NTNU
Norwegian University of
Science and Technology

Forensic Key Discovery and Identification

Finding Cryptographic Keys in Physical Memory

Carsten Maartmann-Moe

Master of Science in Communication Technology
Submission date: June 2008
Supervisor: Svein Johan Knapskog, ITEM
Co-supervisor: Steffen Emil Thorkildsen, Kripos
André Årnes, Oracle

Norwegian University of Science and Technology
Department of Telematics

Problem Description

In this project, the student will study principles and investigate methods for cryptographic key discovery in memory captured from live machines, using a computer forensic perspective. The student will perform searches for keys with the intent to and identify these, based on previous research by Adi Shamir and Nicko van Someren and Torbjörn Pettersson. The primary objective is to analyze and use these methods, and eventually further develop them. Subsequently, the student will develop a proof-of-concept tool to perform key retrieval from memory dumps, using open-source cryptographic software. The master thesis will be written under supervision of the High Tech Crime Division at the National Criminal Investigation Service (NCIS), as an extension to the previous minor thesis "Digital Evidence and Cryptography".

Assignment given: 15. January 2008
Supervisor: Svein Johan Knapskog, ITEM

Abstract

Communication and whole-disk cryptosystems are on the verge of becoming mainstream tools for protection of data, both in corporate laptops and private computing equipment. While encryption is a useful tool, it also present new problems for forensic investigators, as clues to their investigation may be undecipherable. However, contrary to popular belief, these systems are not impenetrable. Forensic memory dumping and analysis can pose as ways to recover cryptographic keys that are present in memory due to bad coding practice, operation system quirks or hardware hacks. The volatile nature of physical memory does however challenge the classical principles of digital forensics as its transitory state may disappear at the flick of a switch.

In this thesis, we analyze existing and present new cryptographic key search algorithms, together with different confiscation and analysis methods for images of volatile memory. We provide a new proof of concept tool that can analyze memory images and recover cryptographic keys, and use this tool together with a virtualized testbed to simulate and examine the different states of platforms with several separate cryptosystems. Making use of this testbed, we provide experiments to point out how modern day encryption in general are vulnerable to memory disclosure attacks. We show that memory management procedures, coding practice and the overall state of the system has great impact on the amount and quality of data that can be extracted, and present simple statistics of our findings. The discoveries have significant implications for most software encryption vendors and the businesses relying on these for data security.

Using our results, we suggest best practices that can help investigators build a more comprehensive data foundation for analysis, by reconstructing virtual memory from RAM images. We also discuss how investigators may reduce the haystack by leveraging memory and process structure on Windows computers. Finally we tie this to current digital forensic procedures, and suggest an optimized way of handling live analysis based on the latest development in the field.

Preface

This Masters thesis is a product of the author's Master studies at the Norwegian University of Science and Technology (NTNU) and was given in cooperation with the Norwegian National Criminal Investigation Service (NCIS, in Norwegian: Kripos), High Tech Crime Division. The research and writing were performed over a five-month period (February-June 2008) at NCIS in Oslo, Norway.

The work may be seen upon as an extension of previous work by the author on *Digital Evidence and Cryptography*, where the usage and states of a cryptosystem were examined.

Acknowledgements

- My tutor and NCIS employee *Steffen E. Thorkildsen* for his creativity, encouraging criticism and skilled advice.
- Tutor *André Årnes* for constructive feedback and his previous work on virtualization software and digital forensics.
- *NCIS*, for generously letting me use their work space and providing me with necessary equipment, software and licenses.
- The Open Source and Digital Forensic community in general, for providing invaluable tools and inspiring source code.

Contents

Abstract **i**

Preface **iii**

Contents **v**

List of Figures **ix**

List of Tables **xi**

List of Listings **xiii**

1 Introduction **1**
1.1 Problem Definition 3
1.2 Cryptographic Key Search Scenarios 3
1.2.1 Confiscation of Computer with Encryption Software . . . 3
1.2.2 Post-capture Decryption of Communications 4
1.3 Scope 4
1.4 Intended Audience 5
1.5 Related Work 5
1.6 Document Structure and Highlights 7

I Background **9**

2 Cryptography **11**
2.1 Terminology 12
2.1.1 Main Cryptographic Goals 13
2.1.2 Good and Bad Guys 14
2.1.3 Cryptographic Attack Models and Problem Size 14
2.2 Introduction to Selected Ciphers 15
2.2.1 Rijndael (AES) 15
2.2.2 Serpent 16
2.2.3 Twofish 16
2.2.4 RSA 16
2.3 Cryptographic Keys 16
2.3.1 Symmetric Cipher Keys 17
2.3.2 Public-key Cipher Keys 18
2.3.3 Pseudo-randomness 19

2.3.4 Key Length 27
2.3.5 Key Management 29
2.4 Implementing Cryptography 38
2.4.1 Purging Keys From Memory 38
2.4.2 Compiler Optimizations 38
2.4.3 String Handling in Auxiliary Applications 39
2.4.4 Prevention of Swapping or Paging 40
2.4.5 Hardware Encryption versus Software Encryption 40

3 Windows Memory Management 41
3.1 The Memory Manager 41
3.1.1 Introduction 42
3.1.2 Memory Structure 42
3.1.3 Paging 43
3.1.4 Address Translation 46
3.2 The Physical Memory as Seen by the Digital Investigator 48

4 Digital Forensics 49
4.1 Digital Forensics Basics 49
4.2 Digital Forensics Principles 51
4.2.1 Digital Forensics and Volatile Data 52
4.2.2 Incident Response and the States of a Crime Scene 52

5 Forensic Memory Acquisition and Analysis 55
5.1 Volatile Memory Acquisition 55
5.1.1 Live Digital Forensics 56
5.1.2 Process Memory Dumping 56
5.1.3 Full Dump of Physical Memory 57
5.1.4 Comparison of Existing Acquisition Techniques 60
5.2 Existing Tools for Windows Memory Dump Analysis 60
5.2.1 The PTFinder Software Tool 62
5.2.2 The PoolFinder Software Tool 62
5.2.3 The Volatility Software Tool 62
5.2.4 The Memparser Software Tool 62
5.2.5 The KnTTools Software Tool 62
5.2.6 Harlan Carvey's Tools 63
5.3 Summary 63

