnCase is its hefty price tag: \$1,995.00 for education and government institutions, and \$2,495.00 for corporations.

1.5.4 ILook Investigator

Another computer forensics suite is ILook Investigator, which was originally written by Elliot Spencer [11]. Spencer later partnered with the U.S. Internal Revenue Service, Criminal Investigation Division, Electronic Crimes Program and today this comprehensive disk analysis tool is made available at no charge, but only to “qualifying law enforcement agencies throughout the world.” Although this comprehensive forensics suite could prove useful to private industry applications, it is not available to them. ILook Investigator utilizes hash values from both the HashKeeper and NIST NSRL databases. ILook Investigator supports numerous FAT, NTFS, Mac, Linux, SCO, Novell Netware, and CD file systems and their variants.

1.5.5 Unix-based Tools

Individual utilities used for hashing files and verifying their validity to a database of known hash values exist primarily as Unix-based solutions. Two examples of these utilities are The Sleuth Kit [12] and HashDig [13]. The Sleuth Kit is an open source set of separate utilities that may be used together in the forensic investigation of FAT, NTFS, EXTxFS, and FFS file systems. Hfind is a utility within The Sleuth Kit that implements a binary sort algorithm to look up hashes within the NIST NSRL, HashKeeper, and custom hash databases created by the MD5 hashing algorithm. HashDig is an open source utility designed to automate the process of creating MD5 hash values for files on a

suspect computer system, comparing these values with the known values in the NIST NSRL, HashKeeper, KnownGoods, Sun's Solaris Fingerprint Database, or any custom built database, and placing each file in either a category of known files and unknown files. Unfortunately, most of these Unix-based programs require the investigator to write complex scripts and work purely within a command line interface. The generated output is usually created within comma separated text files that require the use of another program such as a spreadsheet application to view the output in an easy to read format.

1.6 Problem Statement

A review of previously published work indicates that although the notion of hash filtering has exploded, there are several concerns with existing software used to compare hash values on a suspect machine with a database of known hash values. These concerns include that the software is either packaged within a forensics suite, too expensive, too hard to use, or otherwise unavailable for use by the general public or most small law enforcement agencies. There exists an unfulfilled need for a simple, streamlined, standalone public tool for automating the computer forensic investigative process for files on a disk.

It is proposed that a software tool be designed to automate the analysis of a hard drive under investigation and thus dramatically reduce the number of files that an investigator must individually examine. This tool will utilize the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) database to automatically identify files by comparing hash values of files to "known

good” files (e.g., unaltered application files) and “known bad” files (e.g., exploits). This tool will provide a much smaller list of files to be closely examined.

1.7 Organization of This Work

This research is organized into six chapters. Chapter 1 introduces the subject matter and context into which this thesis fits. Forensic science, specifically the field of computer forensics is presented along with justifications for the research. Previous work in the form of published articles and other available software provides background information on the subject matter. Chapter 2 presents a general overview of the computer forensics methodologies and the investigative process: acquiring, authenticating, and analyzing evidence. Special emphasis is also placed on proper identification, collection, handling, transportation, storage, and backing up of the evidence, as well as chain of custody, documenting the investigation, and preservation and presentation of evidence in a court of law. Chapter 3 provides an introduction to cryptographic hash functions, including their goals and classifications, and overall design structure. An overview of popular hash functions such as MD4, MD5, and SHA-1 is given. A technical description of the MD5 hash algorithm is presented, with analysis of its security from attack and advantages over other algorithms. Chapter 4 provides information about the need for, and construction of, the NIST NSRL and its RDS. This chapter provides examples of how the RDS can be used in computer forensics investigations. Detailed technical information is given about data elements and record structures of the RDS. Also, the effectiveness of the RDS hash sets is examined. Chapter 5, the core of this research, presents the methodology and design for building a prototype software application

written to substantiate this research. This technical walkthrough details how the software application prepares data from the RDS distribution, searches for files and calculates their corresponding hash values, and then compares those hashes with the RDS. Also, the ability to save logs and view files in a hex editor and other views are demonstrated. Chapter 6 provides analysis and implications of this research, including several tests to ensure the software application is both efficient and effective. Recommendations for future work and final conclusions are also presented.

Chapter 2

2.1 Foundations of Computer Forensics Investigations

Computer systems have proliferated society and our business world. It is not surprising that they are both a tool for committing, and the object of, crime. These crimes are varied and include unauthorized use such as stealing a username and password, creating or releasing a malicious computer program such as a worm or virus, harassment and stalking, identity theft, email abuse, pornography, fraud, and theft of proprietary information and intellectual property. All of these computer-related crimes leave digital tracks. These tracks can provide evidence that an alleged perpetrator did or did not commit the suspected crime.

When computers are suspected of being used to commit a crime, investigations usually include examining log files to see what occurred and searching through gigabytes of data to look for specific keywords related to the crime being investigated. When computers are the object of a crime, the systems that were remotely attacked are examined. This procedure is known as incident response. Remote attacks originated over the Internet are rapidly increasing in both frequency and sophistication as network services become more complex, and more vulnerable. As the sophistication of computer technology increases so does the need to anticipate and guard against a corresponding rise in computer related criminal activity.

Computer forensics investigators must exercise due diligence and care while following strict guidelines and procedures, approaching the computer as evidence from the very start. Each investigation must be treated as if it will eventually end up in a court of law. For example, the CERT Coordination Center has produced a report explaining how the Federal Bureau of Investigation gathers evidence within their investigations [14]:

“Preserve the state of the computer at the time of the incident by making a backup copy of logs, damaged or altered files, and files left by the intruder. If the incident is in progress, activate auditing software and consider implementing a keystroke monitoring program if the system log on the warning banner permits.”

Even if the investigator reasonably believes that the situation does not call for a forensic analysis initially, it is imperative that he or she completely document what steps were taken. It is possible that the investigator will later discover a criminal act was indeed committed, and this evidence may still be defensible if the documentation can demonstrate in court that the investigator initially had no reason to suspect the computer was involved in a crime and it was later discovered during routine troubleshooting. Timely and thorough documentation, as well as proper evidence handling, are keys to an investigation and possible litigation. The evidence must meet the legal standard for admission at trial, but even then the defense will attempt to weaken the incriminating evidence or have it thrown out altogether. “Possible challenges include questions about reliability and irregularities, inconsistencies and vulnerability to manipulation” [15].

Evidence can be discovered on a wide variety of devices and drives. Floppy, zip, and jazz disks, CDs, DVDs, magnetic tapes, hard disk drives, and thumb drives are just a few examples. Incriminating data may exist inside emails, text documents, images, temporary, system, and swap files, and the system's cache. Evidence may even reside within hidden, deleted, formatted, or partially overwritten areas on the storage medium. Recovering and analyzing such data (i.e., what was done to a file and when it was done) can help the investigator understand what the suspect was attempting to do and whether or not he or she is innocent or guilty.

Goals of a computer forensics investigation are not limited to determining whether a breach occurred, and if so, establishing who the offender was, and then successfully prosecuting the offender if the breach involved criminal activity. Forensics may also be used to determine the main cause of an event to ensure it will not happen again. To be successful, the investigator must fully understand the extent of the problem and how to respond to it. If the analysis is not complete and the extent of the intrusion or compromise is not found then the problem will only be compounded.

Although there are newer computer forensic techniques suitable for rapidly changing technology, the basic methodology remains the same. The details will vary depending on the circumstances and the investigator's goals, but the basic methodology can be broken down into three key elements [3]:

1. Acquire the evidence without altering or damaging the original.
2. Authenticate that your recovered evidence is the same as the original.
3. Analyze the evidence without modifying it.

The scope of this research lies within the third stage of a forensics investigation. As a foundation to the reader, each of these three basic elements will now be examined in greater detail.

2.2 Step 1: Acquire the Evidence

Evidence must be obtained without altering or damaging the original data. However, computer forensics involves many unknowns and much uncertainty, and no two investigations are exactly the same. For example, there is no guarantee that the suspect computer can be powered off without the loss or corruption of data, booted off a floppy disk or bootable CD, or successfully mirrored for analysis. The investigator must encounter the unknowns, and carefully think and adapt strict investigative procedures to the particular situation at hand. He or she must make sure to act in ways that can be easily explained later, and diligently document all of his or her actions without fail.

There are disagreements among computer forensic investigators about whether to let a computer continue to run, to pull the power plug from the computer, or to perform the normal shutdown process during an investigation. Some argue that the only way to freeze the computer at its current state, examine a copy of the original data, and maintain the most defensible evidence, is to pull the plug. However, this is not always practical or politically acceptable, especially if the system will be shut down for a long or indefinite period of time.

Others argue that pulling the plug will result in the loss of any data associated with an attack in process, and it may corrupt data on the hard drive. “In some situations, especially Internet intrusions, the evidence may be found only in RAM and disconnecting or turning off the computer before capturing an image of the computer will destroy what little evidence exists” [3]. If the system has hostile code, or malware, running on it, then data may become lost or corrupted if the system is powered down. Unfortunately, the investigator may not immediately be able to tell whether or not such code is running.

However, if an investigator makes a mistake on a live system during an investigation, he or she cannot simply click the undo button; pulling the plug may be valid because it will allow time to prepare an action plan and perform a forensic backup of the suspect media.

Using forensic utilities that reside on a compromised system to examine that system may not yield reliable or accurate results about its true state. A perpetrator may

anticipate a live system investigation and alter some of the files within the computer's operating system. Again, every case is different, and a contingency approach must be applied. Keep in mind that if the investigator is not rigorous right from the beginning, and the case is prosecuted, what's done is done, and there is no way to go back and recover what has been lost or compromised.

2.2.1 Handling the Evidence

As previously stated, the investigator must exercise great care in handling the evidence right from the start. Proper storage and transportation are particularly critical. Otherwise, the evidence could be compromised and the chance for successful prosecution of any resulting case could be lost. This paper will discuss the initial collection of evidence as well as its later surrender to law enforcement or the victim.

2.2.1.1 Chain of Custody

“The chain of custody is a process used to maintain and document the chronological history of the evidence” [16]. This process must track all persons who had custody of and responsibility for the evidence from its initial acquisition until its final disposition. Documentation should also include:

- Agency and case number
- Victim's and/or suspect's name
- Brief description of the item
- Who collected it
- How and where it was collected

- How it was stored and protected in storage
- Each person or entity who subsequently took possession of it
- Why each person or entity was in possession of it
- Dates and times the items were collected, transferred, and returned

It is imperative to maintain the integrity of the evidence and limit access to it. Defense attorneys will look for discrepancies and gaps in the records, seek to show a break in the chain of custody, create reasonable doubt that the evidence was not properly safeguarded, and try to argue the evidence was tampered with. Without the chain of custody, the evidence may not be admissible in court.

Record keeping can consist of either receipt and voucher type forms, or a simpler spreadsheet application. The key is to be thorough and consistent and ensure that no information is missing. It is also beneficial to select an evidence custodian who is available to receive and release evidence and attend to all record keeping.

2.2.1.2 Identifying the Evidence

Every item of evidence must be identified, labeled, counted, and cataloged. Evidence should be collected under dual control, and in large scale investigations a custodian should be assigned to help coordinate and control the effort and ensure that evidence is properly accounted for at all times. Useful tools in this stage include a laptop computer, a portable printer, and a label maker or even handwritten tags. The evidence custodian should complete the evidence log and print labels for identification of the

evidence. Electronic logs, forms, and reports can be programmed using software to cross reference and automatically fill in identical information, reducing errors and saving time. This software often includes header information that will automatically appear on each form associated with a particular case. Each label should include [3]:

- The case number
- A brief description
- The investigator's and/or custodian's signature
- The date and time the evidence was collected

The investigator should also photograph the entire crime scene to document the environment. Pictures should also be taken of both the front and the back of the suspect computer, including a picture of the screen, while it is still connected to its cables, if possible. Serial numbers should also be photographed and logged. The condition and state (on, off, screen locked, etc.) of the computer system upon arrival should also be documented.

All evidence and information pertaining to a particular incident, including photographs, storage media, papers, reports, etc., should be stored together in a closeable file folder. The folder should be clearly labeled with the header information (e.g., case or incident number, location, brief description, etc.).

2.2.1.3 Collecting the Evidence

Successful conviction depends upon collecting complete, clear, accurate, convincing, and admissible proof that the accused person is guilty. It is important to be thorough in collecting data. Evidence, such as media files or scraps of paper, that is left behind at the scene may not be available later. This is especially true for log files, which may be routinely overwritten in short periods of time (even minutes), depending on the system producing the logs. Similarly, Internet Service Providers usually keep logs for thirty days or less as a manner of standard practice due to the high storage costs and low benefits involved, and the investigator must act quickly to preserve the evidence.

There are many sources of evidence. Swap files, which are spaces in the hard disk set aside by the operating system to be temporarily accessed when more memory is needed, may include recently copied files and passwords. Temporary files, which are created by Windows in case the operating system crashes, include information about open files. The system registry may contain information about what hardware is attached, user information such as recently browsed web pages, and software installation information such as serial numbers and passwords. Even deleted files, which actually remain on the hard drive for a period of time until they are overwritten, can produce potentially incriminating evidence. Other sources include network backups and emails.

2.2.1.4 Transporting the Evidence

Care must also be taken in the transportation of electronic evidence. For example, hard drives can be damaged if they come into contact with magnetic fields or if the read-

write heads come in contact with the platter; in both cases, evidence can be lost because data can no longer be read off the disk. Laptops and personal digital assistants can also be easily damaged if not handled properly. Packaging used to protect the evidence should be static-free to prevent damage.

When the packaging is closed, it should be closed with a tamper-evident seal, and a signature of someone authorized to open it should be written across the seal. Doing so will indicate if someone other than the authorized person opens it later. If, at some point during the investigation it becomes necessary to open the sealed container, document the following information [3]:

- Whether the initial seal was intact
- Why it was necessary to unseal the container
- Dates and times the evidence was both removed from and returned to storage
- Who had custody of the evidence
- What was done to the evidence

Reseal the evidence inside a second container with a new label with signature, so that the original broken seal is preserved, and return it to storage.

2.2.1.5 Storing the Evidence

Both the physical aspect and integrity of electronic evidence must continue to be protected in the storage phase. Evidence must be stored in an environment that is cool,

clean, and dry. It must be in sealed containers, and in a secure area with limited access which is controlled and logged by a designated custodian.

2.2.2 Creating a Forensic Backup

The forensic analysis should always be performed on an exact bit-for-bit (or bit stream) replica of the original media, if possible, and not the original storage medium. A bit stream image is different than a standard backup because it copies deleted files and the other parts of a hard drive that a computer forensics investigator would want to examine for evidence. Examining a copy will help protect the original data or evidence. If a mistake is made and the data being analyzed is damaged, the copy can be erased and the original image can be restored.

Many forensic investigators recommend making two backups of the original drive—backing up the original drive to a hard drive, and using a tape drive to create a second copy, using the second hard drive as the original this time (preferring to use the original drive as little as possible). For example, a forensic drive cloning utility such as SafeBack, available online at <http://www.forensics-intl.com/safeback.html>, can be used to make the first original to second hard drive copy; this is generally the fastest and most reliable way to collect and back up the original evidence [17]. The second drive-to-tape copy is useful during the analysis for archiving and restoring the image as needed. The investigator should make a file signature (MD5 or SHA-1 hash value) of the newly created drive images before beginning the analysis and document it in his or her notes.

2.2.3 Documenting the Investigation

Without proper and extensive documentation of the forensic investigation methodologies used and findings of the investigation it is nearly impossible to successfully present and defend the findings in court. This is true, even in cases where the investigator is very skilled technically. If the investigator lacks the necessary documentation skills, it is imperative that he or she partner with someone who has them. This person must be diligent to accurately and thoroughly document the investigation process in its entirety, at each step along the way.

Documentation must include what actions were taken and why, and be detailed with the software and version numbers of the software evidence, collection tools, and methods used to collect and analyze the evidence. The investigator's actions will be challenged by the defense, and must be upheld as "reasonable." Thorough and detailed notes will serve as a written record and are invaluable. The investigator must never leave these details to memory, especially since the case may not go to trial for some time, perhaps several years.

2.3 Step 2: Authenticate the Evidence

The investigator must be able to authenticate that the evidence collected in the investigation is the same as the data left behind by the criminal. This is a challenge for many reasons. For example, evidence can be damaged over time by unfavorable environmental conditions such as adverse temperatures, moisture, mold, and dust. The investigator must be able to prove that the chain of custody and other rules for handling

evidence where properly adhered to and that no unanticipated or introduced changes occurred to assure the jury of its integrity.

If possible, create a hash of the entire drive and the individual files before performing any analysis. Computer forensic specialists have proven the effectiveness of cryptographic hashing algorithms as a way to verify the integrity of a sequence of data bits. These algorithms verify the contents of the sequence have not been changed since its collection.

“Hashing can authenticate electronic data and the software used to store and maintain it. Two files with exactly the same bit patterns should hash to the same code using the same hashing algorithm. If a hash for a file stays the same, there is only an extremely minute probability that the file has been changed. On the other hand, if the hashes for the files do not match, then the files are not the same.” [18]

Two algorithms commonly accepted for this purpose are MD5 (Message Digest version 5) which creates a 128-bit signature, and SHA-1 (Secure Hash Algorithm) which creates a 160-bit signature. MD5 is discussed in greater detail within the next chapter. Increasingly, applications such as Tripwire, available online at www.tripwire.com [19], are using multiple hash algorithms, so that if an attack is discovered against one algorithm in the future, the data from the other algorithm will remain valid.

Timestamping the evidence can show that it did, in fact, exist at a particular point in time. This is done by using cryptographic software, such as MD5 or SHA-1, to calculate a hash value that serves as a digital fingerprint or signature. Creating and recording a hash value at the time the data is initially collected will allow the investigator to prove that the copies of the data used in the examination are identical to the original.

2.4 Step 3: Analysis

The investigator must analyze all data that might possibly be relevant without modifying or damaging it, and continue to carefully preserve the evidence during this phase. Once a forensic backup or bit stream copy of the original media has been made, it should be used for all analysis. “The investigator should provide an opinion of the system layout, the file structures discovered, any discovered data and authorship information, any attempts to hide, delete, protect, or encrypt information, and anything else that has been discovered and appears to be relevant to the case” [20].

Similarly, the chain of custody must continue to be preserved via detailed documentation whenever evidence is removed or returned to the secure storage cabinet or custodian as previously discussed. The basic and overriding principle is to not compromise the original evidence by altering or damaging it. The investigator must also be careful to operate within legal boundaries and not go beyond his or her own knowledge without seeking qualified expert assistance as needed.

Forensic investigators also disagree about whether to conduct the analysis within a command-line operating system like DOS, or a graphical system like Windows, although the trend is toward Windows. In either case, the investigator will need to be proficient with a variety of program tools, since no one tool will do everything required during the analysis.

The analysis should begin by first examining the partition table on the suspect drive. It is important to document this information too, and it will also help determine what software tools are supported and therefore can be utilized. Next, the investigator should print a directory listing, including subdirectories, or save it to a file. The file can then be opened into a spreadsheet or with a viewer and analyzed to look for specific data.

What is, or is not, found on the computer may give the investigator some clue as to the suspect's prowess. If the investigator discovers complex programs such as a steganography utility (a tool used for secretly hiding data within other files), he or she should be on the lookout for advanced attempts to conceal data. On the other hand, finding only standard software does not mean the investigator can afford to relax his or her guard.

The investigator can use a hex editor or a forensic program to view the master boot record and the boot sector, look for bad clusters, and view them in hexadecimal format. These utilities allow the investigator to read the raw information off of the storage medium in both hexadecimal and ASCII format. The investigator should record

the cluster size and view the File Allocation Table (in the case of DOS/Windows 9x), or Master File Table (in the case of Windows NT/2000/XP). The investigator should also determine if any data is hidden in the bad blocks, especially if there is reason to believe the suspect is a more skilled computer user, and hunt for keywords related to the case if the hex editor or forensic program has a search capability.

Deleted files can be recovered manually using a hex editor, but this is slow and tedious. Kruse and Heiser explain what happens to deleted files within the Windows environment [3]:

“When a file is deleted in Windows, the first character of the directory entry is changed to a sigma character, the hex value of E5. This indicates to the operating system that this directory should not be displayed because the file has been deleted. The entries in the File Allocation Table assigned to the deleted files are changed to zero, indicating that the sectors they point to are unused and available to the operating system for data storage. The operating system does not do anything to the actual data until another file happens to be saved at the same location, which is why the investigator may be able to find incriminating data that the suspect thought he or she had deleted.”