II Methodology and Practical Work 65

6 Methodology 67
6.1 Cryptographic Key Search Strategies 67
6.1.1 Strategy 1: Brute-Force Dictionary Attack 67
6.1.2 Strategy 2: Compression Trial and Error 68
6.1.3 Strategy 3: Estimating Entropy 68
6.1.4 Strategy 4: Cryptographic Key Schedule Searches 71
6.1.5 Strategy 5: Structural Searches 72
6.2 Preprocessing: Rebuild Virtual Memory 79
6.3 Proof of Concept Tool: Interrogate 80

6.3.1 Choice of Programming Language 80
6.3.2 Usage 81
6.3.3 Sample Output 81
6.4 The Testbed and Environment 82
6.4.1 VMware Server 82
6.4.2 Case Generation Procedure 84
6.5 Cryptographic Software Classes 84
6.5.1 The Whole-disk Encryption Class 85
6.5.2 The Virtual Disk (Container) Encryption Class 85
6.5.3 The Session-based Encryption Class 85
6.6 Definition of Target Operating System States 85
6.6.1 The Live State 85
6.6.2 The Screensaver State 86
6.6.3 The Dismounted State 86
6.6.4 The Hibernation State 86
6.6.5 The Terminated State 86
6.6.6 The Logged out State 86
6.6.7 The Reboot State 86
6.6.8 The Boot State 86
6.7 Cryptographic Applications 87
6.7.1 Truecrypt 88
6.7.2 BitLocker 89
6.7.3 FileVault 90
6.7.4 DriveCrypt 91
6.7.5 BestCrypt 92
6.7.6 PGP 93
6.7.7 ProtectDrive 94
6.7.8 WinZip Encryption 95
6.7.9 WinRAR Encryption 96
6.7.10 Skype 97
6.7.11 Simp Lite MSN 98
6.7.12 OpenSSL and Apache 99
6.8 Expected Results 100

7 Results 101
7.1 Truecrypt Results 102
7.2 BitLocker Results 104
7.3 FileVault Results 107
7.4 DriveCrypt Results 109
7.5 BestCrypt Results 110
7.6 PGP Results 111
7.7 ProtectDrive Results 112
7.8 Results from WinZip and WinRAR Encryption 114
7.9 Skype Results 115
7.10 Simp Lite MSN Results 116
7.11 OpenSSL and Apache Results 117
7.12 Other Keys Found During Research 118

8 Discussion **119**
8.1 Evaluation of Proof of Concept Tool Interrogate 119
8.1.1 Performance Evaluation 119
8.1.2 Limitations 121
8.1.3 Further Improvements 122
8.2 General Discussion 123
8.3 Towards a Forensically Sound Approach to Cryptographic Memory Forensics 126
8.4 A Proposal for Best Practice 128
8.4.1 Key Points for Best Practice Acquisition 128
8.4.2 Key Points for Best Practice Analysis 129
8.5 Limitations and Caveats 130

III Conclusions **131**

9 Conclusions **133**
9.1 Future Work 134

Abbreviations **135**

References **137**
Publications 137
Web References 145

IV Appendices **149**

A Source Code **151**
A.1 interrogate.h 151
A.2 interrogate.c 155
A.3 stat.c 166
A.4 util.c 169
A.5 virtmem.c 174
A.6 rsa.c 177
A.7 aes.c 179
A.8 serpent.c 183
A.9 twofish.c 187
A.10 Makefile 192

B Data Structures Related to Windows Memory Analysis **193**

C Copyright Information **201**
C.1 Interrogate Source Code Licence (GPL) 201
C.2 Wikimedia Content 201
C.3 Copyrighted Content 202

List of Figures

2.1 A classical cryptosystem. 13
2.2 The key distribution problem. Figure adapted from *Handbook of Applied Cryptography*. 17
2.3 Which JPEG image contains the most information? 20
2.4 Entropy-graph for 1800 bytes of memory containing a 512-bit RSA key. The key is located at offset `0x460`. 22
2.5 Example of a (non-random) high-entropy region in memory. . . . 23
2.6 Lattice test for Unix function `rand()`. 25

3.1 Virtual and physical address space relation. 43
3.2 256 MB of RAM Memory from Windows XP (running Truecrypt) visualized by interpreting each byte as a 256-color palette color. The image can be "read" from the upper left corner, row by row. The image has 8192 rows, and is 8 pages wide (8192 x 8 x 4096 = 256 MB). The border of the pages can be seen as vertical stripes in the image. 44
3.3 Output from `pstat` on a system running Truecrypt. 46
3.4 Address translation on a x86 computer using 4 KB page size and no PAE. Figure adapted from Wikipedia (see Appendix C) . . . 47
3.5 The 32-bit virtual address on x86 Windows systems. 47
3.6 Valid x86 hardware PTE (PDE). 47

4.1 The (improved) IDIP model. 50

5.1 Comparison of Existing Memory Imaging Methods. 61

6.1 Entropy and estimate of entropy of a JPEG image (Figure 2.3(a)). Window size 256 bytes, values measured using the two algorithms Naive-Entropy-Search and Entropy-Search 69
6.2 Three visualized 128-bit AES keys with key schedule in memory. The whole key schedule is marked with blue lines. 71
6.3 Plot of entropy from the Twofish S key vectors of 256-bit keys. . 75
6.4 Plot of entropy from the Twofish K key vectors. 76
6.5 Plot of entropy from 4 KB full keying tables from Twofish. . . . 76
6.6 The Truecrypt main window with a Twofish-encrypted virtual disk mounted. 88
6.7 BitLocker in progress. 89
6.8 FileVault preferences pane. 90

6.9 The DriveCrypt Demo main window. 91
6.10 BestCrypt main window with a Serpent virtual disk mounted. . . 92
6.11 PGP Desktop Control panel. 93
6.12 The ProtectDrive pre-boot authentication screen. 94
6.13 WinZip screenshots. 95
6.14 WinRAR main window. 96
6.15 Skype main window. 97
6.16 Simp Lite MSN main window. 98
6.17 Creating a private RSA key with OpenSSL. 99

7.1 The Truecrypt driver (`truecrypt.sys`) running in the `System.exe` process. Screenshot from Sysinternals Process Explorer. 102
7.2 Enabling BitLocker for use without a TPM. 104
7.3 BitLocker successfully set up in VMware. 106
7.4 Screenshot from the process of revealing the FileVault key. 108

8.1 Percentages of found keys sorted by Software Class and State. . . 124
8.2 Taskbar Notification area icons. From left to right: DriveCrypt, Truecrypt, BestCrypt, PGP Desktop and ProtectDrive. 128

List of Tables

2.1 Tolerance intervals for runs of various lengths. 23
2.2 Reference for large numbers. 27
2.3 "AES-security"-matching RSA modulus sizes. All sizes in bits. . 29

6.1 Measured entropy values for the S-box keys of a 256-bit Twofish key schedule. $1 * 10^{12}$ samples were used, and the entropy value rounded off to four decimals. The arrow indicates that there exist many values in the interval $[3.0000, 2.0000]$. 75
6.2 Intervals of measured runs of different lengths in the Twofish key schedule. Runs of 6 or more are all counted in the '6'-bin. 78
6.3 Software classes and their expected results. 100

7.1 Truecrypt disk encryption key search results. 102
7.2 BitLocker key search results. 104
7.3 FileVault key search results. Note that hibernation mode does not exist on Apple OS X. 107
7.4 DriveCrypt key search results. 109
7.5 BestCrypt key search results. 110
7.6 PGP key search results. 111
7.7 ProtectDrive key search results. 112
7.8 WinZip and WinRAR key search results. 114
7.9 Skype key search results. 115
7.10 Simp Lite key search results. 116
7.11 OpenSSL and Apache key search results. 117

8.1 Average runtimes for Interrogate (time in minutes). The entropy algorithms were tested with their default settings (window size is 256 bytes). 120

List of Listings

3.1 Windows method ExAllocatePoolWithTag 45
6.1 Truecrypt Twofish key schedule struct 77
6.2 Twofish key schedule structures 78
A.1 interrogate.h 151
A.2 interrogate.c 155
A.3 stat.c 166
A.4 util.c 169
A.5 virtmem.c 174
A.6 rsa.c 177
A.7 aes.c 179
A.8 serpent.c 183
A.9 twofish.c 187
A.10 Makefile 192
B.1 EPROCESS data structure 193
B.2 KPROCESS data structure 195
B.3 PEB data structure 195
B.4 ETHREAD data structure 196
B.5 KTHREAD data structure 197
B.6 TEB data structure 198
B.7 POOL_HEADER data structure 199

Chapter 1

Introduction

During the last decades *cryptography* has grown to become the most important contributor to the privacy and authentication of data in an increasingly interconnected world. By using modern cryptography, an entity can achieve sufficient confidence in the privacy of its data to enable a wide range of applications that would not be possible without it. E-commerce, Virtual Private Networks (VPNs) and Digital Rights Management (DRM) all use cryptography to provide security mechanisms to the user, to mention some. Often an invisible workhorse, cryptography can bind together the idea of freedom of information that the Internet represents with applications that need security like online banking and private communication. Furthermore, it may do so in a way that allows a carefully balanced relationship between secrecy and openness, a balance that will enable third parties to verify the authenticity and security of the system or protocol.

Freedom of speech, privacy and legal rights are just some of the important values that can be protected by the use of cryptography. For example, encrypting communication can prevent a government suppressing the voice of its population, and signing data using digital signatures may juridically tie a person to a a certificate of authenticity. Using cryptography, a person may choose to be anonymous on the net. The choice is entirely his or hers, and this freedom of choice reflects the power of applied cryptography that strongly embodies principles such as net neutrality and justice. However, it is also possible to use cryptography as a device of restriction; by denying access, protecting digital rights over copyrighted material or hiding contraband and illegal material like child pornography.

The dual-edged nature of cryptography has rendered its usage, designs and applications for heavy debate and often government control [1, 2, 3, 4]. Cryptography has historically been subject to a high level of secrecy and cloaking, including heavy import and export regulations. The reason for this is quite obvious; governments wish to use the strength of cryptography while denying other governments, organizations or individuals opposing the government the same tool. Thus, in high stake situations like wars, diplomatic crisis or other matters of national security, substantial efforts are laid down in terms of funds and resources on both sides of the conflict; inventing new algorithms and breaking the existing.

For a long time, crisis and war were the driving forces behind the invention of new cryptographic algorithms and applications. The intelligence community had more or less monopoly in the field, and today it is still uncertain what magical crypto-cracking machines they may possess [5]. Despite many tinfoil-hat conspiracy theories, the idea that governments want control over data and communications is not far fetched. In fact history shows us that it is reality [2, 6, 7, 4, 8, 9] [RIP00, Gel05]. Breaking ciphers is basically a game of time and resources. Governments often has plenty of both.

As the use of encryption has increased, so has the number of crimes where digital evidence can be found [10]. It is nearly impossible to live in the western world without leaving trace in a digital format, whether it be credit card transactions, telephone records or internet usage. The need for interpretation and acquisition of these data has influenced the rise of the field of *digital forensics*, conducted by both law enforcement and private businesses. The use of cryptography poses as a problem for the digital investigator, as it may be used to hide data that may shed light on the chain of events that led up to or were a part of a crime.

Modern cryptographic best-practices, acquired from countless hard-learned lessons, suggest that open standards that enable peer review and public scrutiny is the preferred practice of gaining confidence in a cryptographic method. Thus it is not the secrecy of the design, but that of a *key* that provides the security of the system, according to Kerckhoffs' principle [11]. No cipher can be said to be 100 percent secure, but a cipher that has resisted ten years with public evaluation and testing is certainly preferable to a new cipher with a higher on-paper grade of security.

Using this principle and joining forces, the academic and intelligence community, corporations and standardization organizations have come up with some remarkably strong ciphers, notably the Advanced Encryption Standard (AES) [12, 13] and public-key schemes like RSA [14]. AES and RSA with proper key lengths seem unbreakable on paper in the foreseeable future, but what about in practice?

When a smart man sees an obstacle, he goes around it. Famous cryptologist Schneier has pointed out that although software-based encryption is common and easy to implement, it does not offer any physical protection of the algorithm or the key [7]. Any person that has physical access to the system can analyze it with debugging or reverse engineering tools, modify the algorithm or look for the key. Such creative attacks has a reputation of defeating cryptosystems that are secure on paper. While hardware systems may offer protection by tamper-resistant devices or other physical defenses, software is dependent on mercy and good behavior[1] from the computer and operating system it runs on.

Recently the attention of the security community has been focused on physical access attacks that can defeat encryption mechanisms [Sch08, Fel08, Zet08]. It is most common to think of these attack vectors as a way for crackers to get in, but nearly all of them require physical access, which involves committing a crime (e.g., theft of laptop) or at least a large risk for the average attacker.