Automated file retrieval software includes Norton UnErase, and Runtime's Software GetDataBack. If the file is fragmented, which is commonly the case; the investigator must manually chain clusters together to make a complete file. Unallocated and slack space, the area on a disk between the end of a file and the end of the cluster that

the file occupies, should also be checked for residual data. This can be done using software tools specifically designed for this purpose.

Save copies of the evidence on the hard disk of the analysis workstation and adjust the formatting as needed for legibility, presentation, and reporting purposes. This will change the saved file's properties, but remember: this is an electric transcript and not the actual evidence. Copy only the relevant, incriminating portions of a file. Document the logical position (page, row, and paragraph) where the data was found and where on the drive the data was recovered (cylinder, head, and sector of the physical drive). The investigator may also be called upon to unzip the retrieved files, searching for and attempting to crack passwords. All removable media collected must all be analyzed as well.

2.5 Preservation and Presentation in Court

Computer evidence is very fragile and it is susceptible to alteration or erasure. As stated before, the investigator must accurately and thoroughly document the chain of custody that accounts for the evidence at all times. He or she must all also store the evidence in a way that it will not be damaged or tampered with in any way. Otherwise, the evidence may not be admissible in court, and the case will be compromised.

Presentation in a court of law is one of the most critical steps in the investigator's case. The investigator must be able to explain to a judge and jury what steps were performed and why the actions he or she took were reasonable. Carefully, completely,

and consistently following the guidelines discussed in this chapter during each phase of the investigation will go a very long way toward a successful presentation in court.

The fundamental process of computer forensics is to acquire, authenticate, and analyze the evidence of an investigation. A balance of strict and disciplined adherence to the rigorous standard procedures of evidence collection and custody, combined with flexibility and out-of-the-box thinking in locating and analyzing the evidence are required for a successful investigation.

Chapter 3

3.1 Introduction to Cryptographic Hash Functions

The word hash means to “chop into small pieces” [21]. Cryptographic hash functions are algorithms used in computer programming to create identifying values of fixed length for data of arbitrary length either for accessing the data or for security purposes such as data integrity and message authentication. These functions impose a mathematical function (or a series of functions) on an input sequence of bits, such as a text string, file, etc., and generating as output a value produced from the algorithm’s influence on those data bits.

“Hashing algorithms fall within the realm of error detection techniques” [22]. Broadly speaking, the algorithm enables a receiver to determine if a message that has been transmitted through a noisy, error-producing channel has been corrupted en route. The receiver computes a hash value that is a function of the received message, and compares it to the hash value of the original message. If the two hash values match, then the message was received as intended; otherwise, the message has been changed.

From a computer forensics standpoint, hashing is an excellent method used to authenticate a sequence of data bits and ensure the immutability of a file’s original content. In this case “the hash value serves as a compact representative image (also referred to as a digital fingerprint, digital signature, or message digest) of an input string, and can be used as if it were uniquely identifiable with that string” [4].

The cryptographic hash function is used to detect any change in the contents of a file, either an accidental change or a change that was made on purpose. By generating a value which serves as a “benchmark” or “fingerprint” for a file, the investigator can be sure that the file has not changed if it is the same as the “known” hash value of the original content.

Message digests are identical as long as they are generated for the same identical file. However, if even one single bit is added (or otherwise changed) in the file, the message digest is not only different, it is entirely different. The hash function ensures that if a single bit of the input is altered in any way a bitwise inversion of roughly half of the bits in the resulting cryptographic result. This is also known as the “avalanche effect” [24]. The smallest amount of change will force the digital signature verification process to fail, since each bit of the hash value depends upon each and every bit of the input. A changed hash value does not tell you how different the changed file is, or what the differences are; it just tells you that there is or is not a difference.

An example of how hash values can be applied to data integrity is given by Menezes and Oorschot, and Vanstone [23]:

“The hash-value corresponding to a particular message x is computed at time T_1 . The integrity of this hash-value (but not the message itself) is protected in some manner. At a subsequent time T_2 , the following test is carried out to determine whether the message has been altered, i.e., whether a message x' is the same as the original message. The hash-value of x' is computed and compared to

the protected hash-value; if they are equal, one accepts that the inputs are also equal, and thus that the message has not been altered. The problem of preserving the integrity of a potentially large message is thus reduced to that of a small fixed-size hash-value.”

Hash functions offer several advantages over encryption. Encryption is slower, and encryption hardware is expensive and optimized to large data. “A digital signature or integrity check can be computed by applying cryptographic processing to the document’s hash value, which is small compared to the document itself. Also, a message digest can be made public without revealing the contents of the document from which it is derived. This is important in digital timestamping where, using hash functions, one can get a document timestamped to establish that it existed on a certain date without revealing its contents to the timestamping service” [25]. This is useful, for example, in the case of copyright disputes.

Cryptographic hash codes can be extended for the use of determining the identity of a file. “By computing the hash of a suspect file and then looking up that hash in a database, it is possible to determine if that suspect file is a copy of a file that has previously been evaluated, characterized, and registered” [7]. CDROMs of hash codes from a wide variety of commercially distributed software packages are now available typically for use by law enforcement agencies in computer forensic investigations.

3.2 Properties, Goals and Classifications of Hash Functions

“At the highest level, hash functions may be split into two classes: unkeyed hash functions whose specification dictates a single input parameter (a message); and keyed hash functions, whose specification dictates two distinct inputs, a message and a secret key” [23]. A more functional classification includes two sub-classifications: Modification Detection Codes (MDCs), which are unkeyed, and Message Authentication Codes (MACs), which are keyed. The algorithmic specifications of cryptographic hash functions are generally said to be public knowledge or unkeyed. Therefore, only MDCs will be elaborated here.

A hash function (in the unrestricted sense) is a function h which has, as a minimum, the following two properties [23]:

1. Compression – h maps an input x of arbitrary finite bitlength, to an output $h(x)$ of fixed bitlength n .
2. Ease of computation – given h and an input x , $h(x)$ is easy to compute.

When employed in cryptography, the hash functions are usually chosen to have some additional properties. These requirements for a cryptographic hash function h include [25]:

1. $h(x)$ is one-way.
2. $h(x)$ is collision-free.

MDCs are mathematical values, created by a cryptographic hashing algorithm that is used to test a given file to verify that the data contained in the file has not been inadvertently or maliciously altered. MDCs may be further classified, to include one-way hash functions (OWHFs) and collision resistant hash functions (CRHFs).

Hash functions are used to condense a string of characters into a shorter fixed-length value that represents the original string. Hashing is a one-way operation if it is hard to invert; the ideal hash function or output can not feasibly be determined by analyzing the hashed values or inputs (preimage resistance), and it is computationally infeasible to find any second input which has the same output as any specified input (2^{nd} -preimage resistance).

“A hash function h maps bit-strings of arbitrary finite length to strings of fixed length, say n bits. For a domain D and range R with $h: D \rightarrow R$ and $|D| > |R|$, the function is many-to-one, implying that the existence of collisions (pairs of inputs with identical output) is unavoidable. Indeed, restricting h to a domain of t -bit inputs ($t > n$), if h were “random” in the sense that all outputs were essentially equiprobable, then about $2^{(t-n)}$ inputs would map to each output, and two randomly chosen inputs would yield the same output with probability $2^{(-n)}$ (independent of t)” [23].

Therefore, a good hash function should not produce the same has value from two different inputs; it should be collision-free or at least collision resistant. Since the existence of collisions is guaranteed in many-to-one mappings, the unique hash value

should be uniquely identifiable with a single input in practice, and collisions should be computationally difficult to find (essentially never occurring in practice). “The term collision-resistant hash function is sometimes used to describe a hash function that possesses all three of the properties described here and it is what most people have in mind when talking about hash functions in general” [26].

“A cryptographic hash function or checksum should, in practice, guarantee that any tampering with a file will result in a different checksum, and that in practice no one will be able to come up with any different file which also produces the same original checksum. They simply prevent anyone from changing a file in any way without leaving evidence that they have done so (in the form of a changed checksum)” [27].

3.3 Hash Function Structure

Ralph Merkle and Ivan Damgard made a significant contribution to cryptographic hash function design by proving that a collision resistant hash function can be constructed using a collision resistant compression function. A compression function takes a fixed-length input and returns a shorter, fixed length output. Given a compression function, a hash function can be defined by an iterative application of the compression function until the entire message has been processed. The computation of the hash value for some message depends on what is called a chaining variable.

“At the start of hashing, this chaining variable has a fixed initial value which is specified as part of the algorithm. In the process, a message of arbitrary length is broken

into blocks whose length depends on the compression function, and “padded” (for security reasons) so the size of the message is a multiple of the block size. The blocks are then processed sequentially by the compression function, taking as input the result of the chaining variable so far and the current message block. This compression function continues recursively until the entire message (and any additional padding specified by the algorithm) has been used. The chaining variable is updated in a suitably complex way under the action and influence of the current message block being hashed. The final output value of the chaining variable is the hash value corresponding to the entire message” [26].

The Merkle-Damgard construction is used by many of the popular hash functions, including MD5 and SHA-1, which will be discussed later in this paper. However, a concern with this construction is that finding a collision resistant function can be difficult. A schematic model of the Merkle-Damgard iterative structure for hash functions is shown below in Figure 2.

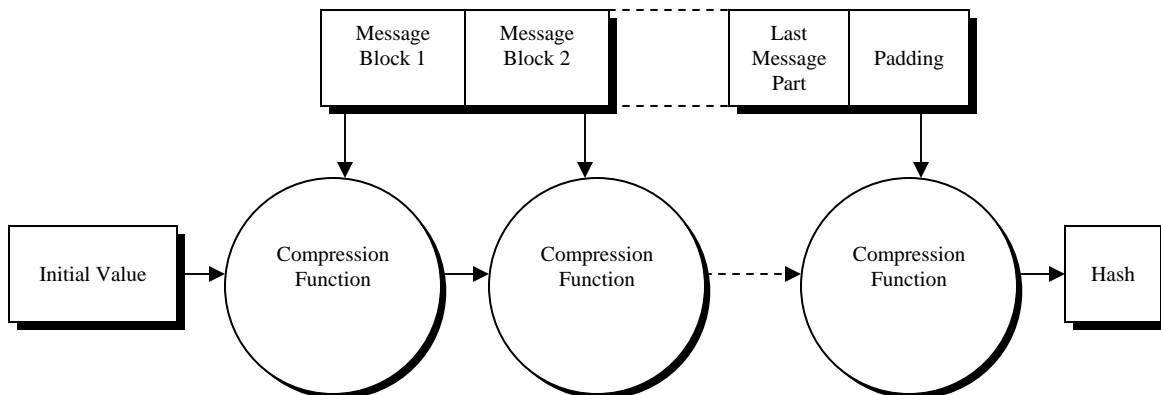


Figure 2: Damgard/Merkle Iterative Structure for Hash Functions
(Source: RSA Laboratories)

3.4 Overview of Popular Hash Functions

Well known hash functions include CRC-32, MD4, MD5, and SHA-1. These widely used hash functions will be briefly overviewed in this section.

The cyclic redundancy check has been an integral part of the computer industry for quite some time. The CRC-32 algorithm, described in ISO 3309, calculates a resulting checksum based that is four octets, or 32 bits, in length [28]. CRC-32 is neither keyed nor collision-proof. Thus, the use of this checksum for message integrity and validation is not recommended.

MD stands for message digest. MD4 and MD5 are algorithms used to verify data integrity through the creation of a 128-bit message digest or fingerprint from data input of any length that is claimed to be unique to that specific data. Both were developed by Professor Ronald L. Rivest of MIT and are optimized for 32-bit machines.

MD4 was developed in 1990. MD5 is an improved version of MD4; it was developed in 1992 in conjunction with RSA Data Security, Inc. after a successful attack was made on MD4. It uses four, more complex, rounds of 16 steps compared to three rounds in MD4. MD5 is slower, but offers more assurance of data security than MD4. MD5 will be discussed in greater detail within the next section. The widespread popularity of the MD family of hash functions is a testament to their innovative and successful design.

SHA stands for secure hash algorithm. SHA-1 was designed by the National Institute of Standards and Technology (NIST) and National Security Agency (NSA). It produces 160-bit hash values and is generally considered to be the preferred hash algorithm. It is more resistant to cryptanalysis than MD5 and uses 20 steps in each of the four rounds. However, it is somewhat slower in execution than MD5.

3.5 The MD5 Hash Function

The MD5 Message-Digest Algorithm was developed in April 1992 by Professor Ronald L. Rivest at the MIT Laboratory for Computer Science in conjunction with RSA Data Security, Inc. This algorithm has become widely adopted by computer security investigators and law enforcement and remains one of the most used hash functions in the world today. In fact, MD5 has enjoyed widespread use within peer-to-peer file-sharing networks, where the ability to download a single file from several sources at once is essentially dependent upon hashing to identify that the files on different machines are identical, regardless of what they have been named. Furthermore, MD5 hash values for downloadable files on many public websites are often posted so that the integrity of a file can be verified once it has been downloaded.

As previously stated, MD5 is basically a way to verify data integrity. The algorithm computes a digest of the entire data of the message, which is used for authentication [29]:

“MD5 takes as input a message of arbitrary length and produces as output a 128-bit message digest, or fingerprint, of the input represented by a 16-digit

hexadecimal number. It is conjectured that it is computationally infeasible to produce two messages having the same message digest [on the order of 2^{64} operations], or to produce any message having a given pre-specified target message digest [on the order of 2^{128} operations].”

“MD5 is a block-chained digest algorithm, computed over data in phases of 512-byte blocks organized as little-endian 32-bit words. Each 512-byte block is digested in 4 phases. Each phase consists of 16 basic steps based on each of 4 logical functions, for a total of 64 basic steps. The first block is processed with an initial seed, resulting in a digest that becomes the seed for the next block. In general, each basic step depends on the output of the prior step. When the last block is computed, its digest is the digest for the entire stream. This chained seeding prohibits parallel processing of the blocks” [30].

This is shown below in Figure 3:

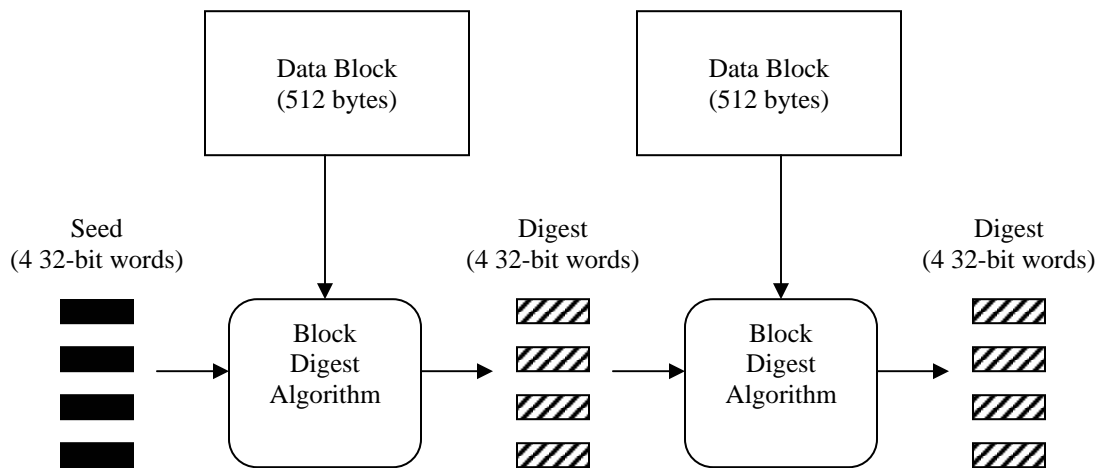


Figure 3: MD5 as a Block-chained Digest Algorithm
(Source: Joseph D. Touch / USC Information Sciences Institute)

The definitive paper on the MD5 hashing algorithm is RFC-1321, written by Rivest. The contents of that request for comments paper relating to the technical workings of the MD5 hashing algorithm have been reproduced within the next two subsections to provide the reader with an understanding of the algorithm. RFC-1321 includes a reference implementation in the C programming language, which can be found in Appendix A of this text.

3.5.1 Terminology and Notation

In this document a ‘word’ is a 32-bit quantity and a ‘byte’ is an eight-bit quantity. A sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of eight bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of four bytes is interpreted as a word with the low-order (least significant) byte given first.

Let the symbol "+" denote addition of words (i.e., modulo- 2^{32} addition). Let $X \lll s$ denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions. Let $\text{not}(X)$ denote the bit-wise complement of X, and let $X | Y$ denote the bit-wise OR of X and Y. Let $X \text{ xor } Y$ denote the bit-wise XOR of X and Y, and let XY denote the bit-wise AND of X and Y.

3.5.2 MD5 Algorithm Description

We begin by supposing that we have a b -bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

$$m_0 m_1 \dots m_{b-1}$$

The following five steps are performed to compute the message digest of the message.

Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

Step 2. Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^{64} , then only the low-order 64 bits of b are used.

(These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.) At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let $M[0 \dots N-1]$ denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest.

Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

Step 4. Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

$$F(X,Y,Z) = XY \mid \text{not}(X) Z$$

$$G(X,Y,Z) = XZ \mid Y \text{ not}(Z)$$

$$H(X,Y,Z) = X \text{ xor } Y \text{ xor } Z$$

$$I(X,Y,Z) = Y \text{ xor } (X \vee \text{not}(Z))$$

In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using \vee instead of \wedge since XY and $\neg(X)Z$ will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, then each bit of $F(X,Y,Z)$ will be independent and unbiased.

The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of $G(X,Y,Z)$, $H(X,Y,Z)$, and $I(X,Y,Z)$ will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

This step uses a 64-element table $T[1 \dots 64]$ constructed from the sine function. Let $T[i]$ denote the i -th element of the table, which is equal to the integer part of 4294967296 times $\text{abs}(\sin(i))$, where I is in radians. The elements of the table are given in the appendix.

Do the following:

```
/* Process each 16-word block. */
For i = 0 to N/16-1 do

/* Copy block i into X. */
For j = 0 to 15 do
  Set X[j] to M[i*16+j].
end /* of loop on j */
```



```

/* Save A as AA, B as BB, C as CC, and D as DD. */
AA = A
BB = B
CC = C
DD = D

/* Round 1. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */

[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

/* Round 2. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */

[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

/* Round 3. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */

[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

/* Round 4. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */

[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

```

```

/* Then perform the following additions. (That is increment each
   of the four registers by the value it had before this block
   was started.) */

A = A + AA
B = B + BB
C = C + CC
D = D + DD

end /* of loop on i */

```

Step 5. Output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.”

3.5.3 How Secure is the MD5 Algorithm?

The MD5 hashing algorithm takes as input a message of arbitrary length and produces as output a 128-bit “fingerprint” or “message digest” of the input. In RFC 1321, Rivest stated, “it is conjectured that it is computationally infeasible to produce two messages having the same message digest [on the order of 2^{64} operations], or to produce any message having a given pre-specified target message digest [on the order of 2^{128} operations” [29]. A 128-bit hash value is so unique that there are 2^{128} or $3.4028e+38$ different possible MD5 hash values, a value so vast when compared to the total number of electronic files that have been created during the course of human history. Thus, hash representations can be treated as “fingerprints” or “signatures” for files: so far no two files have ever been found that have the same MD5 code.

According to Simson Garfinkel's writings on the MD5 hashing algorithm [27]:

“Mathematically, its easy to see that billions and billions of messages have the same MD5 result, because the MD5 function produces only 128 bits of output—just sixteen 8-bit digits. So theoretically, if a message is only 17 characters in length, there would probably be 256 different messages that have the same MD5 code [checksum] (because there would be 256 more possible messages than possible MD5 codes, which means that some codes would have to be reused).

So why does MD5 seem so secure? Because 128-bits allows you to have $2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456$ different possible MD5 codes. That is a number that is billions and billions of times larger than the total number of documents that will ever be created by the human race for the next thousand years. So even though many different documents have the same MD5 code, human beings aren't likely to find many of them in their lifetimes.”

Den Boer and Bosselaers have made the first important advance in the cryptanalysis of the MD5 hashing algorithm by discovering what are termed as pseudo-collisions for the compression function of MD5 [25] [26]:

“A pseudo-collision for the compression function is exemplified by fixing the value of some message block and finding two distinct values for the chaining variable that provide the same output. While the existence of pseudo-collisions is significant on an analytical level, it is of less practical importance. Recall that

only a single chaining variable is used during hashing and so the behavior of two related chaining variables is not directly relevant. Instead, it would be more significant if we could identify the value of a single chaining variable for which two different message blocks produce the same output from the compression function. Such an occurrence would have obvious implications for the collision-resistant property we often desire of a hash function. If the value of the chaining variable involved were not the same as the initial value (as provided in the algorithm specifications) then such an occurrence would be termed a collision for the compression function. If, however, we could identify two message blocks which provide a collision when the pre-specified initial value is used, then we would have full collisions for the hash function.”