[1] Small malicious programs called trojans are often used by attackers to modify Operating System (OS) code so that it may reside undetected by virus defense systems. It is therefore extremely hard to guarantee only good behavior from an operating system. In addition, their complex structure and closed-source nature makes it difficult to even trust them out of the box.

A group of professionals that often *do* have the privilege of total physical access and virtually zero risk, are digital forensic investigators. Since the nature of cryptography makes it attractive for hiding incriminating data, the encrypted material often contain exactly this evidence that investigators seek. In this thesis we consider approaches for the investigators to defeat cryptosystems by means of finding the key or parts of it in volatile memory and swap space.

1.1 Problem Definition

We seek to discover new methods for cryptographic key location, improvements of the existing methods, and perhaps most importantly answer the following question: *How does the state of the system effect the chances of uncovering keys in memory, and how can the chances of such a discovery be maximized?*

By focusing on these problems, the author strive to *unify* memory analysis, cryptography and digital forensics in a way that will allow a higher success rate for law enforcement when encountering cryptographic applications on live digital crime scenes.

Although aim the thesis is aimed at law enforcement agencies, it clearly highlights some of the problems the security community faces today in terms of protection of data using software encryption. Many of the approaches mentioned in our research can be exploited by criminals or people with malicious intent; and these risks are hard to mitigate with todays standard practices.

1.2 Cryptographic Key Search Scenarios

Cryptographic applications are in general required to keep its keys in some sort of form in physical memory when operating. To aid our treatment of cryptographic key searches, we present three different example scenarios where such searches may be feasible, and of help to forensic investigators. The list below is not exhaustive, there may be possible to identify several other scenarios.

1.2.1 Confiscation of Computer with Encryption Software

Whole-disk encryption systems are gaining popularity and are, especially in the business-sector of the market, used to protect valuable and sensitive data. This type of software exist on both laptops and desktops, and in the future, mobile devices. Several operating systems come with such cryptosystems integrated, like Microsoft Vista's BitLocker (Ultimate and Business edition) [15] [Mic08c] and Mac OS X's FileVault [App08]. Using these systems, the user may attempt to conceal all data on his or hers hard drive, effectively thwarting regular forensic investigations of the hard drive.

Additionally, encryption software that feature container or virtual disk encryption can be used to protect a subset of the data on a computer. This is an encryption method that the user can relate to (it is analogous to that of a locked container), and it is therefore quite widespread [16, 17]. OS X comes with such software out of the box, and applications like Truecrypt [Fou08a] and BestCrypt [Jet08] feature this type of encryption. Other applications like WinZip feature both compression and encryption, ensuring the security of files during transfer or storage.

Memory analysis and key extraction can be useful to an investigator when encountering *powered on* computers with encryption software, or if remains of the physical memory can be retrieved from the hard drive. This may be the possible as a result of hibernation or other OS-related processes like paging.

1.2.2 Post-capture Decryption of Communications

Several messaging applications including mail, texting, chat- and voice-chat applications come with encryption options either as standard features or plug-in modules. These applications or modules encrypt communication with similar clients from end-to-end, using strong algorithms. Consequently, law enforcement agencies cannot read the messages passed back and forth, and possibly miss vital data in the context of lawful interception.

Dumping the encrypted data can however yield success if the decryption key is made available at a later point in time, depending on the type of cryptosystem. This may leverage methods where the investigators is able to perform surveillance on suspects over an extended period of time, and decrypt the material after the encryption key is found (either by questioning, cracking or forensic memory and hard drive searches). If the computing equipment that was utilized in the communication is confiscated while it is live or just powered on, key extraction may be possible. It has also been shown that decryption of whole SSL/TLS sessions are possible even when only network dumps and key material is made available [18].

1.3 Scope

This thesis is formed in the mindset of digital forensics. There exists several attempts [19, 20] to formulate best-effort practices and frameworks for memory forensics, but due to the young age of the research and the multitude of different architectures and software available, there is inherent redundancy and lack of standards.

The scope of this thesis limits itself to forensically sound discovery of encryption keys. Consequently, we stress to keep all methods and research within forensically sound practices. The techniques for memory dumping will be addressed, and their feasibility for a forensic investigator analyzed. Searches on the hard drive for any remanence of cryptographic key data, plaintext or other clues will not be considered; even though this is a procedure that an investigator certainly would have conducted.

One other major limitation is time. This thesis was created within a five-month period, and the depth of treatment of several of the subjects are limited because of these time constraints. We also limit our research to Windows operating systems, owing to the fact that Windows is by far the most used OS in the private segment of the market today, and consequently what an investigator encounters most of. In addition to its dominating market share, the aforementioned time restrictions forced us to abandon other platforms. That being said, many of the methods suggested and implemented in this thesis is applicable with none or minor modifications on any device or platform using (volatile) memory.

The basics of cryptographic aspects like key properties, generation and usage will be covered, all which can have great impact on the possibilities of finding cryptographic keys in memory. We will not discuss the encryption procedures at any length, unless it is necessary to clarify certain characteristics of the search methods. For further reading on the art of cryptology, please be referred to Kahn's excellent *The Code Breakers* [2], or for a more technical approach, Bruce Schneier's *Applied Cryptography* [7] is the de-facto standard.

1.4 Intended Audience

The primary audience of this thesis are digital forensics professionals, law enforcement and the IT security community in general. We assume fairly high technical skills, but no theoretical knowledge of cryptography or memory management is needed, as the necessary background will be covered. It is however assumed that the reader knows programming and has basic computer knowledge.

1.5 Related Work

Dumping and examining volatile memory for forensics purposes is a relatively immature procedure, even though the concept has been known for a long time [21]. The memory acquisition process is especially irregular and unstandardized, mostly because of the number of different operation systems and platforms, and there exists a myriad of different approaches. A good comparison of the available methods for Microsoft Windows can be found in the paper *Windows Memory Forensics* by Ruff [22]. The methods for extracting volatile memory ranges from DMA access via FireWire by Dornseif [23] and Martin [24] to simply copying of memory from `/dev/mem` on Unix-flavor platforms.

An approach on cryptographic key search and identification are proposed by Shamir and van Someren, suggesting the prospect of "lunch-time" attacks against mainframes in their article *Playing Hide and Seek with Stored Keys* [25]. In their paper they propose to use simple statistical and visual methods to locate memory regions that (likely) contains encryption keys. In another article, Pettersson discusses searches for structural properties of the code that are holding the key, by analyzing and "guesstimating" the values of surrounding variables [26]. Ptacek [Pta08] drafts how to extract and verify RSA keys from memory, using a simple mathematical analysis of the parameters found. On identifying RSA keys, Klein suggests searching for ASN standard prefixes of the DER-encoding, both identifying certificates and private keys in memory [27].

In a related paper, Harrison and Xu presents experiments [28] showing that Apache web- and OpenSSH servers can be subverted into disclosing its private RSA key by exploiting an information leak in the linux kernel [LF05]. They also discuss methods for mitigating the risk, and show that RSA keys are disclosed statistically within one to five minutes after attack start.

Also related, the authors of Volatools (Walters and Petroni) describes a hypothetical attack against TrueCrypt [Fou08b], by studying its (and the underlying OS's) internal structures and behavior [29]. The attack is used as an argument to incorporate memory forensics in regular digital forensics, but they

do not describe how to locate the different structures in memory, and neither discuss the fact that some of these may be paged out, thereby breaking the chain of data structures that leads to the master key.

A recent breakthrough was released by Halderman et al. during the writing of this thesis, in their paper *Lest We Remember: Cold Boot Attacks on Encryption Keys* [30]. In the article, they demonstrate that it is possible to leverage remanence effects[2] in DRAM modules to "coldboot" the target computer, load a custom OS extracting the memory to an external drive, locate the key material and decrypt hard drives automatically. Because of the risk of bit errors in a decaying memory image, they even suggest methods for correcting such errors by utilizing the inherited "dead state" structure of the DRAM modules and an error-correcting code. A complete and automated cracking procedure is demonstrated several places at the Internet [McC08, HSH$^+$08], but at the time of writing no source code has been released. The authors do however promise to do so in the future [McC08]. Nonetheless, they seem to focus on *malicious attacks* on the systems using whole-disk crypto like Bitlocker, FileVault and TrueCrypt, and not a forensic investigation.

Most of these methods treat the memory as a large blob of bytes, although in fact memory is highly structured. Some of the methods suggest skipping duplicate regions and reserved address space, but do not consider to reduce the haystack by just looking at the probable regions of the memory. Such reduction may be performed by dumping the memory space of processes that are involved in encryption, and analyzing the output. Process dumping and analysis has been done in other fields of memory analysis, where analysts have dumped the memory address space of a specific process by fetching pages from RAM and swap space. Using the dump they are able to verify[3] and sometimes be able to totally reconstruct an executable file [31], even from dead processes. According to several articles (see Schuster [32] and Carvey [33]), these techniques are able to identify trojans, rootkits and viruses that are stealthy and/or armored in Windows memory dumps.

Research has also been performed to indicate the age of freed user process data in physical memory. Solomon et al. have shown that large segments of pages are unlikely to survive more than five minutes, even on a lightly loaded system. However, they are able to find smaller segments and single pages up to two hours after initial commit [34].

There also exist several publications on the digital forensics field from the author's home institution the Norwegian University of Science and Technology, notably Bent Kristoffer Onshus' minor thesis *Cryptographic Credentials and Encrypted Data in Digital Evidence* [35] and Andreas G. Furuseths *Digital Forensics: Methods and tools for retrieval and analysis of security credentials and hidden data* [36]. These do, however, tend to focus on the hard drive rather than the memory as a target of investigation.

Despite all this contemporary research, there exist little information on *what* and *how much* data that can be found in memory dumps. In the minor thesis *Digital Evidence and Cryptography* [18], we attempted to shed light on how the different *states* of the system impacts the data that can be found. We used stan-

[2]Remanence effects is the effect that all Dynamic Random Access Memory (DRAM) modules keep their state for a time (typically a few seconds) before it needs to be refreshed by the memory controller.

[3]By using tools like SSDeep by J. Kornblum [Kor07].

dard hard drive forensic tools together with memory dumps to identify plaintext copies of encrypted content, effectively subverting crypto. This thesis will focus on how to use all of the above memory forensics methods in combination together with cryptographic knowledge to extract key material from volatile memory, and perform controlled experiments that will indicate the probability of such an extraction to be successful.

1.6 Document Structure and Highlights

Throughout this thesis, code will mainly be represented as C or C++ unless otherwise noted, while pseudocode is printed using the syntax from *Introduction to Algorithms* (famously known as CLRS, abbreviated from its authors) [37]. The exception is the description of the cryptographic ciphers themselves, that rather will be given in mathematical notation.

The rest of the thesis is divided into four main parts that are organized as follows:

- The first part of the thesis treats the theoretic background necessary to discuss cryptographic key searches and memory acquisition and analysis. Specifically, we treat the theoretic background of cryptographic keys, digital forensics and Windows memory internals.

- In the second part, the theory is merged with the practical part performed as a part of this thesis. A complete methodology is presented, together with algorithm descriptions and the virtualized testbed utilized to perform our scientific experiments. The applications that are tested are introduced, and we provide well-defined *cryptographic software classes* and *operating system states* to facilitate case generalization and a more clear discussion of the subject. Moreover, we present the results from these experiments and provide a broad discussion of their significance and implications. Finally, we outline an approach towards a more forensically sound practice for volatile memory acquisition and analysis.

- The third part concludes the thesis, and provides a summary of our findings and conclusions. Suggestions for future work are also proposed.

- The appendices contains the source code of the proof of concept tool *Interrogate* developed as a part of this thesis, Windows memory-related data structures provided as a convenience for the reader and copyright information.

Part I

Background

Chapter 2

Cryptography

Cryptography, derived from the Greek *kryptó* "hidden" and *gráfo* "to write", is the ancient science and art of hiding the content of a message from prying eyes. Although now considered a branch of modern number theory and computer science, it was originally literarily done by hand as early as 4000 years ago [2]. The Egyptians employed *substitution ciphers* to substitute hieroglyphs with less common varieties of hieroglyphs in inscriptions on grave chambers, presumably in order to obfuscate the meaning of the inscriptions for people who did not know how to reverse the substitutions.

Number theory and its applications is also an ancient art, dating back to the Pythagoreans [38]. Two of the oldest number-theoretic algorithms, *the Euclidean algorithm* and *the sieve of Eratosthenes* (both conceived around 300 BC) are still used and are closely related to most modern cryptosystems.