In 1996 it was announced that Hans Dobbertin’s research showed MD5 to be vulnerable to collision search attacks. “While no collisions for MD5 have yet been found, Dobbertin demonstrated collisions for the MD5 compression function in around 10 hours on a PC” [23]. Since the MD5 hash algorithm was specified in 1992, computational power has increased exponentially, and some would argue it is no longer computationally infeasible to intentionally duplicate an MD5 hash. “Van Oorschot and Wiener have considered a brute-force search for collisions in hash functions, and they estimate a collision search machine designed specifically for MD5 (costing \$10 million in 1994) could find a collision for MD5 in 24 days on average. The general techniques can be applied to other hash functions” [25].

“Existing signatures that were generated using MD5 are likely to remain safe from compromise since it seems that current techniques used to cryptanalyze MD5 do not offer any advantage in finding a second preimage. Existing signatures should not be considered as being at risk of compromise at this point. Likewise the random-looking appearance of the output from MD5 and the property of being one-way are not considered to be seriously in question” [26].

So, how safe is MD5? No one knows for certain. However, there are sound reasons to think that MD5 is still quite safe for most purposes, and currently impossible to defeat in practice. While we do not know that some mathematician will come up with a systematic method of modifying files without changing their MD5 hash values, we do know that so far no one has published such a technique. MD5 has resisted a considerable amount of professional analysis by cryptographers attempting to see if it can be defeated.

3.5.4 Why Choose MD5 Over Other Standard Hashing Algorithms?

Rivest states that “although MD5 is slightly slower than MD4, it is a strengthened algorithm and more conservative in design. MD5 was designed because it was felt that MD4 was “at the edge” in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security” [29].

While MD4 is considered obsolete due to its ease of cryptanalytic attack, MD5 is still considered to be safe. SHA-1 is a revision of the Secure Hash Algorithm (SHA),

which was revised due to an unreported fault in the original implementation. Even though SHA-1 now appears to be stronger cryptographically, MD5 can still be considered for use in hashed Modification Detection Codes (MDCs) for applications where the superior performance of MD5 is critical. Again, users must remain aware of possible cryptanalytic developments regarding any particular cryptographic hash function, as new discoveries regarding existing algorithms are made and as new algorithms are developed. The MD5 hashing algorithm is designed to be simple to implement and very efficient on 32-bit computer systems. It does not require any large substitution tables and can be coded very compactly. Table 3 below depicts a summary of popular hash functions and their relative speeds and upper bounds on strength. The number of cipher or compression function operations currently believed necessary to find preimages and collisions are also specified.

Name	Bitlength	Operations to Find Collision	Operations to Find Preimage	Rounds x Steps per round	Relative Speed
MD4	128	2^{20}	2^{128}	3 x 16	1.00
MD5	128	2^{64}	2^{128}	4 x 16	0.68
SHA-1	160	2^{80}	2^{160}	4 x 20	0.28

Table 3: Summary of selected hash functions based on MD4

Chapter 4

4.1 The Need for a National Software Reference Library

A typical desktop computer contains gigabytes of information—perhaps as many as 100,000 individual files or even more. In some investigations, multiple computers and various storage devices such as magnetic tapes, thumb drives, and other types of media are involved. To manually investigate each and every one of the files stored within a typical computer system would take a monumental effort and be very expensive. Such an undertaking could take literally thousands of staff hours and months to complete. Compounding the problem is the large increase in criminal cases involving electronic evidence over the past decade. “The FBI alone investigated well over 5,000 cases [in 2002], compared with a few hundred just 10 years ago” [31]. These cases include child pornography, racketeering, hacking, cyber-attacks, illegal gambling, Internet fraud, identity theft, and software piracy.

Many of the files residing on a typical computer are executable files, operating system files, library files, installation files, etc., and many do not produce evidentiary value toward an investigation. Computer forensics investigators must develop methods and automated tools to efficiently and effectively identify and filter out these unaltered, common system files. Law enforcement officials often utilize databases to identify evidence such as fibers, inks, firearms, and fingerprints in their investigations. With this in mind, the National Institute of Standards and Technology (NIST) developed the National Software Reference Library (NSRL) as a database of cryptographic hashes of

files from legitimate software packages. These cryptographic hashes can be compared to the hashes of files from the body of evidence, and an investigator can dramatically eliminate as many known files as possible that are not relevant to the investigation.

4.2 Established Criteria by Law Enforcement

The NSRL is designed to meet four criteria established to counter law enforcement's objections to other computer forensics tools available in the marketplace. The objections and criteria for a software library and signature database were [32]:

- 1) Objection: "There are no unbiased and neutral organizations involved in the implementation of investigative tools."

Criteria: NIST is a neutral organization (not law enforcement or a software vendor) chosen for its international reputation in providing clean, unbiased, and objective reference data.

- 2) Objection: "Law enforcement has no control over the quality of data provided by the available tools since they come from independent market-driven sources."

Criteria: NIST provides an open rigorous process for assuring the quality of the data.

- 3) Objection: “There are no repositories of original software available from which data can be reproduced.”

Criteria: The NSRL will become an international resource software repository for the constituent file information included in the data. NIST data is traceable and court-admissible.

- 4) Objection: “Each tool provides only a limited set of capabilities with respect to the information that can be obtained from file systems under investigation.”

Criteria: The reference data will include full information on each file including cross-reference of data for use by other tools.

4.3 Construction of the National Software Reference Library

The NSRL project is supported by the U.S. Department of Justice’s National Institute of Justice, NIST’s Office of Law Enforcement Standards, the Federal Bureau of Investigation, the Department of Defense Computer Forensics Laboratory, the Department of Justice’s Technical Support Working Group, the U.S. Customs Service, and numerous other federal, state and local law enforcement, government, software vendors, and industry organizations. The NSRL project is designed to provide research, development, and evaluation of new and existing forensic technologies and methods to further the effective and efficient use of technology used in the investigation of computer related crimes.

The NSRL is a physical repository of nearly 4,000 software titles including operating systems, utilities and applications, database management systems, graphics packages, component libraries, games, etc. The library contains a balance of the most popular (most encountered by investigators) and most desired (most pirated by criminals) software products. The NSRL currently contains software in 32 languages. Information about the software such as application name, version, manufacturer, etc. is entered into a database, and a unique identifier is allocated to the software package, as well as identifiers for each piece of media in the package. The NSRL also contains file profiles and file signatures (or “fingerprints”) that can be used to identify known and unknown files on computer systems that are being analyzed as part of an investigation. Each software title is catalogued and stored on shelves at NIST for archival and reference purposes.

The NSRL gathers its software from numerous sources. Original commercial off-the-shelf software is both purchased by, and donated to, the NSRL. Individual software manufacturers and other organizations make donations are made either via the original commercial media or a download from a corporate or other website. This software is documented as an original source for known files and stored on CD, DVD, diskette, or magnetic tape as a permanent part of the NSRL. “The concept is to collect as many different examples, versions, and updates of software as possible in order to generate file signatures for as many known files as possible” [33].

“The NSRL is also investigating downloaded files from websites, by burning the downloads onto CDs that can be stored on [its] shelves. The digital signatures from these files are not traceable to [NIST’s] level of satisfaction and are not included in the RDS, but are available as they may be of interest to the community.” [34]

4.4 The NSRL Reference Data Set (RDS)

The NSRL gathers its software from many sources as stated above and then integrates file profiles computed from this software containing file signatures and other identifying information into a Reference Data Set (RDS). As of December 2003, the RDS contains nearly 18 million files (consisting of nearly 8 million hash codes). Law enforcement, government, and industry can utilize the RDS to review computer files by matching file profiles with those in the database to make their investigations more efficient.

This is achieved by calculating a unique identifier or hash code which is a hexadecimal character string for each file based on its contents. The hash code is computed in such a way that if one bit in the file is changed, a completely different hash code is produced. “To minimize the possibility that two different files may generate the same hash code, a sufficiently large hash value is computed” [3].

Each fingerprint is unique to a specific file and can be used to determine if [35]:

- A file has been altered.
- A file has been renamed or other means to hide it have been attempted.
- A file is what it purports to be.
- A file is missing when it should be found.
- A file is actually present on a disk.

Four file signatures are created for each file within the RDS. The hash values used in NSRL's Reference Data Set are the Secure Hash Algorithm (SHA-1), Message Digest 4 (MD4), Message Digest 5 (MD5), and a 32-bit Cyclical Redundancy Checksum (CRC32). The use of multiple algorithms allows any one particular hash value to be cross-referenced. "Additionally, this further ensures that no two files will have the same set of hash values" [3].

The hash values, directory name, file name, file size, version, and other source information for each file are stored within the RDS. The computed hash values are validated by a separate, parallel, and independent process to ensure they can be verified to identify specific files in the RDS. Upon successful verification and validation, the RDS is written to a master CD, duplicated, and distributed through NIST's Standard Reference Data Office as Special Database #28, available online at <http://www.nist.gov/srd/nistd28.htm> [35].

RDS subscriptions are available from NIST at a current cost of \$90 per year, which entitles the purchaser to receive up to four quarterly releases. Those who contribute to the NSRL receive one release at no cost. NIST encourages, and does not charge for, redistribution of the RDS.

Special Database #28 was first released in October 2001. The RDS is distributed when sufficient changes to the database have been made, which to date has been quarterly. Each release is a cumulative, full version. The NSRL is used by many law enforcement and computer forensics organizations by importing data from the RDS into various computer forensic tools.

A permuted index accessible via the Internet lists software made available with the latest RDS release. It can be sorted by product name (i.e., "Age of Empires" or "Nero"), manufacturer name (i.e., "Microsoft" or "Ahead Softwrae"), application type (i.e., "Game" or "CD Burning"), language, operating system, or product code (not intuitive but included for cross references).

4.5 Uses of the RDS

Law enforcement and computer forensics investigators are using cryptographic hash databases like the NSRL more and more frequently. "By computing the hash of a suspect file and then looking up that hash in the RDS database, for example, it is possible to determine if that suspect file is a copy of a file that has previously been evaluated, characterized, and registered" [7]. If a specific file's profile and cryptographic hash

value match the database of known files, then the file can be eliminated from close scrutiny. If they do match, the file is unknown and should be examined in greater detail.

An example of this file-reduction technique would be a forensic investigation regarding child pornography on a Windows XP machine. The Windows XP operating system itself contains nearly 6,000 images, which are known gifs, jpegs, icons, etc. By applying the hash sets within the NSRL, the investigator will not have to look at any of those files that match the known file signatures right off the bat. Table 4 below demonstrates three typical graphics files and one rogue graphic file from an investigation.



(© Microsoft Corporation)

bliss.bmp
MD5: AE3FAD12977E9950D0D59E9ABB896616

Matches RDS:
FileSize: 51127
ProductCode: 1746 (Microsoft Windows XP)
OpSystemCode: WIN (Microsoft Windows XP)

The file is flagged as known, and the investigator can disregard it for evidentiary value.

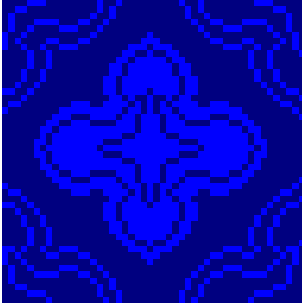


(© Microsoft Corporation)

win2000l.gif
MD5: 0D3B774F122D2CBF22671EA52085D0E6

Matches RDS:
FileSize: 9644
ProductCode: 1746 (Microsoft Windows XP)
OpSystemCode: WIN (Microsoft Windows XP)

The file is flagged as known, and the investigator can disregard it for evidentiary value.



(© Microsoft Corporation)

Blue Lace 16.bmp
MD5: 58EB2320D062E6560872ED8B5809F589

Matches RDS:
FileSize: 5868
ProductCode: 1746 (Microsoft Windows XP)
OpSystemCode: WIN (Microsoft Windows XP)

The file is flagged as known, and the investigator can disregard it for evidentiary value.



enterxxx.jpg
MD5: F4ACDCA7290E07132FD1AC9E4FD88D2B

Does Not Match RDS

The file is flagged as unknown, and an investigator can closely examine it for evidentiary value.

Table 4: Use of the RDS in Examining Graphics Files

Investigators can also search for files that are something other than what they purport to be. “It is not uncommon for a suspect to hide evidence (i.e., a pornographic .BMP image) by renaming the file to the same name found in standard operating systems (i.e., a .BAT file) or software applications” [6]. The contents and corresponding hash value derived from the camouflaged image will not match the file it claims to be, and it will not match the entry for the system file within the RDS.

Conversely, the suspect could disguise a known malicious executable file as a harmless .JPG image, hoping it will go unnoticed. Even if the filename and extension are changed, the contents and corresponding hash values derived from the file will not

change. If the computed hash value exists as a known malicious file within the RDS, the suspect's attempts to thwart detection will fail.

"The NSRL contains both benign and malicious software and is intended to be used as a filter of 'known' file signatures, not 'known good'" [36]. Investigators can also search for files that match a certain profile in the RDS, such as pirated software in the case of a suspected intellectual property case. Another example would be to search for a malicious hacking tool or cracked software.

Another use of the RDS is to determine if expected files are missing from a computer system. This would be a red flag to the investigator, causing him or her to probe further. For example, the investigator may determine that the suspect has attempted to hide illegal activity by deleting the missing files.

4.6 Specifications and File Formats of the RDS

NIST has produced a detailed report outlining the formats of data included in the NSRL RDS distribution [37]. The contents of that document relating to individual data elements and logical and physical record structures have been reproduced within this section to provide the reader with an understanding of the database.

4.6.1 RDS Data Elements

Table 5 below represents data elements used in the NSRL RDS distribution package. Char represents data of type character using UTF-8 encoding of 8-bit bytes.

Integer represents data of type integer including variations of the integer type (short, long, etc.)

All of the data is stored in the distribution files in human-readable form. No binary data or nonstandard characters are used. Char fields are represented by alphabetic, numeric, and punctuation character strings surrounded by double quotes (“”). Integers are represented by unquoted strings of decimal digits.

DATA ELEMENT	TYPE	MAXIMUM LENGTH (IN CHARACTERS)	DESCRIPTION
ApplicationType	Char	50	Character string that identifies a general use of the software product
CRC32	Char	8	32-bit Cyclic Redundancy Checksum (file signature) of a specific file as defined in CCITT X.25 link-level protocol and FIPS PUB 71
FileName	Char	255	Name of a specific file within a software product
FileSize	Integer	15	Size in bytes of a specific file
Language	Char	150	Character string that identifies the language(s) used in the software product
MD5	Char	32	128-bit Message Digest 5 (file signature) of a specific file as defined in IETF RFC 1321
MfgCode	Char	15	Character identifier of a specific vendor or manufacturer
MfgName	Char	150	Identifying name of the vendor or manufacturer of the software product, e.g., “Microsoft”

DATA ELEMENT	TYPE	MAXIMUM LENGTH (IN CHARACTERS)	DESCRIPTION
OpSystemName	Char	150	Identifying name of the operating system on which the software product executes, e.g., "Windows NT"
OpSystemCode	Char	15	Code identifier of a specific operating system version
OpSystemVersion	Char	15	Characters that identify individual versions of an operating system on which the software product executes, e.g., "4.0"
ProductCode	Integer	15	Identifier of a specific software product, e.g., "103"; maps to the NSRL database
Product Name	Char	150	Identifying name of the software product, e.g. "Netscape Communicator"
ProductVersion	Char	15	Characters that identify individual versions of a software product, e.g., "3.0"
RDSVersion	Char	20	Character string that identifies the date and version of the RDS distribution
SHA-1	Char	40	160-bit Secure Hash Algorithm message digest (file signature) of a specific file as defined in FIPS PUB 180-2
SpecialCode	Char	1	A single character field that identifies special file signature entries, such as malicious code signatures or other types of special entries

Table 5: Data Elements of the NIST NSRL Reference Data Set
(Source: National Institute of Standards and Technology)

4.6.2 Logical Record Structure of the RDS

A logical record forms one item or grouping of information from the data elements defined in the above table within the NSRL RDS. There are five such logical record types:

1. File record
2. Manufacturer record
3. Product record
4. Operating system record
5. Version record

Each is described in Tables 6 through 10 below. Examples of each type of record are also provided. Figure 4, also shown below, illustrates how these files relate to each other.

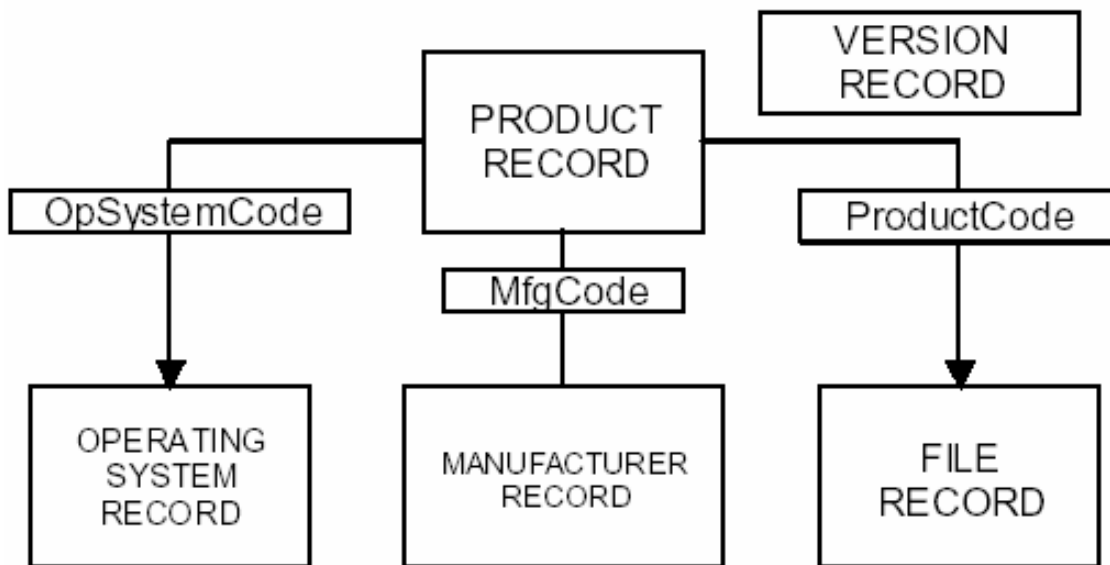


Figure 4: NSRL RDS Logical Record Relationships
(Source: National Institute for Standards and Technology)

RECORD FORMAT	EXAMPLE	COMMENTS
SHA-1	“AC91EF00F33F12DD491CC91EF00F33F12DD491CA”	
MD5	“DC2311FFDC0015FCCC12130FF145DE78”	
CRC32	“14CCE9061FFDC001”	
FileName	“WORD.EXE”	
FileSize	1217654	In bytes
ProductCode	103	The Product record will contain more information about this product code.
OpSystemCode	“NT4WKS”	The Operating System record will contain more information about this operating system code.
SpecialCode	“”	Blank (no value) – normal file “M” – malicious file “S” – special file

Table 6: File Record
(Source: National Institute of Standards and Technology)

RECORD FORMAT	EXAMPLE	COMMENTS
MfgCode	“Microsoft”	MfgCode is referenced in the Operating System and Product records. MfgCode is unique within this record set.
MfgName	“Microsoft Corporation”	

Table 7: Manufacturer Record
(Source: National Institute of Standards and Technology)

RECORD FORMAT	EXAMPLE	COMMENTS
OpSystemCode	“NT4WKS”	OpSystemCode is referenced in the File record and is unique within the Operating System record set.
OpSystemName	“Windows NT”	
OpSystemVersion	“4.0”	
MfgCode	“Microsoft”	MfgCode references an entry in the Manufacturer record.