Ever since computers first were employed in the field of cryptography, complexity of newborn cryptographic algorithms were permitted to rise. A computing device is ideal for encryption and decryption, it is deterministic and can perform sequential and parallel tasks that would have been impossible for a human at the same speed. These advances has not only improved cryptography, but also the art of breaking codes, namely *cryptanalysis*.

It is a well-known fact that there is a ceaseless battle between creating and breaking cryptographic algorithms and protocols. Algorithms used today may be broken in the future, as a consequence of technology advances and increased computing power. One thing that has not changed since the Egyptians carved their legacy in stone, is the fundamental property of *the key*, that is, one or more objects that must be kept secret to maintain the security of the cryptographic system. Most theoretic attacks on cryptography therefore often concentrate on finding this key in addition to flaws in algorithms design, but also practical attacks can benefit from focusing on the key.

We will use this chapter to discuss cryptography in a somewhat superficial way; there is no need to fully understand each algorithm or intricate mathematical evidence to realize that if the key of a cipher is compromised, all is lost in terms of security. To be able to locate cryptographic keys in volatile memory, we need to develop an understanding their properties and usages. Therefore the focus will lie on cryptographic topics that relate to the generation, management, usage and storage of cryptographic keys.

First, we will discuss the basics of cryptography, before introducing the ci-

phers analyzed in this thesis. Secondly we will treat cryptographic keys, with emphasis on the key management procedures of the selected ciphers. An introduction to randomness and its applications in cryptography is also given, before we discuss implementation-specific issues related to key management and volatile memory analysis.

2.1 Terminology

Cryptography uses its own terminology, that we will attempt to follow throughout this thesis. Generally, the cryptographic terminology in this thesis is consistent with Schneier's *Applied Cryptography*.

First of all, *crypto* will often be used as shorthand for cryptography, or even cryptology. *Encryption* is the process of encoding a message to hide its content, and *decryption* is the inverse operation. The mathematical notation for these two operations are $E()$ and $D()$, respectively. A visualization of these processes can be seen in Figure 2.1.

Cipher will denote the cryptographic algorithms discussed, and *plaintext* and *ciphertext* the corresponding pair of plain and enciphered messages. In diagrams and formulas these two objects are often denoted m for plaintext and c for ciphertext. Using this and the mathematical notations for encryption, we can see that:

$$E(m) = c \tag{2.1}$$

The decryption process can thus be noted as:

$$D(c) = m \tag{2.2}$$

And by swapping $E(m)$ for c we easily see that decryption of encrypted content work as follows:

$$D(E(m)) = m \tag{2.3}$$

The ciphers are dependent on one ore more *keys* for their secrecy, as mentioned in Chapter 1. We indicate encryption and decryption of plaintext m using a key K as:

$$E_K(m) = c \tag{2.4}$$
$$D_K(c) = m \tag{2.5}$$

Keys are also in some cases denoted e and d (mostly in public-key contexts), for encryption key and decryption key respectively. Some algorithms use different keys for encryption and decryption. Such keys are enumerated using subscripts, e.g., K_1, K_2.

A system consisting of a cipher, all possible plaintext/ciphertext pairs and corresponding keys is called a *cryptosystem*. The classical cryptosystem is pictured in Figure 2.1. Please note that the keys may or may not be the same, as described above.

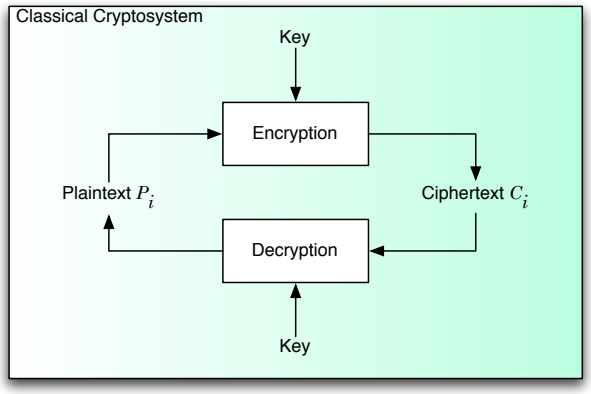


Figure 2.1: A classical cryptosystem.

2.1.1 Main Cryptographic Goals

Menezes et al. [1] defines four goals or services cryptography attempts to provide:

1. Confidentiality *is a service used to keep the content of information from all but those authorized to have it.* Secrecy *is a term synonymous with confidentiality and privacy. There are numerous approaches to providing confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.*
2. Data integrity *is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such things as insertion, deletion, and substitution.*
3. Authentication *is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. [...]*
4. Non-repudiation *is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary.[...] A trusted third party is needed to resolve the dispute.*

It is possible to identify many other applications of crypto, but these functions are usually the basis of any such application. For example, Digital Rights Management (DRM) may be employed through use of the confidentiality, integrity and authentication mechanisms. As may be seen, the borders between these services are not set in stone, and some overlap do occur.

2.1.2 Good and Bad Guys

When discussing crypto, we usually set up a scenario where an malicious attacker attempts to intercept and interpret a confidential message transmitted over some insecure or open medium. The two entities attempting to communicate are often called *Alice* and *Bob*, and we will stick to this convention throughout this thesis. We will also use *Mallory* as our name for the adversary, although the adversary in our text might be an investigator with presumably "good" or lawful intentions. In fact, we assume that the adversary is such an investigator; by using Mallory as denomination for all attackers, we illustrate the fact that the methods discussed can be utilized by any person with sufficient technical skills.

While the classic scenario of sending confidential messages over an insecure link (e.g., the Internet) is adequate for most purposes when discussing cryptosystems, we also need to consider a scenario where Bob or Alice encrypts his or her hard-drive. The necessity of such a protection can be linked to many different scenarios, for example enterprise laptop theft or simply privacy considerations. In this seemingly straight-forward scenario, the cryptosystem needs to prevent any access to the protected data, even if the adversary gets physical access. Actually, this scenario is a version of our classical scenario, where the insecure medium is the same as the platform where the encryption/decryption takes place. As we shall see, this has significant influence on the security of the cryptosystem.