Table 8: Operating System Record
(Source: National Institute of Standards and Technology)

RECORD FORMAT	EXAMPLE	COMMENTS
ProductCode	103	ProductCode is referenced in the File record and is unique within the Product record set.
ProductName	“Microsoft Word”	
ProductVersion	“2000”	
OpSystemCode	“Win98”	OpSystemCode is referenced in the Operating System record.
MfgCode	“Microsoft”	MfgCode references an entry in the Manufacturer record.
Language	“English”	If multiple languages are present, they will be comma separated within this field.
ApplicationType	“Operating System”	

Table 9: Product Record
(Source: National Institute of Standards and Technology)

RECORD FORMAT	EXAMPLE	COMMENTS
SHA-1	“AC91EF00F33F12DD491CC91EF00F33F12DD491CA”	This value of SHA-1 is computed from the SHA-1 values of the four other files.
RDSVersion	“2001/03/08 0.2”	Assigned to each quarterly release of the RDS.

Table 10: RDS Version Record
(Source: National Institute of Standards and Technology)

4.6.3 Physical Record Structure of the RDS

The RDS consists of five physical data files that correspond to the five logical record types, one file per logical record type. The character format is UTF-8 (8-bit ASCII), one logical record per physical line terminated with ASCII characters 13 and 10 (hexadecimal 0D0A). Individual fields are separated by comma (,) within each line. Character field values are surrounded by double quotation marks (“”). The first record of each file contains the field names instead of data values. Examples of the contents of each file are presented in Tables 1 through 15 below. The first record in each figure represents the first or header record found in each file. The second record in each figure represents all subsequent or detail records in each file.

”SHA-1”,”MD5”,”CRC32”,”FileName”,”FileSize”,”ProductCode”,”OpSystemCode”, ”SpecialCode” <13><10>
“AC91EF00F33F12DD491CC91EF00F33F12DD491CA”,“DC2311FFDC0015FCCC1 2130FF145DE78”,“14CCE9061FFDC001”, “WORD.EXE”,1217654,103, ”T4WKS”,”” <13><10>

Table 11: Example FILE Data
(Source: National Institute of Standards and Technology)

“MfgCode”, “MfgName” <13><10>
“Microsoft”, “Microsoft Corporation” <13><10>

Table 12: Example MANUFACTURER Data
(Source: National Institute of Standards and Technology)

“OpSystemCode”, “OpSystemName”, “OpSystemVersion”, “MfgCode” <13><10>
“NT4WKS”, “Windows NT”, “4.0”, “Microsoft” <13><10>

Table 13: Example OPERATING SYSTEM Data
(Source: National Institute of Standards and Technology)

“ProductCode”, “ProductName”, “ProductVersion”, “MfgCode”, “OpSystemCode” <13><10>
“103”, “Microsoft Office”, “2000”, “Microsoft”, “Win98”, “English”, “Word Processor” <13><10>

Table 14: Example PRODUCT Data
(Source: National Institute of Standards and Technology)

“SHA-1”, “RDSVersion” <13><10>
“DD161AEFCC271124533FFFA1445764BDE12515AE”, “2001/03/08 0.2” <13><10>

Table 15: Example RDS VERSION Data
(Source: National Institute of Standards and Technology)

4.7 Current Version of the RDS

The latest version of RDS is Version 2.3, which was released in December 2003 on four CDs. Each CD can be used separately as a targeted hash set for any of four categories: non-English files, operating systems, applications, and images. Each hash set is variable in size with a full complement of files from one or more packages. “The files contained within the RDS are named NSRLFILE.TXT, NSRLOS.TXT, NSRLMFG.TXT, NSRLPROD.TXT, and VERSION.TXT” [38]. The investigator can determine whether to use these files separately, or to concatenate and arrange them into a

combined database of information, as needed. The contents of the latest RDS version are shown below in Table 16.

CD #	Contents	Files	Unique SHA-1 Values
CD "A"	Non-English Files	4,644,674	1,465,141
CD "B"	Operating Systems	2,513,772	1,078,149
CD "C"	Applications	7,545,675	3,209,385
CD "D"	Images	3,205,843	2,568,575
TOTAL		17,909,964	7,198,856

Filename	Corresponding MD5 and SHA-1 Values
NSRLMfg.txt	MD5: 0AD310394129BB2F031ECE85DA019CD5
	SHA-1: A7F79564A95CC4AC023012191539BA536AE4C606
NSRLOS.txt	MD5: 0E36C2617221AB6962CCB3D70F835D9A
	SHA-1: E316A8F86CCE25AAC996BC4FBF217DD5393DA040
NSRLProd.txt	MD5: 4CEB71FFBFF905EDE38F3D6A6317F514
	SHA-1: E5B3C6815C4AA0966F86D524DB356414063F93E4
NSRLFile.txt (CD "A")	MD5: C0214C9E873742175F37728277C1081E
	SHA-1: B7DAFF4A43D39AF918947254E2634559829E754A
NSRLFile.txt (CD "B")	MD5: EC72F9E0509E9B6FF1F7B0E938E2EB3C
	SHA-1: 7FE9F1986A6DD7589ED9DAD7E9CD3C9FACF8D954
NSRLFile.txt (CD "C")	MD5: DC48A4A42BE49D86563C104DF534E289
	SHA-1: C684A18C2A6297F6ACFC875766A767C28FE73CD6
NSRLFile.txt (CD "D")	MD5: B15B2DAFEA1A0C821CC3117251B43B1D
	SHA-1: 211CA86816C235BE02A9D89C3E3801D22298AB91

Table 16: Current Version of the NSRL RDS
(Source: National Institute of Standards and Technology)

4.8 Effectiveness of the RDS Hash Sets

The NSRL allows the investigator to focus on unknown files which do not have profiles and fingerprints in the NSRL database. “The reference library is a tool that can cut an investigator’s time by 25 to 95 percent, depending on the number of files on the hard drive,” according to Gary Fisher of the NIST’s Information Technology Laboratory and project manager for NSRL [31]. Table 17, a reprint of Table 1 from Chapter 1, is shown below with the number and percentages of files successfully identified by the NSRL RDS.

OS/APPS	FILES INSTALLED	PERCENT IDENTIFIED	FILES UNKNOWN	FILES ON DISTRIBUTION CD(S)
Virgin Win98	4,266	93%	297	18,662
Virgin NT4 WS	1,659	86%	239	17,904
Virgin Win2K Pro	5,963	86%	839	16,539
Virgin Win ME	5,169	93%	383	11,512
Win98+Office 2K	23,464	98%	596	43,327
Win ME+Office 2K	24,112	98%	526	32,758

Table 17: Effectiveness of the NSRL RDS
(Source: National Institute of Standards and Technology)

Chapter 5

5.1 Software Design Methodology

In the analysis stage of a computer forensics investigation, it is not typically possible or practical to examine all suspect files. Therefore, investigators rely upon effective yet efficient methods that can quickly reduce the number of files requiring close examination. One such method, to group files into two general categories: known and unknown, is implemented by this research. The resulting software application is detailed within this chapter. The underlying design methodology is to calculate the hash values for suspect files and compare them with a database of known file hash values and file profiles, i.e., the NSRL RDS database. An overview of this process is given below in Figure 5.

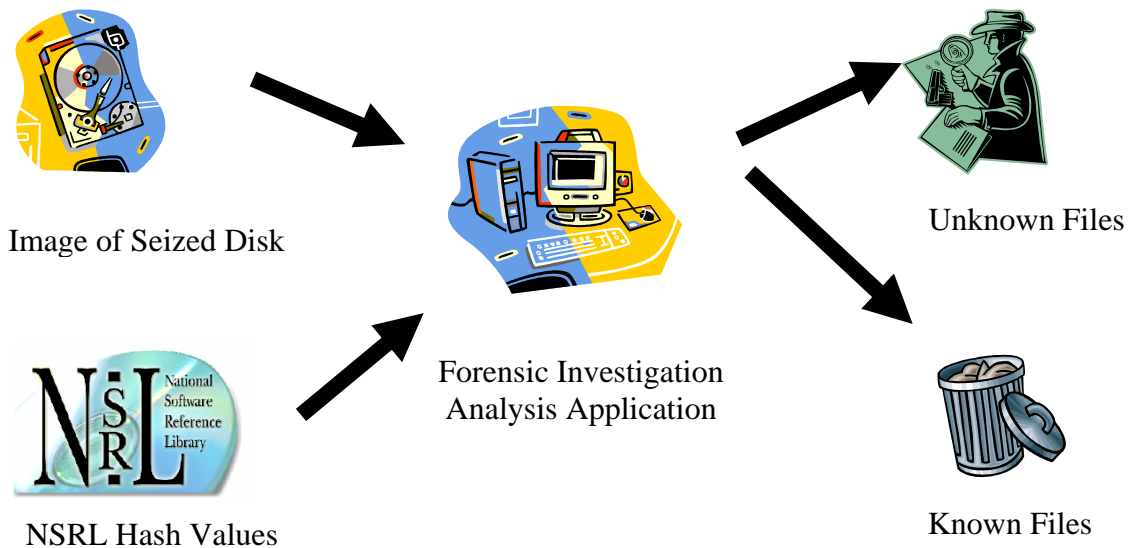


Figure 5: Software Design Methodology

This software application assumes that the drive being analyzed is a forensic bit stream backup of the original storage media. This ensures that none of the original evidence can possibly be damaged or corrupted during the analysis process. The application also assumes that any and all previously deleted files have been restored using a separate forensic file recovery application.

This software application was created using Visual Basic 6.0, Service Pack 5. This programming environment allows for rapid application development and the creation of a graphical user interface, which is quickly becoming the preferred operating environment by forensics analysts. The full source code listings for this software application can be found in Appendix B of this text. Figure 6, shown below, shows a sample execution of the software application.

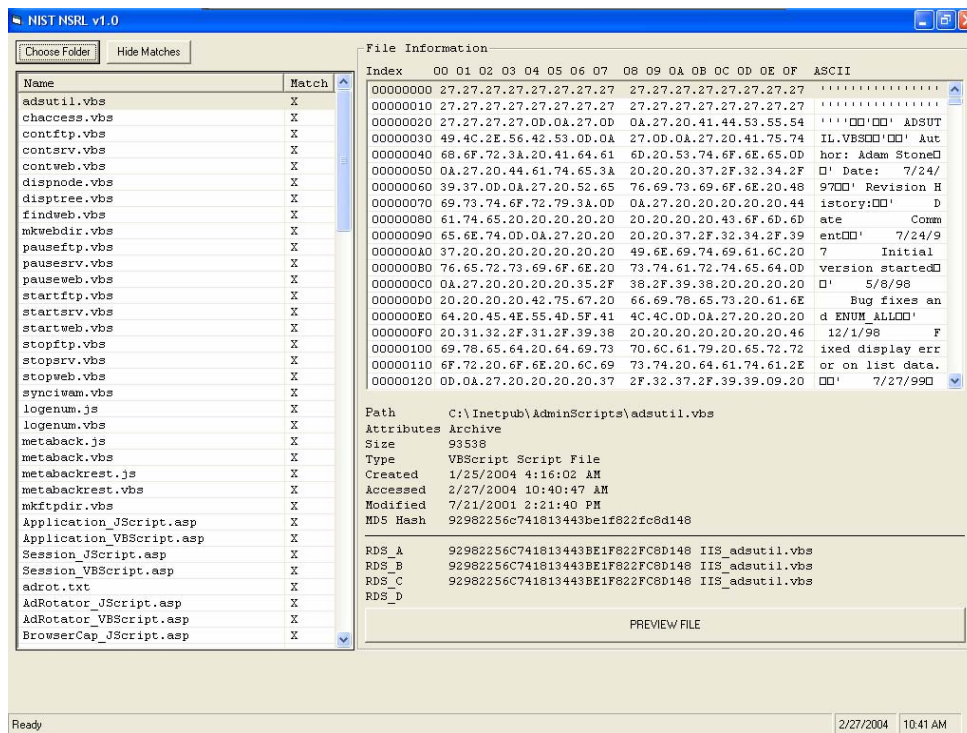


Figure 6: Sample Execution of the Software Application

5.2 Preparing Data from the RDS

As stated in Chapter 4, the RDS contains signatures and file profiles within a four-CD distribution, each of which targets one of the following areas: non-English files, operating systems, applications, and images. Information defining each file's profile is contained in a relational format between several comma separated text files on the RDS distribution:

- NSRLMfg.txt contains information relating individual files to any of 514 software manufacturers.
- NSRLOS.txt relates individual files to any of 28 operating system configurations. These entries include details of the operating system such as name, version, and manufacturer.
- NSRLProd.txt contains information relating individual files to any of 4,726 specific software products. These entries include details of the software product, including name, version, operating system, manufacturer, language, and application type.
- NSRLFile.txt contains the central file profile information. This file contains the SHA-1, MD5, and CRC32 hash values for each file within the RDS, file name, file size, product information, operating system, and a special code. This special code serves as a red flag for the investigator, denoting whether the file is "known good" or "known bad." The entire RDS contains 17,909,964 unique file profiles and 7,198,856 unique hash values.

This data must be organized in such a format that the software program will efficiently access it. A simple lookup from a sequential file will not serve this purpose, and typical spreadsheet applications cannot handle nearly eighteen million unique records. Therefore, a database application such as Microsoft Access is required to facilitate the RDS database and its interaction with the software application.

Each of the files contained within the RDS is imported as tables into Microsoft Access databases. Although this research is only relevant to the NSRLFile file, all other files are imported so that they may be implemented for cross-referencing purposes in the future. To ensure manageability of data, four databases were created (one for each of the four CDs within the RDS). This is done using the Import Wizard to import the files as a comma delimited format. The first row contains field names and text is qualified within quotation marks. Each of the four files on the CD are imported into four tables, each with its respective name. Data types (i.e., text, long integer, etc.) are assigned for each of the fields. The NSRLFile table is then sorted and indexed by the MD5 hash code. This is essential for comparing the calculated hash values from the seized hard drive with the database in an expedient manner. No primary key is needed, since each table has uniquely identifiable codes.

Figures 7 – 10, shown below, show each of the four database tables created from the comma delimited information contained within the RDS database: NSRLMfg, NSRLOS, NSRLProd, and NSRLFile. Each of these particular tables is derived from CD_A, the database containing non-English software.

SHA-1	MD5	CRC32	FileName	FileSize	ProductCode	OpSystemCode	SpecialCode
4CDE504FCD0E0038156AB9F7F60E6E6B6266BF4D	0000238B43AF52E6BF9780D25173C	051E410B	MOBSYNC.CH_	11312	2619	WIN	
4CDE504FCD0E0038156AB9F7F60E6E6B6266BF4D	0000238B43AF52E6BF9780D25173C	051E410B	MOBSYNC.CH_	11312	2633	WIN	
4CDE504FCD0E0038156AB9F7F60E6E6B6266BF4D	0000238B43AF52E6BF9780D25173C	051E410B	MOBSYNC.CH_	11312	2964	WIN	
4CDE504FCD0E0038156AB9F7F60E6E6B6266BF4D	0000238B43AF52E6BF9780D25173C	051E410B	MOBSYNC.CH_	11312	3291	WIN	
9D46D1C502A70CF6F56A63F3EE0DC5DC05D01195	000016F07018F95BF4B01E7E11583484	96E19391	428NNTS4.GIF	1080	2886	WIN	
9D46D1C502A70CF6F56A63F3EE0DC5DC05D01195	000016F07018F95BF4B01E7E11583484	96E19391	428NNTS4.GIF	1080	2949	WIN	
9D46D1C502A70CF6F56A63F3EE0DC5DC05D01195	000016F07018F95BF4B01E7E11583484	96E19391	428NNTS4.GIF	1080	2797	WIN	
A6B9FD3433EC17734C922A3B5C87273C66110270	000022286FAC25C704D56A6B8331129	F3BD6558	netprot.cat	6066	3172	WIN	
196A5914FFFD17DAB8FF84428D8B961D4474AEF8	00002C6A13B13FB588B87C6204E73B0C	18067EA5	BOOTSTRAP.MSI	79360	2825	WIN	
196A5914FFFD17DAB8FF84428D8B961D4474AEF8	00002C6A13B13FB588B87C6204E73B0C	18067EA5	BOOTSTRAP.MSI	79360	2997	WIN	
1E3A1F5ED28C49541A786E01C0ABAF485880E1E	0000315467D336EB5EF32C584AE63EC	7B77ACCA	SETUPLDR.EX_	106089	2964	WIN	
1E3A1F5ED28C49541A786E01C0ABAF485880E1E	0000315467D336EB5EF32C584AE63EC	7B77ACCA	SETUPLDR.EX_	106089	2972	WIN	
1E3A1F5ED28C49541A786E01C0ABAF485880E1E	0000315467D336EB5EF32C584AE63EC	7B77ACCA	SETUPLDR.EX_	106089	2633	WIN	
1E3A1F5ED28C49541A786E01C0ABAF485880E1E	0000315467D336EB5EF32C584AE63EC	7B77ACCA	SETUPLDR.EX_	106089	2619	WIN	
1E3A1F5ED28C49541A786E01C0ABAF485880E1E	0000315467D336EB5EF32C584AE63EC	7B77ACCA	SETUPLDR.EX_	106089	2633	WIN	
E4B2295732E38B4A98C0CA61853DC8BE52E1E627	00003EB4947FBFB0840BE1CDD7627A69	AB24D56F	imm32.dll	96768	3148	WIN	
34BAC2F4DC05F5007DA6F5B6AE9CA5ECBE235	00004343D9802EC1BC95EA308E0F1FE8	65A97B13	ICONNECT.HTM_MU_	2521	1697	UNK	
C49FC9C512A711046AF1893384127E8337C8931	000049127C704B1439A71903B0957D0B	8CAC3474	mmsw.txt	178	3250	WIN	
C49FC9C512A711046AF1893384127E8337C8931	000049127C704B1439A71903B0957D0B	8CAC3474	mmsw.txt	178	3147	WIN	
C49FC9C512A711046AF1893384127E8337C8931	000049127C704B1439A71903B0957D0B	8CAC3474	mmsw.txt	178	3172	WIN	
028D24EF69EE439F377E78FC8B65951137A3E16	00004D6B900255B8D41F1DD5D3CF4883	0E9593A5	lz32.dll	24576	3156	WIN	
AF3CD9E381E674B2C6B83CF3900DDA822BC5C0	00005B410A87A2AC4925DDDF96EAF5AF	45502D91	connect.hlp	46773	3156	WIN	
CC2704B9C6134A99C33EFD51E1F3C9A325083F4	000062D41CD86146968F0FD6215645DB	534212D2	CHCP.CO_	3363	2949	WIN	
CC2704B9C6134A99C33EFD51E1F3C9A325083F4	000062D41CD86146968F0FD6215645DB	534212D2	CHCP.CO_	3363	3269	WIN	
CC2704B9C6134A99C33EFD51E1F3C9A325083F4	000062D41CD86146968F0FD6215645DB	534212D2	CHCP.CO_	3363	2886	WIN	
A96CD563DBE9E5EB8DDB4BAEAD092D7B35115E0	000069C2686A95F5B334A36CEBAEC42	A4953EBD	FUSION.STY	55828	1794	UNK	
C47FD0E61F98DF711B642E5A2BCE83B8652615C	0000761515A867A4AF658D268F2F1B39	6DC2BDFB	CDM_DL_	4760	2603	WIN	
C47FD0E61F98DF711B642E5A2BCE83B8652615C	0000761515A867A4AF658D268F2F1B39	6DC2BDFB	CDM_DL_	4760	2949	WIN	
C47FD0E61F98DF711B642E5A2BCE83B8652615C	0000761515A867A4AF658D268F2F1B39	6DC2BDFB	CDM_DL_	4760	2886	WIN	

Figure 7: Sample NSRFile Table

MfgCode	MfgName
3Com	3Com
3M	3M
602	Software602, Inc.
AB	Applied Biosystems
Abbyy	Abbyy USA Software House, Inc.
Absoft	Absoft Corporation
Acbel	Acbel Technologies, Inc.
AccessData	AccessData
ACD	ACD Systems, Ltd.
Acer	Acer Communications & Multimedia America, Inc.
ACM	Association for Computing Machinery
Active	ActiveState Corporation
Activision	Activision, Inc.
Adaptec	Adaptec, Inc.
Addison-Wesley	Addison-Wesley Publishing Company, Inc.
Adobe	Adobe Systems Incorporated
Advanced	Advanced Gravis Computer Technology Ltd.
AEC	Aec software
AGE	AGE Logic, Inc.
Ahead	Ahead Software AG
Aker	Aker Corporation
Aladdin	Aladdin Systems Inc.
Aldus	Aldus Corporation
Alexsys	Alexsys Corporation

Figure 8: Sample NSRLMfg Table

OpSystemCode	OpSystemName	OpSystemVersion	MfgCode
Mac71	Macintosh 7.1	7.1	Apple
AIX	AIX	Generic	Unknown
AIX433	AIX 4.3.3	NA	Unknown
AS/400	AS/400	N/A	Unknown
AT	AT	NA	Unknown
CE	CE	Unknown	Microsoft
Compaqalpha	Compaq Alpha Tru64	unknown	Compaq
Compaqdos331	Compaq Dos 3.31	3.31	Microsoft
DOS	MSDOS	Generic	Microsoft
DOS2.1	MSDOS 2.1	2.1	Microsoft
DOS2.11	MSDOS 2.11	2.11	Microsoft
DOS20	MSDOS 2.0	2.0	Microsoft
DOS3.1	MSDOS 3.1	3.1	Microsoft
DOS3.3	MSDOS 3.3	3.3	Microsoft
DOS3.5	MSDOS 3.5	3.5	Microsoft
dos32	MSDOS 3.2	3.2	Microsoft
DOS401	MSDOS 4.01	4.01	Microsoft
DOS6.22	MSDOS 6.22	6.22	Microsoft
DOS6.3	MSDOS 6.3	6.3	Microsoft
DOS60	MSDOS 6.0	6.0	Microsoft
dos62	MSDOS 6.2	3.2	Microsoft
dos621	MSDOS 6.21	6.21	Microsoft
drdos5	DR Dos 5	5	Unknown
drdos6	DR Dos 6	5	Unknown

Figure 9: Sample NSRLOS Table

ProductCode	ProductName	ProductVersion	OpSystemCode	MfgCode	Language	ApplicationType
1	Norton Utilities	2.0 WinNT 4.0	WINNT	SYM	English	Utility
2	CRT	2.4	Gen	Unknown	English	Telnet
7	Harvard Graphics	3.0 Upgrade	DOS	SPC	English	Presentation
8	ScreenShow	N/A	DOS	SPC	English	Screen Saver
9	Norton Utilities	8	DOS	SYM	English	Utility
9	Norton Utilities	8	Gen	SYM	English	Utility
9	Norton Utilities	8	WIN	SYM	English	Utility
14	FastTrackSchedule	Windows	WIN	AEC	English	Calendar
16	Report Writer	N/A	dos621	CLA	English	Reports
16	Report Writer	N/A	DOS2.11	CLA	English	Reports
16	Report Writer	N/A	DOS3.3	CLA	English	Reports
16	Report Writer	N/A	DOS2.1	CLA	English	Reports
16	Report Writer	N/A	DOS3.5	CLA	English	Reports
16	Report Writer	N/A	DOS	CLA	English	Reports
16	Report Writer	N/A	DOS3.1	CLA	English	Reports
16	Report Writer	N/A	MSDOS5	CLA	English	Reports
17	R&R Report Writer	4	DOS6.22	CDS	English	Reports
17	R&R Report Writer	4	DOS3.1	CDS	English	Reports
17	R&R Report Writer	4	MSDOS5	CDS	English	Reports
17	R&R Report Writer	4	DOS	CDS	English	Reports
17	R&R Report Writer	4	dos621	CDS	English	Reports
17	R&R Report Writer	4	DOS3.5	CDS	English	Reports
17	R&R Report Writer	4	DOS6.3	CDS	English	Reports
17	R&R Report Writer	4	DOS3.3	CDS	English	Reports

Figure 10: Sample NSRLProd Table

In order to facilitate data between the four database tables, relationships are required between common fields (i.e., ProductCode, OpSystemCode, MfgCode). Figure 11, shown below, demonstrates the relationships that are created between data tables.

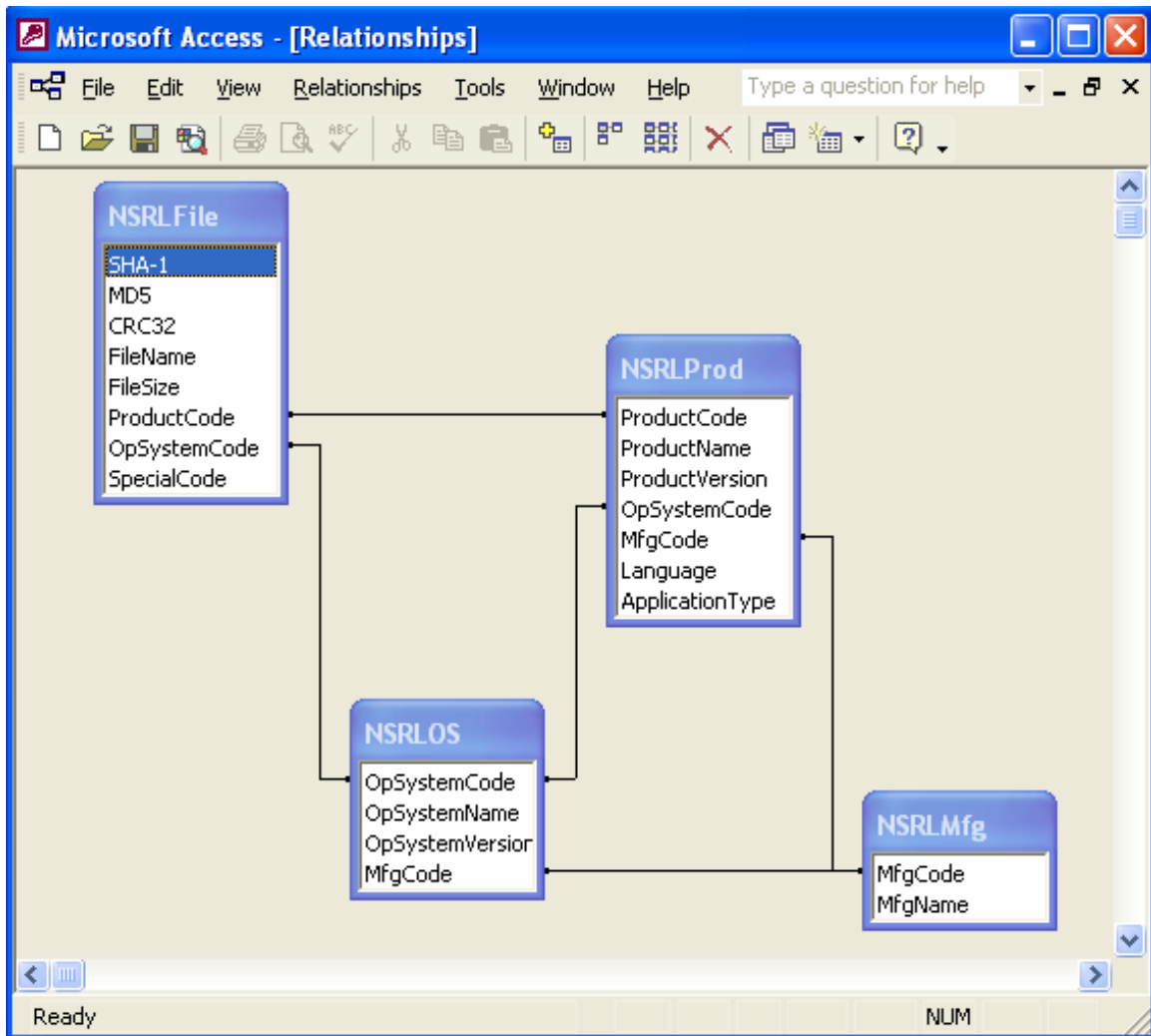


Figure 11: Relationships Created Between Data Tables

5.3 Logging the Investigation

Two types of logs are kept by the software application. These logs can be used to further assist the investigator, i.e., he or she may take the information produced in comma

separated format by these log files and manipulate the data using a spreadsheet application. The first is a general application log that contains the date and time of every command performed by the investigator. This log file is created within a directory named “Logs” at the beginning of the software application’s execution and the filename corresponds with the current date and time at execution. For example, if an investigation were performed on February 27, 2004 at 10:40:39 (24-hour clock), the resulting filename would be 02272004104039.txt. This naming convention ensures that no two log files will have the same name. An example of the contents of a typical log file is shown below:

```
02/27/2004 10:40:39 Log File Created
02/27/2004 10:40:47 Disk Analysis of C:\Inetpub created.
                    Filename: 02272004104047.txt
02/27/2004 10:42:21 File preview of filename: printer.gif
02/27/2004 10:43:02 Disk Analysis of C:\Windows created.
                    Filename: 02272004104302.txt
02/27/2004 11:24:36 Log File Closed
```

The other log file created by the software application maintains information about each of the files investigated by the analysis process. This log file is created within a directory named “Diskanalysis” at the beginning of each execution of the “Choose Folder” command. The filename is created using the same naming convention as the general application log. This log contains the path of each file investigated by the software application, its MD5 hash result, and a flag alerting the investigator whether or not the file matches the RDS database. An example of the contents of a typical log file is shown below:

```
"Path", "MD5 Hash", "Match"  
"C:\WINDOWS\Web\TSWeb\bluebarh.gif", "409f500aca53f8102d9a8c2dbd1f1a61", "X"  
"C:\WINDOWS\Web\TSWeb\bluebarv.gif", "f915c1b57047a31fe0e257e8e853e5f9", "X"  
"C:\WINDOWS\Web\TSWeb\default.htm", "06c36aalb2c265acc7d4b49745eda57", "X"  
"C:\WINDOWS\Web\TSWeb\msrdp.cab", "7da462cd62642f2a61e8fec78cdf52a1", "X"  
"C:\WINDOWS\Web\TSWeb\Thumbs.db", "718f03a2785433c26c8d960f64879b25", ""  
"C:\WINDOWS\Web\TSWeb\win2000l.gif", "0d3b774f122d2cbf22671ea52085d0e6", "X"  
"C:\WINDOWS\Web\TSWeb\win2000r.gif", "3540a7d2df234eafcbb475d795284f29", "X"
```

5.4 Searching for Files

The first step in analyzing files contained on the forensic backup of the original media begins with the computer forensics investigator pressing the “Choose Folder” command from the toolbar. The software application utilizes the BrowseForFolder functionality of Visual Basic as shown below:

```
getdir = BrowseForFolder(Me, "Select A Directory", "c:\")
```

A function then executes to recursively list each of the files within the specified folder, including all of the folder’s subdirectories. Using the FileSystemObject Object Model, the software application can easily access folders and files. The function, shown below, extracts information from each file, including name, size, type, creation date and time, accessed date and time, modification date and time, path, and attributes. During this process, the status bar at the bottom of the application window informs the investigator which file is currently being processed. The information gathered by this function is then stored in hidden fields of a ListView control corresponding to the file under investigation for future use.

```

Set fol = fso.GetFolder(sPath)

For Each fil In fol.Files

Set listobj = ListView1.ListItems.add(, , fil.Name)
listobj.SubItems(1) = fil.Size
listobj.SubItems(2) = fil.Type
listobj.SubItems(3) = fil.DateCreated
listobj.SubItems(4) = fil.DateLastAccessed
listobj.SubItems(5) = fil.DateLastModified
listobj.SubItems(6) = fil.Path
listobj.SubItems(7) = fil.Attributes

For Each sub1 In fol.SubFolders
    ShowAllFiles sub1.Path
Next

```

5.5 Hashing Files

Visual Basic does not natively contain any file hashing functionality. The MD5 reference implementation is written in the C programming language, so a dynamic linked library (DLL) is programmed using the source code from the MD5 reference implementation. This DLL, also written in C, allows the software application to compute MD5 digest strings for files. To do so, a Visual Basic module is implemented to contain a wrapper function that takes a filename as input, calls the DLL which generates the MD5 digest of the file's content, and passes the resulting hash value back to the software application. These hash results are stored within hidden fields for each file under examination in the ListView control for later use in comparing hash values with the RDS database.

5.5.1 A Note Regarding Zero-Byte Files

As previously stated, the MD5 file hashing algorithm is performed over an entire message and the resulting hash value depends upon each and every bit of the input. Zero-

byte files contain zero bits; therefore they will always result in the exact hash value. The message digest for zero-byte files is D41D8CD98F00B204E9800998ECF8427E. The NSRL RSD contains numerous zero-byte hash values and file profiles, so an alternative method of filtering these files is required. If a particular file under investigation results in a match with the RDS database, and the resulting hash value is equal to that of a zero-byte file, then other characteristics of that file must be compared to the file profile within the RDS. If the file name matches the one found in the profile, then the file is classified as known. Otherwise, it is classified as unknown, and the investigator can conduct further analysis on the file.

It is also worth noting that zero byte files, if contained on a Windows NT, 2000, or XP NTFS partition, may contain data streams. These data streams are cleverly concealed from the investigator, and do not show as data within the zero-byte file. Thus, each and every zero-byte file on the suspect hard drive should be analyzed for data streams.

5.6 Comparing File Hashes with the RDS

Once a directory and its corresponding subdirectory tree is successfully hashed by the software application, the hash values must be compared with the RDS database and classified as either known or unknown. This procedure is conducted using ADODB connections to each of the four RDS database files created in the file importation process. The software application takes the hash value of the file currently being examined from the ListView control and queries the four databases, in sequential order, using a function

provided by the Microsoft.Jet.OLEDB.4.0 provider named Seek. This function is very efficient in finding the hash values from the millions of records inside the indexed table of files and their corresponding hash values and profiles within the database. If a match is determined, the matching hash value and file name are sent back to the software application where they are stored in hidden fields inside the ListView corresponding to the file currently under examination. Additionally, the file is red-flagged with an “X” in a visible field labeled as “Match” to alert the investigator whether or not the file matches the RDS database. An example of a query performed by the software application on the first database file is shown below:

```
Rset1.Seek hashval
  If Not Rset1.EOF Then
    fname = Rset1.FileName
    outhash = Rset1!MD5
    matcha = outhash & " " & fname
    matchfound = "X"
  Else
    matcha = ""
  End If
```

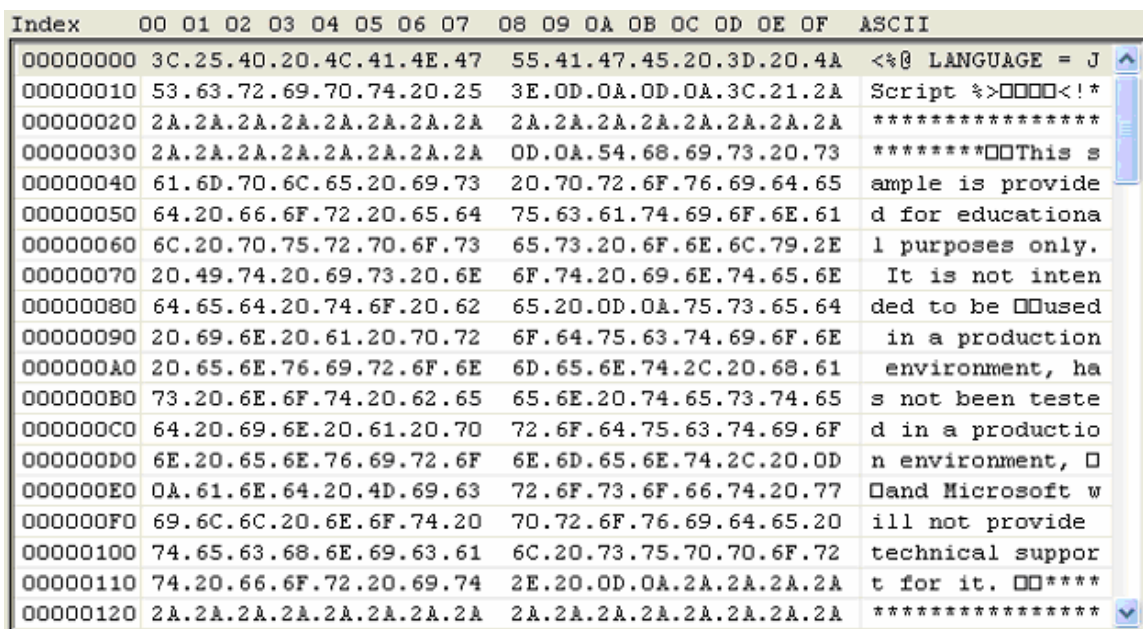
5.7 Investigative Analysis Views

Computer forensics analysis requires investigators to work in a hands-on environment and utilize adept visual sensory skills. To aid the investigator in his or her examination of those files that are unknown (i.e., those files not matching the RDS database of known hash values and file profiles), the software application developed in association with this research features a wealth of visual information. This visual information includes a hex editor view, file information view, and file preview view. These features are explained in detail within the following three subsections.

5.7.1 Hex Editor View

The Hex Editor view allows a computer forensics investigator to determine the hexadecimal and ASCII contents of a file under analysis. By viewing a file's header information, an investigator can quickly determine if a file is camouflaged (e.g., a renamed image, sound, or video file) or if other metadata contained within a file is of evidentiary value.

This procedure loads the file selected from the ListView control and opens it for binary access read access. The contents of the file are read into a string variable named HexText. The file is then closed, and the HexText string is translated by a function named FileToHex which translates the data into its corresponding hexadecimal and ASCII format. The results are displayed in a specially formatted ListView, as shown in the example in Figure 12 below.



Index	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000000	3C	25	40	20	4C	41	4E	47	55	41	47	45	20	3D	20	4A	<%@ LANGUAGE = J
00000010	53	63	72	69	70	74	20	25	3E	0D	0A	0D	0A	3C	21	2A	Script %>□□□□<!*
00000020	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	*****
00000030	2A	2A	2A	2A	2A	2A	2A	2A	0D	0A	54	68	69	73	20	73	*****□□This s
00000040	61	6D	70	6C	65	20	69	73	20	70	72	6F	76	69	64	65	ample is provide
00000050	64	20	66	6F	72	20	65	64	75	63	61	74	69	6F	6E	61	d for educationa
00000060	6C	20	70	75	72	70	6F	73	65	73	20	6F	6E	6C	79	2E	l purposes only.
00000070	20	49	74	20	69	73	20	6E	6F	74	20	69	6E	74	65	6E	It is not inten
00000080	64	65	64	20	74	6F	20	62	65	20	0D	0A	75	73	65	64	ded to be □□used
00000090	20	69	6E	20	61	20	70	72	6F	64	75	63	74	69	6F	6E	in a production
000000A0	20	65	6E	76	69	72	6F	6E	6D	65	6E	74	2C	20	68	61	environment, ha
000000B0	73	20	6E	6F	74	20	62	65	65	6E	20	74	65	73	74	65	s not been teste
000000C0	64	20	69	6E	20	61	20	70	72	6F	64	75	63	74	69	6F	d in a productio
000000D0	6E	20	65	6E	76	69	72	6F	6E	6D	65	6E	74	2C	20	0D	n environment, □
000000E0	0A	61	6E	64	20	4D	69	63	72	6F	73	6F	66	74	20	77	□□and Microsoft w
000000F0	69	6C	6C	20	6E	6F	74	20	70	72	6F	76	69	64	65	20	ill not provide
00000100	74	65	63	68	6E	69	63	61	6C	20	73	75	70	70	6F	72	technical suppor
00000110	74	20	66	6F	72	20	69	74	2E	20	0D	0A	2A	2A	2A	2A	t for it. □□****
00000120	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	2A	*****

Figure 12: Hex Editor View

5.7.2 File Information View

A file information view is also provided to the computer forensics investigator. The information provided in this window includes the complete path name of the file being examined, file attributes, file size, file type, file creation date and time, file accessed date and time, file modified date and time, MD5 hash value, and matching hash values and file profiles that correspond with each of the four RDS databases (if applicable).

This feature provides the investigator with a quick and easy to read overview of the file being examined and its properties. Each of the values listed in the file information view are extracted from hidden fields in the ListView control for the file under examination. The values for the file's attributes are stored as integer values and bit manipulations must be performed in order to create a readable text output value to the investigator. Table 18 below shows the values for possible file attributes. Files can have any of the following values or any logical combination of these values. Descriptions for each attribute and the source code written to determine a file's attributes are also shown.