2.1.3 Cryptographic Attack Models and Problem Size

Considering attacks on cryptographic applications or ciphertexts, we generally divide the attack types into four distinct *attack models* [6]:

- *Ciphertext only* attacks are mounted by trying to recover the key or plaintext from the ciphertext. Only the ciphertext is available to the adversary.
- A *known plaintext* attack is performed if the cryptanalyst has access to the ciphertext and some of the plaintext.
- In a *chosen plaintext* attack the analyst may choose the plaintext, and obtains the ciphertext by encrypting it.
- *Chosen ciphertext* is the opposite, here the analyst may choose the ciphertext, and obtain the plaintext by decryption. The goal for these last two attacks are to uncover the key, and may be difficult to mount in real-life.

These attack models are relevant to all cryptography, but the attack model in this thesis is somewhat different: We attempt to locate a key in an arbitrary amount of data. To test that the found key indeed is the key we are looking for, the above attack models may be used. For example, one approach for identifying a key could be to attempt to apply all subsets of the volatile memory as a key, and decrypting a chosen ciphertext that has a known plaintext. If the ciphertext decrypts to the correct plaintext, the key is found.

We do however need to pay attention to the computational effort needed to run such a brute-force attempt. To describe computational complexity, we define the following terms:

Time Complexity

The *time complexity* denotes the expected time to solve a problem, in our case this often means expected time for cracking the cipher. Given a cipher where brute-force key search is the best option, this value is directly dependent on the key size. Note that this terminology does not express the time complexity in time measurement units like seconds or years, but rather in *problem size*. The time complexity for guessing a 56-bit key is therefore around 2^{55}, how long time it will really take to guess it depends on your resources and luck.

Space Complexity

Just like time complexity, *space complexity* denotes the problem size in terms of space requirements. There exists methods for cipher-cracking that requires huge amounts of data, for example Rainbow Tables [39] and differential cryptanalysis [40, 41]. Space complexity is, like time complexity, expressed in problem size.

Also please note that even if some cryptographic vendors advertise with "unbreakable" and "military grade" ciphers, no cipher is unbreakable. We use the term *computationally infeasible* to denote all tasks that are so computationally heavy that they are impossible to perform with available resources, either present or future [7]. Using this, we can see that a cipher that has a key size of 256 bits and no better way of breaking it than guessing the correct key has a predicted solving complexity of $2^{256-1} = 2^{255}$. Given that a MIPS year[1] is around 31.5 trillion instructions per year, a typical Intel Core 2 @ 3.2 GHz computer would (theoretically) use 529.812.463 years to break the key. An array of a million distributed processors with the same specifications would still use 530 years. Consequently, we would consider this algorithm *computationally secure*. Some people would reason that this means the cipher is impossible to break. But impossible is a word that should be carefully weighed when used together with cryptography. To put these huge numbers in perspective see Section 2.3.4, that treats key lengths.

2.2 Introduction to Selected Ciphers

In this thesis, we will focus our attention towards some selected ciphers, whose keys are to be searched for in memory. We've selected the three block ciphers with highest vote-counts from the AES selection process [42], namely Rijndael (now AES), Serpent and Twofish, and one of the most popular public-key cipher RSA. We will briefly introduce each of these algorithms here as their key properties will be treated more in-depth later in the thesis (see Section 2.3.5).

2.2.1 Rijndael (AES)

The Rijndael cipher was selected as the Advanced Encryption Standard in 2001 [12], formed from a proposal by Joan Daemen and Vincent Rijmen [43]. It is a Substitution-Permutation (SP)-network based cipher that works on 128-bit

[1] MIPS (Million Instructions Per Second) is a measuring unit equalling one million processing steps per second [Wik08]. As a result, a MIPS year is $1000000 * 365 days/year * 86400 seconds/day$, or approximately 31.5 trillion instructions.

blocks, and can use either 128, 198 or 256 bit keys. AES is widely in use, fast in both software and hardware and is regarded as the de-facto standard in most new cryptographic applications. AES encryption is present in a vast range of applications, among others Truecrypt, Vista BitLocker, OS X FileVault, DriveCrypt, BestCrypt, PGP, ProtectDrive, WinZip, WinRAR, Skype, Simp Lite and OpenSSL.

2.2.2 Serpent

Serpent came second in the AES selection process [And00], after a submission from Ross Anderson, Eli Biham and Lars Knudsen [44]. It is a 128-bit block cipher based on a SP-network. To provide reliable and scrutinized security properties, it reuses the S-boxes from DES, perhaps the world's most analyzed cipher. While primarily intended for use with 256-bit keys, all keys are padded up to 256 bits if needed, and the cipher therefore accept shorter keys. Examples of applications that feature Serpent encryption are among others Truecrypt and BestCrypt.

2.2.3 Twofish

Twofish ended third at the last AES conference, and it is a 128-bit cipher that accepts variable-length keys up to 256 bits [Sch98]. The cipher is based on a 16-round feistel structure with a bijective encryption function F made up by key-dependent S-boxes, matrix multiplication over a *Galois Field* ($GF(2^8)$) and several other transformations described in Section 2.3.5. It was submitted by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson [45]. Applications that feature Twofish encryption are among others Truecrypt, BestCrypt and PGP.

2.2.4 RSA

RSA (abbreviated from its authors, Rivest, Shamir and Adleman) is an algorithm for public-key cryptography first described in 1977. It can operate on variable plaintext lengths, and use keys of variable length, usually powers of two (1024, 2048, etc.). It is in wide use in communications protocols and key exchanges, and also in areas like mail encryption. Being a public-key algorithm, it is far slower than the block ciphers described above. RSA is utilized in many applications, among them PGP, Simp Lite, Skype and OpenSSL.

2.3 Cryptographic Keys

This paper attempts to shed light on the possibilities of finding cryptographic keys in volatile memory. As a consequence, a basic theoretic treatment of such keys and their properties is warranted. Cryptographic keys have many usages, storage options, protocols and best practices associated with them, some of which we will attempt to summarize in this section. Traditionally, ciphers are divided into two main categories based on the key types, namely *symmetric ciphers* and *public-key ciphers*.

2.3.1 Symmetric Cipher Keys

Symmetric ciphers are based on a single key that usually are used both for encryption and decryption. All parties that has access to this secret key are able to decrypt ciphertext encrypted under the key. Some of the most commonly used algorithms today are the Data Encryption Standard (DES and 3DES) [46], Advanced Encryption Standard (AES) [13, 12], Twofish [47, 45], Serpent [44, 48], CAST [49, 50], and IDEA [51] (which is patent protected).