Constant	Value	Description	Source Code
Normal	0	Normal file. No attributes are set.	If attributeval And &H80
ReadOnly	1	Read-only file. Attribute is read/write.	If attributeval And &H1
Hidden	2	Hidden file. Attribute is read/write.	If attributeval And &H2
System	4	System file. Attribute is read/write.	If attributeval And &H4
Archive	32	File has changed since last backup. Attribute is read/write.	If attributeval And &H20
Compressed	128	Compressed file. Attribute is read-only.	If attributeval And &H800

Table 18: File Attribute Values, Descriptions, and Associated Source Code
(Source: Microsoft Corporation)

An example of the file information view is given below in Figure 13.

```

Path      C:\Inetpub\iissamples\jdk\asp\simple\Looping_JScript.asp
Attributes Archive
Size      1418
Type      Active Server Page
Created   1/25/2004 4:16:13 AM
Accessed  4/1/2004 10:39:23 AM
Modified  7/21/2001 2:22:26 PM
MD5 Hash  1b42859594c13d2211dadddf8add4aed

-----
RDS_A    1B42859594C13D2211DADDDF8ADD4AED IIS_Loopin_JScript.asp
RDS_B    1B42859594C13D2211DADDDF8ADD4AED IIS_Loopin_JScript.asp
RDS_C    1B42859594C13D2211DADDDF8ADD4AED IIS_Loopin_JScript.asp
RDS_D

```

Figure 13: File Information View

5.7.3 File Preview View

A file preview capability is provided to the computer forensics investigator. This feature allows the investigator to open any file residing on the forensic backup of the suspect media (e.g., images, sounds, videos, text files, spreadsheets, web pages, executables, etc.) within the native operating system environment. This can be done only

if an application exists to handle the particular file format. The file preview function can be used in conjunction with the hex editor and can be especially useful in investigations regarding specialized analysis such as pornography, intellectual property, or identity theft cases.

This capability is made possible by calling the Shell "explorer.exe" function within Visual Basic:

```
Shell "explorer.exe" & ListView1.SelectedItem.SubItems(6)
```

The Windows Explorer kernel executes the file specified by the path name in the selected item of the ListView control containing the files being examined. An example of the file preview capability is given below in Figure 14.



Figure 14: File Preview Capability

Chapter 6

6.1 Success of Research Work

The notion of hash filtering has exploded; however, there are several concerns with existing software used to compare hash values on a suspect machine with a database of known hash values. These concerns include that the software is either packaged within a forensics suite, too expensive, too hard to use, or otherwise unavailable for use by the general public or most small law enforcement agencies.

The aim of this research is to create a software tool to automate the analysis of a hard drive under investigation and thus dramatically reduce the number of files that an investigator must individually examine. This tool utilizes the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) database to automatically identify files by comparing hash values of files to “known good” files (e.g., unaltered application files) and “known bad” files (e.g., exploits). This tool provides a much smaller list of files to be closely examined.

The goal of creating a simple, streamlined, standalone public tool for automating the computer forensic investigative process for files on a disk is successfully implemented in this research.

6.2 Implications for Computer Forensics Investigators

The scope of this research lies within the analysis phase of a computer forensics investigation. This presents many implications for the computer forensics investigator. Computer forensics analysis has been customarily performed within a command-line operating system like DOS, or a graphical system like Windows, although analysis within a graphical user interface is being performed more and more frequently. The software application written as part of this research promotes the trend towards analysis within a graphical user interface.

Two of the most important goals of the analysis phase are preserving the integrity of evidence and thorough documentation of the examination. This software application utilizes a forensic bit stream backup of the original storage medium. The investigator can analyze all data on the backup copy that might possibly be relevant to the investigation without modifying or damaging it. This software application also provides the investigator with complete documentation of what evidence is found during the forensic analysis. Log files are created and maintained by the software application with all files discovered and whether or not the files match the RDS database of known hash values and file profiles.

Hex editors are tools that have been invaluable to investigators since the beginning of computer forensics analysis. This software application couples the RDS database with a hex editor. When a particular file is determined to be unknown (by virtue of not matching any of the hash values and file profiles within the RDS) the investigator

can easily use the hex editor view to determine the hexadecimal and ASCII contents of the file. By viewing the file's header information, an investigator can quickly determine if a file is camouflaged (e.g., a renamed image, sound, or video file) or if other metadata contained within a file is of evidentiary value. Furthermore, files can be previewed by the operating system via a "preview file" command inside the software application.

Forensic utilities must be widely adopted by forensic professionals before they can become admissible as evidence examination tools in a court of law. Because NIST is a neutral organization (not law enforcement or a software vendor) with an international reputation in providing clean, unbiased, and objective reference data that has been rigorously validated and verified for quality, the NSRL data contained within the RDS is traceable and court admissible. It has previously been stated that this software application does not modify or damage the original evidence, a prerequisite for admissibility in court.

6.3 Software Application Testing

A battery of tests has been created to examine the effectiveness and efficiency of the software created by this research. These tests were performed on an Intel 2.0 GHz Pentium 4 system with 1 GB of 800MHz Kingston RDRAM and 100GB 7500 RPM 8MB cache Western Digital hard disk drive running Microsoft Windows XP Professional Edition.

6.3.1 Efficiency Tests

The first test is to examine the speed at which the software program can identify files, calculate their MD5 hash values, and compare those hash values with the information contained within the RDS.

This was performed on a variety of data:

- The first test was performed on 1 GB of random-sized data files. The entire process executed in approximately 35 seconds.
- Another test was performed on 1 GB of data consisting only of 1 MB text files. The program executed the process in approximately 50 seconds.
- The final test was performed on 1 GB of data consisting only of 100 MB video files. The program executed the process in approximately 30 seconds.

In summary, a computer forensics investigator can expect this program to execute within roughly 50 – 80 minutes on a 100 GB hard disk drive that is filled to capacity with data. This expected wait time is comparable to other forensic utilities.

6.3.2 Effectiveness Tests

Using the examples for usage of the RDS presented in Chapter 4, another series of tests was created to determine the effectiveness of the software created by this research. The first of these tests examined the scenario in which a suspect may attempt to hide evidence by renaming files to the same names found in standard operating systems or software applications. Four image files were disguised as nondescript operating system files of approximate size. In each case, the files were not recognized by the software

application and were classified as unknown. An example of this test is shown below in Figure 15. Note that the file named “netstat.exe” is actually a camouflaged image file. Examination of the HEX editor view and by executing the “Preview File” feature alerts the investigator that the file is indeed a JPEG image file disguised as an executable file.

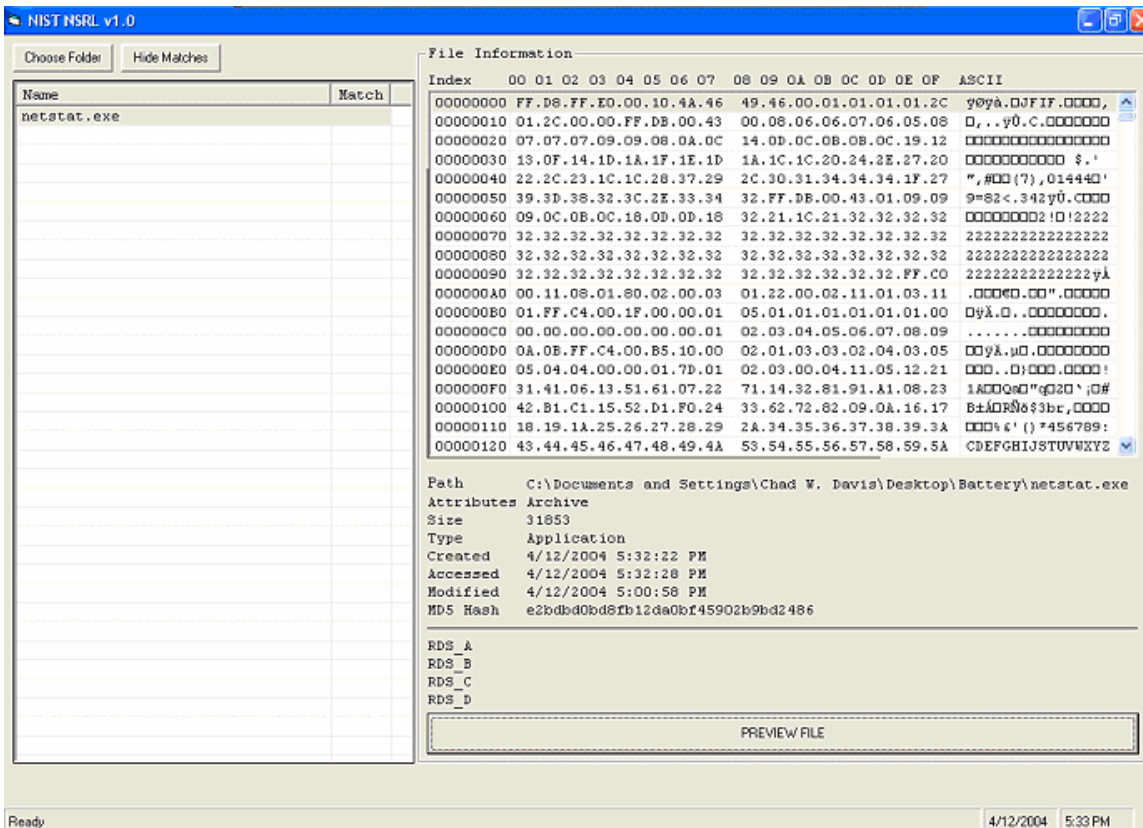


Figure 15: Disguised Image File as an Executable File

A similar test was produced for detecting known files that were renamed to other filename extensions. The software application successfully recognized the files as known, and those files were eliminated from close examination. An example of this test is shown below in Figure 16. Note that the file named “computer.jpg” is actually a camouflaged executable file, “netstat.exe.” Examination of the HEX editor view and by

executing the “Preview File” feature alerts the investigator that the file is indeed an executable file disguised as an image file.

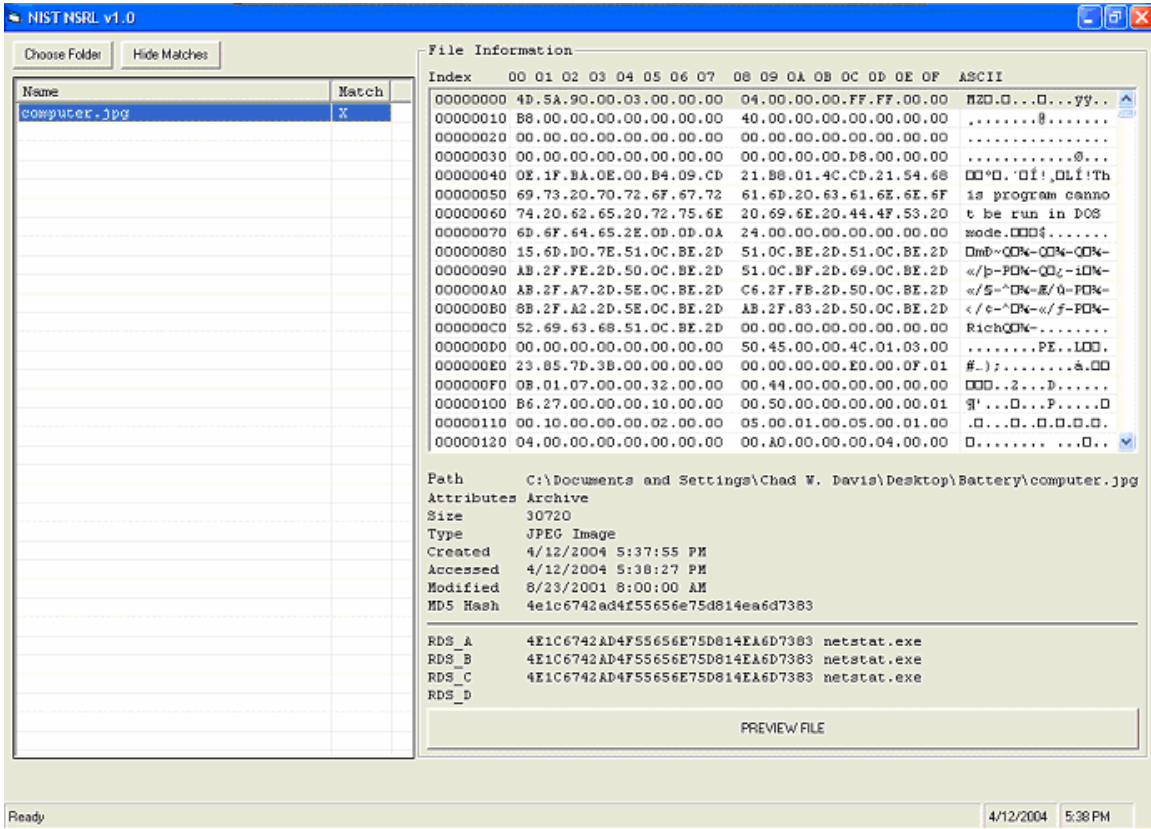


Figure 16: Disguised Executable File as an Image File

A simple test was developed to compare the results of analysis between a legitimate and cracked version of Symantec’s Norton Ghost 2002 utility. The results of the examination of the legitimate version are shown below in Figure 17, and the legitimate version in Figure 18. Notice that three of the files have changed hash values, and no longer match the RDS database of known hash values. The software application properly identified the legitimate and cracked versions as known and unknown, respectively.

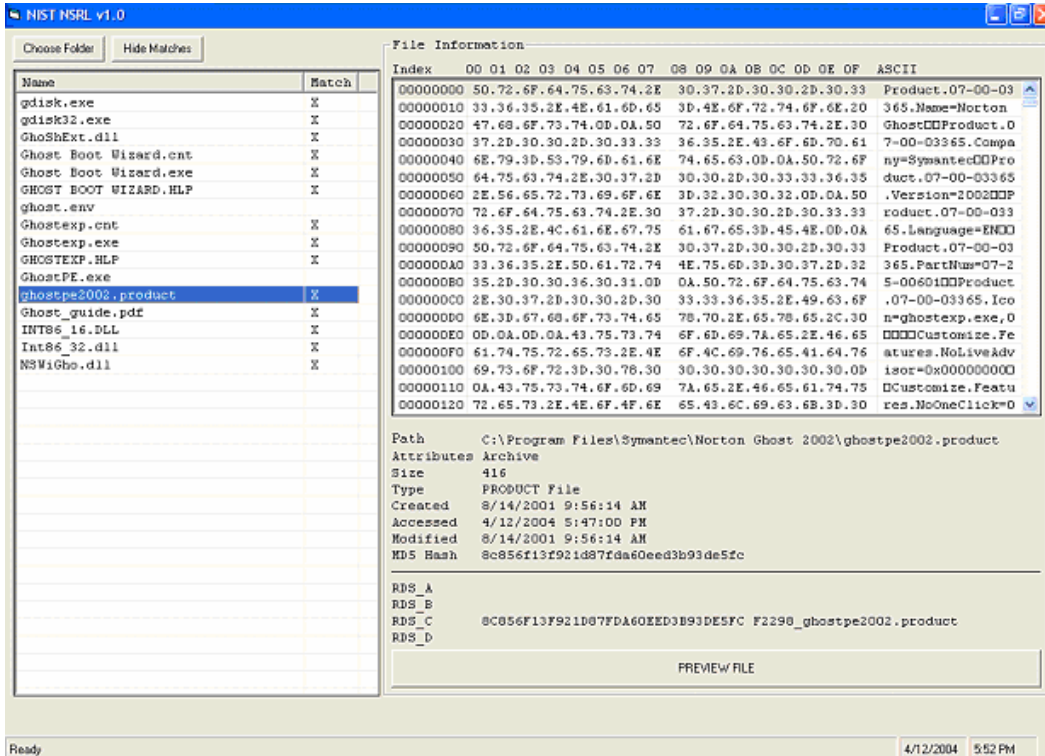


Figure 17: Legitimate Version of Symantec's Norton Ghost 2002 Utility

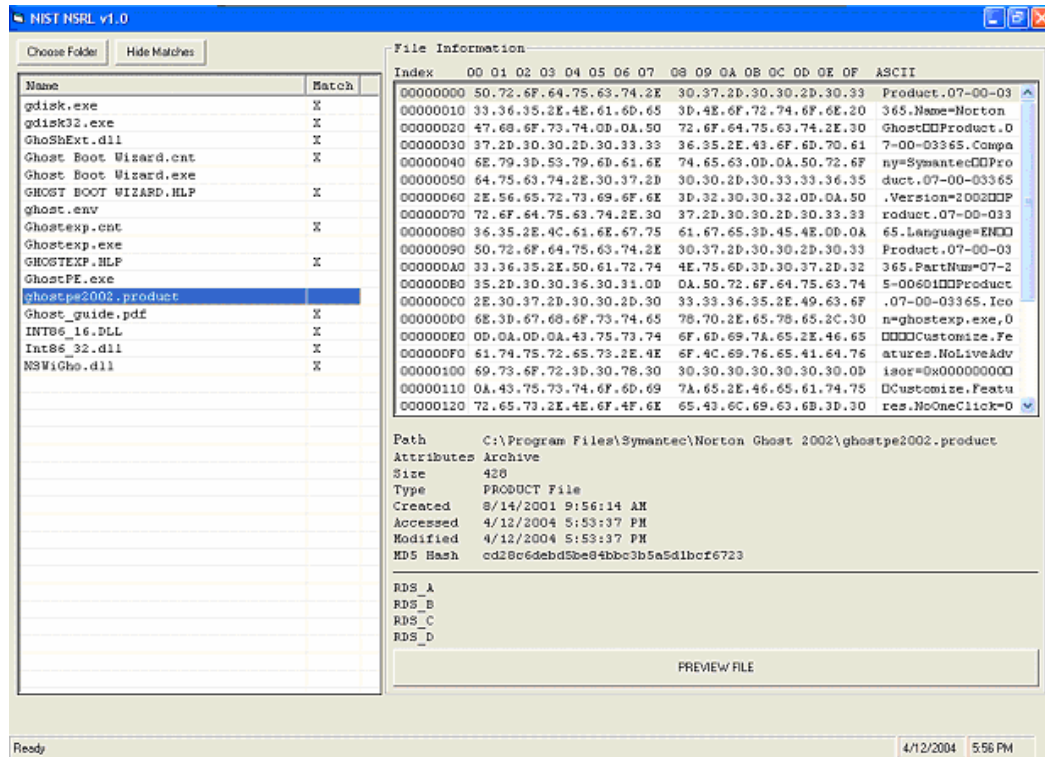


Figure 18: Cracked Version of Symantec's Norton Ghost 2002 Utility

A final test was created to determine the software application’s effectiveness at identifying files containing steganography. The default Windows XP desktop image, “Bliss.bmp,” was injected with the secret message “Computer Forensics” using the wbStego steganography utility and named “Bliss Stego.bmp.” The two files were compared to the RDS using the software application. The program accurately calculated a changed hash value for the “Bliss Stego.bmp” file, and characterized it as unknown. This is depicted below in Figure 19. A computer forensics investigator may determine that steganalysis of the rogue file is appropriate for his or her investigation.

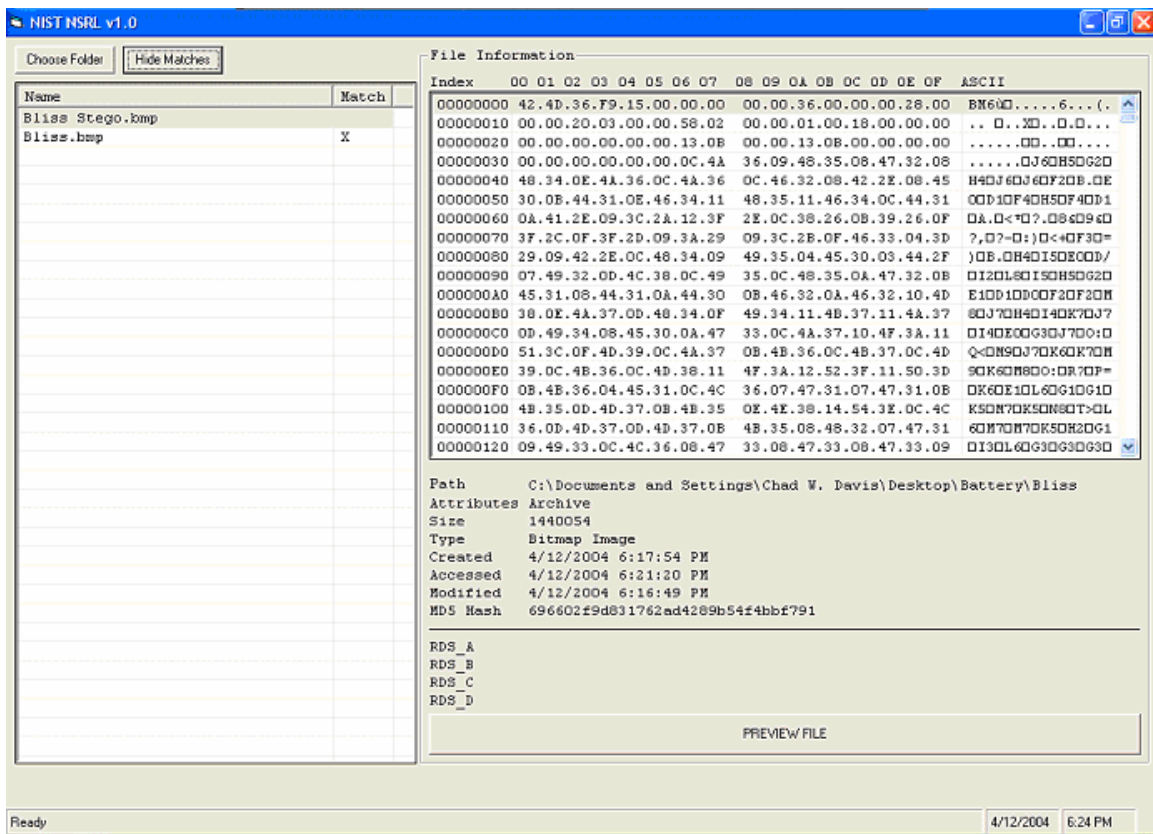


Figure 19: Steganography Within Known Image Files

6.4 Recommendations for Future Work

While the software implemented through this research has made improvements to the analysis phase of a computer forensics investigation, there is still work to be done. Additional research should be performed to improve the software application's efficiency and ease of use. The application should be able to import the data from the RDS distribution in a more intuitive and automated fashion, rather than importing each of the individual hash sets and supporting data files into a relational database. Additionally, researching various database implementations (e.g., SQL, Access, etc.) and their efficiency in looking up data could improve the overhead wait time created within the application when comparing hash values with the RDS.

Additional functionality should be researched and implemented in future versions of this software application. The ability to perform text analysis and queries would be valuable features to a computer forensic investigator. Zero-byte files should also be closely scrutinized for data streams and other concealed information. Additionally, files characterized as unknown by the software application could be red flagged or classified further by the investigator based upon whether the file is of evidentiary value.

This software application currently only identifies those files visible and hidden by the operating system. It relies upon other forensic utilities that recover deleted files. Future research could be performed so that a file recovery feature could be integrated into the analysis provided by this application, thus doing away with the need for a separate computer forensics application.

The most interesting research that would elevate the usefulness of this software application would be to implement a web-based service for accessing the most up-to-date hash values and file profiles from the RDS. Such a web service would allow the software application to search the RDS for matches, for example via an XML query consisting of the MD5 hashes of files on the media being examined. The web service would return an XML document consisting of file profiles matching the query. To maintain the traceability and validity of hash values, such a web service must only be implemented in cooperation with NIST.

6.5 Final Conclusions

This research explores the use of the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) database in a hash filtering software application that is simple, streamlined, standalone and for use by the general public. It is the author's hope that the ideas contained within this research will be furthered by the students of computer forensics in the future.

Bibliography

- [1]. Thornton, John I. *Special Report: The Beginnings of Forensic Science*. World Book Online Reference Center. 2004. World Book, Inc. 20 Jan. 2004.
<<http://www.aolsvc.worldbook.aol.com/wb/Media?id=sr399014&st=Forensics>>.
- [2]. Noblett, Michael G., Pollitt, Mark M., and Presley, Lawrence A. *Recovering and Examining Computer Forensic Evidence*. Forensic Science Communications, October 2000, Volume 2, Number 4.
- [3]. Kruse, Warren G. and Heiser, Jay G. *Computer Forensics: Incident Response Essentials*, 2002.
- [4]. CERT Coordination Center Website. <<http://www.cert.org>>, 2004.
- [5]. Johansson, Christian. *Computer Forensic Text Analysis with Open Source Software*. Master Thesis, Blekinge Institute of Technology, 2003.
- [6]. Stallard, Tye Brown. *Automated Analysis for Digital Forensic Science*. Master Thesis, University of California, Davis, 2002.
- [7]. Garfinkel, Simson L. *A Web Service for File Fingerprints: The Goods, the Bads, and the Unknowns*, 2003.
- [8]. KnownGoods Website. <<http://www.knowngoods.org>>, 2004.
- [9]. HashKeeper Website. <<http://www.hashkeeper.org>>, 2004.
- [10]. EnCase Website. <<http://www.guidancesoftware.com/products/EnCaseForensic/index.shtm>>, 2004.
- [11]. ILook Investigator Website. <<http://www.ilook-forensics.org>>, 2004.
- [12]. The Sleuth Kit Website. <<http://www.sleuthkit.org/sleuthkit/index.php>>, 2004.
- [13]. HashDig Website. <<http://ftimes.sourceforge.net/FTimes/HashDig.shtml>>, 2004.

- [14]. CERT Coordination Center, *How the FBI Investigates Computer Crime*,
<www.cert.org/tech_tips/FBI_investigates_crime.html>
- [15]. Coleman, Ronald D. *Computer Forensics Roundup*, January 2002.
- [16]. *Chain of Custody*., <www.peace-officers.com/content/glossary/def-chain.shtml>,
2004
- [17]. SafeBack Website. <<http://www.forensics-intl.com/safeback.html>>, 2004
- [18]. Mares and Company, LLC. *Data Integrity: How To Authenticate Your Electronic Records*, May 2003.
- [19]. Tripwire Website. <www.tripwire.com>, 2004.
- [20]. Robbins, Judd. *An Explanation of Computer Forensics*, 2004.
- [21]. *Webster's Revised Unabridged Dictionary*, 1996.
- [22]. Boland, Tim and Fisher, Gary. *Selection of Hashing Algorithms*, June 30, 2000.
- [23]. Menzies, A. et.al. *Handbook of Applied Cryptography*, CRC Press, Inc., 1997.
- [24]. Schneier, Bruce. *Applied Cryptography*, 2nd, New York: John Wiley & Sons, 1996.
- [25]. RSA Security, Inc. *Frequently Asked Questions About Today's Cryptography*, 2002.
- [26]. Robshaw. *RSA Laboratories Bulletin No. 4*, Nov 12, 1996.
- [27]. Harrison, John S. *MD5, md5sum and Related Topics*, Dec. 2000,
<<http://hills.ccsf.org/~jharri01/project.html>>.
- [28]. International Organization for Standardization, *Information Processing Systems--Data Communication High-Level Data Link Control Procedure--Frame Structure*, IS 3309, October 1984, 3rd Edition.
- [29]. Rivest, Ronald L. *The MD5 Message-Digest Algorithm*,
<<http://www.ietf.org/rfc/rfc1321.txt>>, 1992.

- [30]. Touch, Joseph E. *Performance Analysis of MD5*, 2004.
- [31]. TechBeat. *Taking the Byte Out*, Winter 2002
- [32]. NIST. *National Software Reference Library (NSRL) Technical Report*, 2004.
- [33]. NIST. *ITL Bulletin*, November 2001.
- [34]. NIST. *NSRL Website - Library Contents*
<http://www.nsrl.nist.gov/Library_Contents.htm>, 2004.
- [35]. NIST. *Special Database 28: National Software Reference Library (NSRL)*,
<<http://www.nist.gov/srd/nistsd28.htm>>
- [36]. NIST. *NSRL Website – Project Overview*
<http://www.nsrl.nist.gov/Project_Overview.htm>, 2004.
- [37]. NIST. *Data Formats of the NSRL Reference Data Set (RDS) Distribution*, 2004.
- [38]. NIST. *NSRL Website – RDS Notes*, <http://www.nsrl.nist.gov/RDS_Notes.htm>,
2004.
- [39]. NIST. *NSRL Website*, <<http://www.nsrl.nist.gov>>, 2004.

Appendices

Appendix A: MD5 Hashing Algorithm Reference Implementation

This appendix contains the following files taken from RSAREF: A Cryptographic Toolkit for Privacy-Enhanced Mail:

- global.h -- global header file
- md5.h -- header file for MD5
- md5c.c -- source code for MD5

The appendix also includes the following file:

- mddriver.c -- test driver for MD2, MD4 and MD5

The implementation is portable and should work on many different platforms. However, it is not difficult to optimize the implementation on particular platforms, an exercise left to the reader. For example, on "little-endian" platforms where the lowest-addressed byte in a 32-bit word is the least significant and there are no alignment restrictions, the call to Decode in MD5Transform can be replaced with a typecast.

```

global.h

/* GLOBAL.H - RSAREF types and constants
 */

/* PROTOTYPES should be set to one if and only if the compiler supports
function argument prototyping.
The following makes PROTOTYPES default to 0 if it has not already
been defined with C compiler flags.
 */

#ifdef PROTOTYPES
#define PROTOTYPES 0
#endif

/* POINTER defines a generic pointer type */
typedef unsigned char *POINTER;

/* UINT2 defines a two byte word */
typedef unsigned short int UINT2;

/* UINT4 defines a four byte word */
typedef unsigned long int UINT4;

/* PROTO_LIST is defined depending on how PROTOTYPES is defined above.
If using PROTOTYPES, then PROTO_LIST returns the list, otherwise it
returns an empty list.
 */
#ifdef PROTOTYPES
#define PROTO_LIST(list) list
#else
#define PROTO_LIST(list) ()
#endif

```


md5.h

```
/* MD5.H - header file for MD5C.C
 */
```

```
/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.
```

```
License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.
```

```
License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.
```

```
RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.
```

```
These notices must be retained in any copies of any part of this
documentation and/or software.
```

```
 */

/* MD5 context. */
typedef struct {
    UINT4 state[4];           /* state (ABCD) */
    UINT4 count[2];          /* number of bits, modulo 2^64 (lsb first) */
    unsigned char buffer[64]; /* input buffer */
} MD5_CTX;

void MD5Init PROTO_LIST ((MD5_CTX *));
void MD5Update PROTO_LIST
    ((MD5_CTX *, unsigned char *, unsigned int));
void MD5Final PROTO_LIST ((unsigned char [16], MD5_CTX *));
```

md5c.c

```
/* MD5C.C - RSA Data Security, Inc., MD5 message-digest algorithm
*/
```

```
/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.
```

```
License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.
```

```
License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.
```

```
RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.
```

```
These notices must be retained in any copies of any part of this
documentation and/or software.
```

```
*/
```

```
#include "global.h"
#include "md5.h"
```

```
/* Constants for MD5Transform routine.
*/
```

```
#define S11 7
#define S12 12
#define S13 17
#define S14 22
#define S21 5
#define S22 9
#define S23 14
#define S24 20
#define S31 4
#define S32 11
#define S33 16
#define S34 23
#define S41 6
#define S42 10
#define S43 15
#define S44 21
```

```
static void MD5Transform PROTO_LIST ((UINT4 [4], unsigned char [64]));
static void Encode PROTO_LIST
((unsigned char *, UINT4 *, unsigned int));
static void Decode PROTO_LIST
((UINT4 *, unsigned char *, unsigned int));
static void MD5_memcpy PROTO_LIST ((POINTER, POINTER, unsigned int));
static void MD5_memset PROTO_LIST ((POINTER, int, unsigned int));
```

```

static unsigned char PADDING[64] = {
    0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G, H and I are basic MD5 functions.
 */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
    (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    \
    (a) += (b); \
}
#define GG(a, b, c, d, x, s, ac) { \
    (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}
#define HH(a, b, c, d, x, s, ac) { \
    (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}
#define II(a, b, c, d, x, s, ac) { \
    (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}

/* MD5 initialization. Begins an MD5 operation, writing a new context.
 */
void MD5Init (context)
MD5_CTX *context; /* context */
{
    context->count[0] = context->count[1] = 0;
    /* Load magic initialization constants.
 */
    context->state[0] = 0x67452301;
    context->state[1] = 0xefcdab89;
    context->state[2] = 0x98badcfe;
    context->state[3] = 0x10325476;
}

/* MD5 block update operation. Continues an MD5 message-digest

```

```

    operation, processing another message block, and updating the
    context.
*/
void MD5Update (context, input, inputLen)
MD5_CTX *context; /* context */
unsigned char *input; /* input block */
unsigned int inputLen; /* length of input block */
{
    unsigned int i, index, partLen;

    /* Compute number of bytes mod 64 */
    index = (unsigned int)((context->count[0] >> 3) & 0x3F);

    /* Update number of bits */
    if ((context->count[0] += ((UINT4)inputLen << 3))
        < ((UINT4)inputLen << 3))
        context->count[1]++;
    context->count[1] += ((UINT4)inputLen >> 29);

    partLen = 64 - index;

    /* Transform as many times as possible.
*/
    if (inputLen >= partLen) {
        MD5_memcpy
            ((POINTER)&context->buffer[index], (POINTER)input, partLen);
        MD5Transform (context->state, context->buffer);

        for (i = partLen; i + 63 < inputLen; i += 64)
            MD5Transform (context->state, &input[i]);

        index = 0;
    }
    else
        i = 0;

    /* Buffer remaining input */
    MD5_memcpy
        ((POINTER)&context->buffer[index], (POINTER)&input[i],
        inputLen-i);
}

/* MD5 finalization. Ends an MD5 message-digest operation, writing the
the message digest and zeroizing the context.
*/
void MD5Final (digest, context)
unsigned char digest[16]; /* message digest */
MD5_CTX *context; /* context */
{
    unsigned char bits[8];
    unsigned int index, padLen;

    /* Save number of bits */
    Encode (bits, context->count, 8);

    /* Pad out to 56 mod 64.

```

```

*/
index = (unsigned int)((context->count[0] >> 3) & 0x3f);
padLen = (index < 56) ? (56 - index) : (120 - index);
MD5Update (context, PADDING, padLen);

/* Append length (before padding) */
MD5Update (context, bits, 8);

/* Store state in digest */
Encode (digest, context->state, 16);

/* Zeroize sensitive information.
*/
MD5_memset ((POINTER)context, 0, sizeof (*context));
}

/* MD5 basic transformation. Transforms state based on block.
*/
static void MD5Transform (state, block)
UINT4 state[4];
unsigned char block[64];
{
    UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

    Decode (x, block, 64);

    /* Round 1 */
    FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
    FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
    FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
    FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
    FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
    FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
    FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
    FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
    FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
    FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
    FF (c, d, a, b, x[10], S13, 0xfffff5bb1); /* 11 */
    FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
    FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
    FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
    FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
    FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

    /* Round 2 */
    GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
    GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
    GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
    GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
    GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
    GG (d, a, b, c, x[10], S22, 0x2441453); /* 22 */
    GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
    GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
    GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
    GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
    GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */

```

```

GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH (a, b, c, d, x[ 5], S31, 0xffffa3942); /* 33 */
HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH (d, a, b, c, x[ 0], S32, 0xeaal27fa); /* 42 */
HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH (b, c, d, a, x[ 6], S34, 0x4881d05); /* 44 */
HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */

/* Round 4 */
II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */

state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;

/* Zeroize sensitive information.

*/
MD5_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len is
a multiple of 4.
*/
static void Encode (output, input, len)

```

```

unsigned char *output;
UINT4 *input;
unsigned int len;
{
    unsigned int i, j;

    for (i = 0, j = 0; j < len; i++, j += 4) {
        output[j] = (unsigned char)(input[i] & 0xff);
        output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
        output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
        output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
    }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len is
   a multiple of 4.
*/
static void Decode (output, input, len)
UINT4 *output;
unsigned char *input;
unsigned int len;
{
    unsigned int i, j;

    for (i = 0, j = 0; j < len; i++, j += 4)
        output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
            (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
*/

static void MD5_memcpy (output, input, len)
POINTER output;
POINTER input;
unsigned int len;
{
    unsigned int i;

    for (i = 0; i < len; i++)

        output[i] = input[i];
}

/* Note: Replace "for loop" with standard memset if possible.
*/
static void MD5_memset (output, value, len)
POINTER output;
int value;
unsigned int len;
{
    unsigned int i;

    for (i = 0; i < len; i++)
        ((char *)output)[i] = (char)value;
}
mddriver.c

```

```

/* MDDRIVER.C - test driver for MD2, MD4 and MD5
*/

/* Copyright (C) 1990-2, RSA Data Security, Inc. Created 1990. All
rights reserved.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
*/

/* The following makes MD default to MD5 if it has not already been
defined with C compiler flags.
*/
#ifdef MD
#define MD MD5
#endif

#include <stdio.h>
#include <time.h>
#include <string.h>
#include "global.h"
#if MD == 2
#include "md2.h"
#endif
#if MD == 4

#include "md4.h"
#endif
#if MD == 5
#include "md5.h"
#endif

/* Length of test block, number of test blocks.
*/
#define TEST_BLOCK_LEN 1000
#define TEST_BLOCK_COUNT 1000

static void MDString PROTO_LIST ((char *));
static void MDTimeTrial PROTO_LIST ((void));
static void MDTestSuite PROTO_LIST ((void));
static void MDFile PROTO_LIST ((char *));
static void MDFilter PROTO_LIST ((void));
static void MDPrint PROTO_LIST ((unsigned char [16]));

#if MD == 2
#define MD_CTX MD2_CTX
#define MDInit MD2Init
#define MDUpdate MD2Update
#define MDFinal MD2Final
#endif
#if MD == 4
#define MD_CTX MD4_CTX

```



```

#define MDInit MD4Init
#define MDUpdate MD4Update
#define MDFinal MD4Final
#endif
#if MD == 5
#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final
#endif

/* Main driver.

Arguments (may be any combination):
  -sstring - digests string
  -t       - runs time trial
  -x       - runs test script
  filename - digests file
  (none)   - digests standard input
*/
int main (argc, argv)
int argc;

char *argv[];
{
    int i;

    if (argc > 1)
    for (i = 1; i < argc; i++)
        if (argv[i][0] == '-' && argv[i][1] == 's')
            MDString (argv[i] + 2);
        else if (strcmp (argv[i], "-t") == 0)
            MDTimeTrial ();
        else if (strcmp (argv[i], "-x") == 0)
            MDTestSuite ();
        else
            MDFile (argv[i]);
    else
    MDFilter ();

    return (0);
}

/* Digests a string and prints the result.
*/
static void MDString (string)
char *string;
{
    MD_CTX context;
    unsigned char digest[16];
    unsigned int len = strlen (string);

    MDInit (&context);
    MDUpdate (&context, string, len);
    MDFinal (digest, &context);

    printf ("MD%d (\"%s\") = ", MD, string);

```

```

    MDPrint (digest);
    printf ("\n");
}

/* Measures the time to digest TEST_BLOCK_COUNT TEST_BLOCK_LEN-byte
   blocks.
   */
static void MDTimeTrial ()
{
    MD_CTX context;
    time_t endTime, startTime;
    unsigned char block[TEST_BLOCK_LEN], digest[16];
    unsigned int i;

    printf
    ("MD%d time trial. Digesting %d %d-byte blocks ...", MD,
     TEST_BLOCK_LEN, TEST_BLOCK_COUNT);

    /* Initialize block */
    for (i = 0; i < TEST_BLOCK_LEN; i++)
        block[i] = (unsigned char)(i & 0xff);

    /* Start timer */
    time (&startTime);

    /* Digest blocks */
    MDInit (&context);
    for (i = 0; i < TEST_BLOCK_COUNT; i++)
        MDUpdate (&context, block, TEST_BLOCK_LEN);
    MDFinal (digest, &context);

    /* Stop timer */
    time (&endTime);

    printf (" done\n");
    printf ("Digest = ");
    MDPrint (digest);
    printf ("\nTime = %ld seconds\n", (long)(endTime-startTime));
    printf
    ("Speed = %ld bytes/second\n",
     (long)TEST_BLOCK_LEN * (long)TEST_BLOCK_COUNT/(endTime-startTime));
}

/* Digests a reference suite of strings and prints the results.
   */
static void MDTestSuite ()
{
    printf ("MD%d test suite:\n", MD);

    MDString ("");
    MDString ("a");
    MDString ("abc");
    MDString ("message digest");
    MDString ("abcdefghijklmnopqrstuvwxyz");
    MDString
    ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy0123456789");
    MDString

```

```

    ("1234567890123456789012345678901234567890\
1234567890123456789012345678901234567890");
}

/* Digests a file and prints the result.
*/
static void MDFile (filename)
char *filename;
{
    FILE *file;
    MD_CTX context;
    int len;
    unsigned char buffer[1024], digest[16];

    if ((file = fopen (filename, "rb")) == NULL)
        printf ("%s can't be opened\n", filename);

    else {
        MDInit (&context);
        while (len = fread (buffer, 1, 1024, file))
            MDUpdate (&context, buffer, len);
        MDFinal (digest, &context);

        fclose (file);

        printf ("MD%d (%s) = ", MD, filename);
        MDPrint (digest);
        printf ("\n");
    }
}

/* Digests the standard input and prints the result.
*/
static void MDFilter ()
{
    MD_CTX context;
    int len;
    unsigned char buffer[16], digest[16];

    MDInit (&context);
    while (len = fread (buffer, 1, 16, stdin))
        MDUpdate (&context, buffer, len);
    MDFinal (digest, &context);

    MDPrint (digest);
    printf ("\n");
}

/* Prints a message digest in hexadecimal.
*/
static void MDPrint (digest)
unsigned char digest[16];
{
    unsigned int i;

```

```
    for (i = 0; i < 16; i++)
    printf ("%02x", digest[i]);
}
```

A.5 Test suite

The MD5 test suite (driver option "-x") should print the following results:

MD5 test suite:

```
MD5 ("") = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789")
=
d174ab98d277d9f5a5611c2c9f419d9f
MD5
("123456789012345678901234567890123456789012345678901234567890123456
78901234567890") = 57edf4a22be3c955ac49da2e2107b67a
```

Security Considerations

The level of security discussed in this memo is considered to be sufficient for implementing very high security hybrid digital-signature schemes based on MD5 and a public-key cryptosystem.

Appendix B: Project Source Code

mainform.frm

```
Option Explicit
```

```
Private getdir As String
```

```
Private Declare Function SendMessageArray Lib "user32" Alias  
"SendMessageA" _  
    (ByVal hWnd As Long, ByVal wParam As Long, _  
    ByVal lParam As Long, lParam As Any) As Long  
Const LB_SETTABSTOPS = &H192
```

```
Dim filenum As Integer  
    Dim matcha As String  
    Dim matchb As String  
    Dim matchc As String  
    Dim matchd As String  
    Dim matchfound As String
```

```
Private Sub Command1_Click()  
    getdir = BrowseForFolder(Me, "Select A Directory", "c:\")  
    If Len(getdir) = 0 Then Exit Sub  
    Screen.MousePointer = vbHourglass  
    filenum = FreeFile  
    Print #1, Format$(Date, "mm/dd/yyyy") & " " & Format$(Hour(Now),  
"00") & ":" & Format$(Minute(Now), "00") & ":" & Format$(Second(Now),  
"00") & " " & "Disk Analysis of " & getdir & " created. Filename: " &  
Format$(Date, "mddyyyy") & Format$(Hour(Now), "00") &  
Format$(Minute(Now), "00") & Format$(Second(Now), "00") & ".txt"  
    Open ("diskanalysis\" & Format$(Date, "mddyyyy") &  
Format$(Hour(Now), "00") & Format$(Minute(Now), "00") &  
Format$(Second(Now), "00") & ".txt") For Output As filenum  
    Write #filenum, "Path", "MD5 Hash"  
    ShowAllFiles getdir  
    Screen.MousePointer = vbNormal  
    StatusBar1.Panels(1).Text = "Ready"  
End Sub
```

```
Private Sub ShowAllFiles(ByVal sPath As String)  
    Dim fso As New FileSystemObject  
    Dim fil As File  
    Dim fol As Folder  
    Dim sub1 As Folder  
    Dim md5hash As String  
    Dim listobj As ListItem  
    Set fol = fso.GetFolder(sPath)  
    For Each fil In fol.Files  
        StatusBar1.Panels(1).Text = "Processing: " & fol & "\" & fil.Name  
        On Error Resume Next  
        md5hash = Hashmyfile(sPath & "\" & fil.Name)  
        Set listobj = ListView1.ListItems.add(, , fil.Name)  
        listobj.SubItems(1) = fil.Size  
        listobj.SubItems(2) = fil.Type
```

```

listobj.SubItems(3) = fil.DateCreated
listobj.SubItems(4) = fil.DateLastAccessed
listobj.SubItems(5) = fil.DateLastModified
listobj.SubItems(6) = fil.Path
listobj.SubItems(7) = fil.Attributes
listobj.SubItems(8) = md5hash
comparehash (md5hash)
listobj.SubItems(9) = matcha
listobj.SubItems(10) = matchb
listobj.SubItems(11) = matchc
listobj.SubItems(12) = matchd
listobj.SubItems(13) = matchfound
Write #filenum, (fol & "\" & fil.Name), md5hash
Next
For Each sub1 In fol.SubFolders
    ShowAllFiles sub1.Path
Next
Set fil = Nothing
Set sub1 = Nothing
Set fol = Nothing
Set fso = Nothing
End Sub

Private Sub Command2_Click()
    Shell "explorer.exe " & ListView1.SelectedItem.SubItems(6)
End Sub

Private Sub Command3_Click()
    If Command3.Caption = "Hide Matches" Then
        Command3.Caption = "Show Matches"
    Else
        If Command3.Caption = "Show Matches" Then
            Command3.Caption = "Hide Matches"
        End If
    End If
End Sub

Private Sub Form_Load()
    Open ("logs\" & Format$(Date, "mmddyyyy") & Format$(Hour(Now), "00")
    & Format$(Minute(Now), "00") & Format$(Second(Now), "00") & ".txt") For
    Output As #1
    Print #1, Format$(Date, "mm/dd/yyyy") & " " & Format$(Hour(Now),
    "00") & ":" & Format$(Minute(Now), "00") & ":" & Format$(Second(Now),
    "00") & " " & "Log File Created"
    StatusBar1.Panels(1).Text = "Ready"
    With Me.ListView1
        .View = lvwReport
        .HideSelection = False
        .GridLines = True
        .LabelEdit = lvwManual
        .ColumnHeaders.add , "name", "Name", 4200
        .ColumnHeaders.add , "size", "Size", 0
        .ColumnHeaders.add , "type", "Type", 0
        .ColumnHeaders.add , "datecreated", "Date Created", 0
        .ColumnHeaders.add , "datelastaccessed", "Date Last Accessed", 0
        .ColumnHeaders.add , "datelastmodified", "Date Last Modified", 0
        .ColumnHeaders.add , "path", "Path", 0
    End With
End Sub

```

```

        .ColumnHeaders.add , "attributes", "Attributes", 0
        .ColumnHeaders.add , "md5hash", "MD5 Hash", 0
        .ColumnHeaders.add , "matcha", "RDS_A", 0
        .ColumnHeaders.add , "matchb", "RDS_B", 0
        .ColumnHeaders.add , "matchc", "RDS_C", 0
        .ColumnHeaders.add , "matchd", "RDS_D", 0
        .ColumnHeaders.add , "matchfound", "Match", 800
    End With
    Set Conn1 = New ADODB.Connection
    Set Rset1 = New ADODB.Recordset
    Set Conn2 = New ADODB.Connection
    Set Rset2 = New ADODB.Recordset
    Set Conn3 = New ADODB.Connection
    Set Rset3 = New ADODB.Recordset
    Set Conn4 = New ADODB.Connection
    Set Rset4 = New ADODB.Recordset
    With Conn1
        .Provider = "Microsoft.Jet.OLEDB.4.0"
        .ConnectionString = App.Path & "\RDS_A.mdb"
        .Open
    End With
    With Conn2
        .Provider = "Microsoft.Jet.OLEDB.4.0"
        .ConnectionString = App.Path & "\RDS_B.mdb"
        .Open
    End With
    With Conn3
        .Provider = "Microsoft.Jet.OLEDB.4.0"
        .ConnectionString = App.Path & "\RDS_C.mdb"
        .Open
    End With
    With Conn4
        .Provider = "Microsoft.Jet.OLEDB.4.0"
        .ConnectionString = App.Path & "\RDS_D.mdb"
        .Open
    End With
    Rset1.CursorLocation = adUseServer
    Rset2.CursorLocation = adUseServer
    Rset3.CursorLocation = adUseServer
    Rset4.CursorLocation = adUseServer
    Rset1.Open "NSRFile", Conn1, adOpenKeyset, adLockReadOnly,
adCmdTableDirect
    Rset2.Open "NSRFile", Conn2, adOpenKeyset, adLockReadOnly,
adCmdTableDirect
    Rset3.Open "NSRFile", Conn3, adOpenKeyset, adLockReadOnly,
adCmdTableDirect
    Rset4.Open "NSRFile", Conn4, adOpenKeyset, adLockReadOnly,
adCmdTableDirect
    Rset1.Index = "MD5"
    Rset2.Index = "MD5"
    Rset3.Index = "MD5"
    Rset4.Index = "MD5"
    Dim LBTAB(1) As Long
    LBTAB(0) = 30
    LBTAB(1) = 60
End Sub

```

```

Private Sub ListView1_Click()
    Dim lsFileName As String
    Dim liX As Integer
    Dim HexText As String
    Dim attributeval As Integer
    Dim attributetext As String
    attributetext = ""
    outpath = ListView1.SelectedItem.SubItems(6)
    attributeval = ListView1.SelectedItem.SubItems(7)
    If attributeval And &H20 Then
        attributetext = attributetext & "Archive "
    End If
    If attributeval And &H800 Then
        attributetext = attributetext & "Compressed "
    End If
    If attributeval And &H10 Then
        attributetext = attributetext & "Directory "
    End If
    If attributeval And &H2 Then
        attributetext = attributetext & "Hidden "
    End If
    If attributeval And &H80 Then
        attributetext = attributetext & "Normal "
    End If
    If attributeval And &H1 Then
        attributetext = attributetext & "ReadOnly "
    End If
    If attributeval And &H4 Then
        attributetext = attributetext & "System "
    End If
    If attributeval And &H100 Then
        attributetext = attributetext & "Temporary "
    End If
    outattributes = attributetext
    outsize = ListView1.SelectedItem.SubItems(1)
    outtype = ListView1.SelectedItem.SubItems(2)
    outcreated = ListView1.SelectedItem.SubItems(3)
    outaccessed = ListView1.SelectedItem.SubItems(4)
    outmodified = ListView1.SelectedItem.SubItems(5)
    outhash = ListView1.SelectedItem.SubItems(8)
    outa = ListView1.SelectedItem.SubItems(9)
    outb = ListView1.SelectedItem.SubItems(10)
    outc = ListView1.SelectedItem.SubItems(11)
    outd = ListView1.SelectedItem.SubItems(12)
    On Error GoTo ErrAboardLoad
    lsFileName = ListView1.SelectedItem.SubItems(6)
    On Error GoTo 0
    LstHexView.Visible = False
    On Error GoTo ErrLoadFile
    liX = FreeFile
    Open lsFileName For Binary Access Read As #liX
    HexText = Space$(LOF(liX) + 16)
    Get #liX, , HexText
    Close #liX
    On Error GoTo 0
    LstHexView.ListItems.Clear
    FileToHex (HexText)

```



```

    LstHexView.Visible = True
    GoTo ErrCont
ErrAboardLoad:
    GoTo ErrCont
ErrLoadFile:
    MsgBox "File is too large..."
    GoTo ErrCont
ErrCont:
    On Error GoTo 0
End Sub

Private Sub comparehash(hashval As String)
    Dim fname As String
    Dim outhash As String
    matcha = ""
    matchb = ""
    matchc = ""
    matchd = ""
    matchfound = ""
    Rset1.Seek hashval
    If Not Rset1.EOF Then
        fname = Rset1!FileName
        outhash = Rset1!MD5
        matcha = outhash & " " & fname
        matchfound = "X"
    Else
        matcha = ""
    End If
    Rset2.Seek hashval
    If Not Rset2.EOF Then
        fname = Rset2!FileName
        outhash = Rset2!MD5
        matchb = outhash & " " & fname
        matchfound = "X"
    Else
        matchb = ""
    End If
    Rset3.Seek hashval
    If Not Rset3.EOF Then
        fname = Rset3!FileName
        outhash = Rset3!MD5
        matchc = outhash & " " & fname
        matchfound = "X"
    Else
        matchc = ""
    End If
    Rset4.Seek hashval
    If Not Rset4.EOF Then
        fname = Rset4!FileName
        outhash = Rset4!MD5
        matchd = outhash & " " & fname
        matchfound = "X"
    Else
        matchd = ""
    End If
End Sub

```

hexedit.bas

```
` Open source code adapted within this module made freely available by  
` Michael Werren at http://www.planetsourcecode.com/vb/scripts/  
` ShowCode.asp?txtCodeId=13898&lngWId=1
```

```
Option Explicit
```

```
Function WriteHex(Cnt As Integer, Val As String) As String  
    WriteHex = String(Cnt, Val)  
End Function
```

```
Sub AddHexLine(HexIndex As String, HexText As String, AsciiText As  
String)  
    Dim itmX As ListItem  
    Set itmX = Form1.LstHexView.ListItems.add  
    itmX.Text = HexIndex  
    itmX.SubItems(1) = HexText  
    itmX.SubItems(2) = AsciiText  
End Sub
```

```
Sub FileToHex(TransText As String)  
    Dim HexText As String  
    Dim lsVal As String  
    Dim lsOrgText As String  
    Dim lsHexCode As String  
    Dim lsHexLine As String  
    Dim lsHexIndex As String  
    Dim liHexIndex As Long  
    Dim liVal As Integer  
    Dim liPointer As Integer  
    Dim liX As Long  
    Dim liProcent As Integer  
    Dim liProcentOld As Integer  
    HexText = TransText  
    Screen.MousePointer = vbHourglass  
    liPointer = 1  
    liHexIndex = 0  
    For liX = 1 To Len(HexText)  
        If liPointer <= 16 Then  
            liPointer = liPointer + 1  
            lsVal = Mid(HexText, liX, 1)  
            liVal = Asc(lsVal)  
            lsHexCode = Hex(liVal)  
            If Len(lsHexCode) < 2 Then  
                lsHexCode = "0" + lsHexCode  
            End If  
            If liPointer <= 16 Then  
                If liPointer <> 9 Then  
                    lsHexLine = lsHexLine + lsHexCode + "."  
                Else  
                    lsHexLine = lsHexLine + lsHexCode + " "  
                End If  
            Else  
                lsHexLine = lsHexLine + lsHexCode  
                ' Enum the translation in procent
```

```

        liProcentOld = liProcent
        liProcent = liX * 100 \ Len(HexText)
        If liProcent <> liProcentOld Then
            DispInfo "Translate the file " + Str(liProcent) + "%"
        End If
    End If
    If Asc(lsVal) = 0 Then
        lsOrgText = lsOrgText + "."
    Else
        lsOrgText = lsOrgText + lsVal
    End If
Else
    lsHexIndex = WriteHex(8 - Len(Hex(liHexIndex)), "0") +
Hex(liHexIndex)
    AddHexLine lsHexIndex, lsHexLine, lsOrgText
    liPointer = 1
    liHexIndex = liHexIndex + 16
    lsHexLine = ""
    lsOrgText = ""
    liX = liX - 1
End If
Next liX
If lsHexLine <> "" Then
    If Mid(lsHexLine, Len(lsHexLine), 1) = "." Then
        lsHexLine = Mid(lsHexLine, 1, Len(lsHexLine) - 1)
    End If
    lsHexIndex = WriteHex(8 - Len(Hex(liHexIndex)), "0") +
Hex(liHexIndex)
    AddHexLine lsHexIndex, lsHexLine, lsOrgText
End If
DispInfo ""
Screen.MousePointer = vbDefault
End Sub

Sub DispInfo(Text As String)
    DoEvents
End Sub

```

md5file.bas

```
'      The MD5 algorithm is defined in RFC1321.
'
'      The basic C code implementing the algorithm is derived
'      from that in the RFC and is covered by the following
'      copyright: Copyright (C) 1991-2, RSA Data Security, Inc.
'      Created 1991. All rights reserved.
'
'      License to copy and use this software is granted provided
'      that it is identified as the "RSA Data Security, Inc. MD5
'      Message-Digest Algorithm" in all material mentioning or
'      referencing this software or this function.
'
'      License is also granted to make and use derivative works
'      provided that such works are identified as "derived from
'      the RSA Data Security, Inc. MD5 Message-Digest Algorithm"
'      in all material mentioning or referencing the derived
'      work.
'
'      RSA Data Security, Inc. makes no representations
'      concerning either the merchantability of this software or
'      the suitability of this software for any particular
'      purpose. It is provided "as is" without express or implied
'      warranty of any kind.
'
'      These notices must be retained in any copies of any part
'      of this documentation and/or software.

Private Declare Sub MDFile Lib "md5file.dll" (ByVal outmd As String,
ByVal outstring As String)

Public Function Hashmyfile(outmd As String) As String
    Dim outstring As String * 32
    outstring = Space(32)
    MDFile outmd, outstring
    Hashmyfile = r
End Function
```

filesearch.bas

```
` Open source code adapted within this module made freely available by  
` Serge Lachapelle at http://www.planetsourcecode.com/vb/scripts/  
` ShowCode.asp?txtCodeId=49326&lngWid=-10
```

```
Option Explicit
```

```
Private Const BIF_STATUSTEXT = &H4&  
Private Const BIF_RETURNONLYFSDIRS = 1  
Private Const BIF_DONTGOBELOWDOMAIN = 2  
Private Const MAX_PATH = 260
```

```
Private Const WM_USER = &H400  
Private Const BFFM_INITIALIZED = 1  
Private Const BFFM_SELCHANGED = 2  
Private Const BFFM_SETSTATUSTEXT = (WM_USER + 100)  
Private Const BFFM_SETSELECTION = (WM_USER + 102)
```

```
Private Declare Function SendMessage Lib "user32" Alias "SendMessageA"  
(ByVal hWnd As Long, ByVal wParam As Long, ByVal lParam As Long, ByVal  
lParam As String) As Long  
Private Declare Function SHBrowseForFolder Lib "shell32" (lpbi As  
BrowseInfo) As Long  
Private Declare Function SHGetPathFromIDList Lib "shell32" (ByVal  
pidList As Long, ByVal lpBuffer As String) As Long  
Private Declare Function lstrcat Lib "kernel32" Alias "lstrcatA" (ByVal  
lpString1 As String, ByVal lpString2 As String) As Long
```

```
Private Type BrowseInfo  
    hWndOwner As Long  
    pIDList As Long  
    pszDisplayName As Long  
    lpszTitle As Long  
    ulFlags As Long  
    lpfnCallback As Long  
    lParam As Long  
    iImage As Long  
End Type
```

```
Private m_CurrentDirectory As String 'The current directory  
,
```

```
Public Function BrowseForFolder(owner As Form, Title As String,  
StartDir As String) As String  
    'Opens a Treeview control that displays the directories in a computer
```

```
    Dim lpIDList As Long  
    Dim szTitle As String  
    Dim sBuffer As String  
    Dim tBrowseInfo As BrowseInfo  
    m_CurrentDirectory = StartDir & vbNullChar
```

```
    szTitle = Title  
    With tBrowseInfo  
        .hWndOwner = owner.hWnd
```

```

        .lp.szTitle = lstrcat(szTitle, "")
        .ulFlags = BIF_RETURNONLYFSDIRS + BIF_DONTGOBELOWDOMAIN +
BIF_STATUSTEXT
        .lpfnCallback = GetAddressofFunction(AddressOf BrowseCallbackProc)
'get address of function.
    End With

    lpIDList = SHBrowseForFolder(tBrowseInfo)
    If (lpIDList) Then
        sBuffer = Space(MAX_PATH)
        SHGetPathFromIDList lpIDList, sBuffer
        sBuffer = Left(sBuffer, InStr(sBuffer, vbNullChar) - 1)
        BrowseForFolder = sBuffer
    Else
        BrowseForFolder = ""
    End If

End Function

Private Function BrowseCallbackProc(ByVal hWnd As Long, ByVal uMsg As
Long, ByVal lp As Long, ByVal pData As Long) As Long

    Dim lpIDList As Long
    Dim ret As Long
    Dim sBuffer As String

    On Error Resume Next 'Suggested by MS to prevent an error from
        'propagating back into the calling process.

    Select Case uMsg

        Case BFFM_INITIALIZED
            Call SendMessage(hWnd, BFFM_SETSELECTION, 1, m_CurrentDirectory)

        Case BFFM_SELCHANGED
            sBuffer = Space(MAX_PATH)

            ret = SHGetPathFromIDList(lp, sBuffer)
            If ret = 1 Then
                Call SendMessage(hWnd, BFFM_SETSTATUSTEXT, 0, sBuffer)
            End If

    End Select

    BrowseCallbackProc = 0

End Function

' This function allows you to assign a function pointer to a variable.
Private Function GetAddressofFunction(add As Long) As Long
    GetAddressofFunction = add
End Function

```

dbconnect.bas

```
Public Conn1 As ADODB.Connection  
Public Rset1 As ADODB.Recordset
```

```
Public Conn2 As ADODB.Connection  
Public Rset2 As ADODB.Recordset
```

```
Public Conn3 As ADODB.Connection  
Public Rset3 As ADODB.Recordset
```

```
Public Conn4 As ADODB.Connection  
Public Rset4 As ADODB.Recordset
```