Symmetric keys must be kept secret from unauthorized entities, and this can often lead to the famous *key distribution problem* (Figure 2.2); if no contact has been made beforehand by two communicating parties, how can they agree on a common key? If the key is to be transmitted from Bob to Alice, that would require some sort of mechanism to provide confidentiality and integrity of the key, but that is exactly what we are trying to archive in the first place by using cryptography. Thus we are facing the same problem (establishing a shared secret) over again.

Figure 2.2 illustrates this problem. Alice and Bob are attempting to communicate securely, facing and adversary that can eavesdrop on their messages. To establish the shared secret e, a secure channel is needed.

As we can see, a *key distribution protocol* that solves this problem is needed to effectively use a symmetric algorithm in a communications scenario. Several such protocols exist; the Diffie-Hellman (DH) key agreement method [52] is commonly used, but another elegant solution to the problem is available through public-key cryptography.

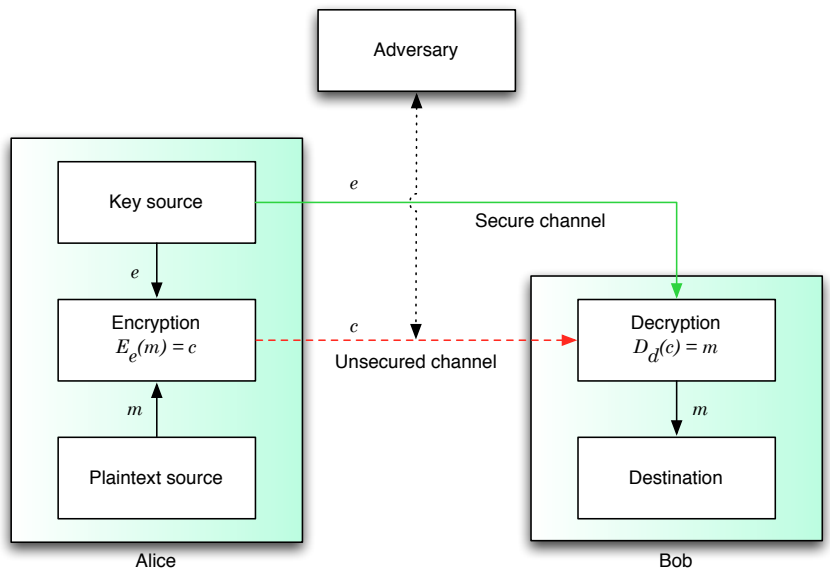


Figure 2.2: The key distribution problem. Figure adapted from *Handbook of Applied Cryptography*.

2.3.2 Public-key Cipher Keys

A *public-key cipher* is a cryptographic algorithm that uses a mathematically linked *pair* of keys, one *public key* (here denoted by K_{pub}) that can be distributed freely, and a private (K_{priv}) key that must be kept secret from anyone else than the owner. Using our formerly established naming conventions, the public key is used by the sending entity to encipher messages, and decrypted by the receiver using the its private key:

1. Alice: $E_{K_{pub}}(m) = c$
2. Bob: $D_{K_{priv}}(c) = m$

Public and private keys are usually a collection of mathematical primitives (depending on the cipher type) used in the encipher/decipher calculations. For example, in RSA the tuple (e, n) is the public key, where e is a number relative prime to the product $\phi = (p-1)(q-1)$ and n is the modulus ($n = pq$) of the calculations. The private key is a number d relative prime to n such that:

$$ed \equiv 1 \pmod{\phi} \tag{2.6}$$

As opposed to symmetric keys, public keys are not just random bits, but (potentially large) numbers with distinct and provable mathematical properties (like primality). The security of the cipher relies on these properties, so proper selection and testing of the qualities of the numbers are of utmost importance (see Section 2.3.3).

Since public-key cryptography was made famous by the invention of RSA [14, 53], several other algorithms have been suggested [1, 4], among others the Digital Signature Algorithm/Standard (DSA or DSS) [54], Diffie-Hellman [52], ElGamal encryption [55], NTRU [56], and Elliptic Curve (EC) versions of these [1, 57, 58]. Although in widespread use, symmetric key algorithms are favored for bulk encryption because of performance reasons; public-key ciphers are significantly slower than their symmetric brethren, and their key sizes must be much larger to provide the equivalent security (see Section 2.3.4).

Returning to the key distribution problem, we can see that public-key crypto solves it by simply encrypting the shared secret (K) with the public key of the receiver:

$$E_{K_{pub}}(K) = c \tag{2.7}$$

The receiver can retrieve the key by decrypting c:

$$D_{K_{priv}}(c) = K \tag{2.8}$$

The receiver can now decrypt the key using his or her private key. Public-key cryptography solves the key distribution problem, but introduces another one; namely how ascertain that a given public key actually belongs to the entity claiming it. The problem has thus been transformed to an authentication issue.

2.3.3 Pseudo-randomness

One of the building blocks for symmetric and public-key cryptosystem key generation are *Random-Number Generators* (RNGs). These are used to generate random keys or nonces with desired properties, like for example large prime numbers. There exists a number of academic papers on the subject [59, 60, 61, 62], even whole books filled with random numbers taken from decaying radioactive material. Generating true random sequences is however not as easy as it may seem.

True randomness is hard to define accurately. It is not possible to say that a sequence of bits is not random; the output '`0101010101010101`' may very well be the output of a truly random process, even if it does not look random to a human observer. As a result, randomness is a highly objective property. The one thing separating truly random from pseudo-random is that the sequence *cannot be reliably reproduced* [7]. Sources that are believed to be random are the decaying of radioactive material, movement of particles suspended in liquid or gas (Brownian motion) or simply the sampling of movement of the international stock market[2]. All these are *stochastic processes* lacking order and predictability, and therefore they may be interpreted as truly random.

The problem is that it is hard to produce truly random bit sequences on a computer; it is per definition a deterministic machine. If you input data, and get some data out in return, you know that if you input the same data at the same state, you will get the same output.

Consequently, it is not feasible to generate real random numbers using computers[3]. Instead, *pseudo-random* sequences can be generated efficiently at a computing device. These are numbers that appear random, but are deterministically computed from a given state or *seed*. In this thesis, we use Schneier's definitions [7] of pseudo-random sequences; that it *must look random*. That is, it passes chosen statistical tests, some of which are covered in Section 2.3.3. We call this the *pseudo-random property*.

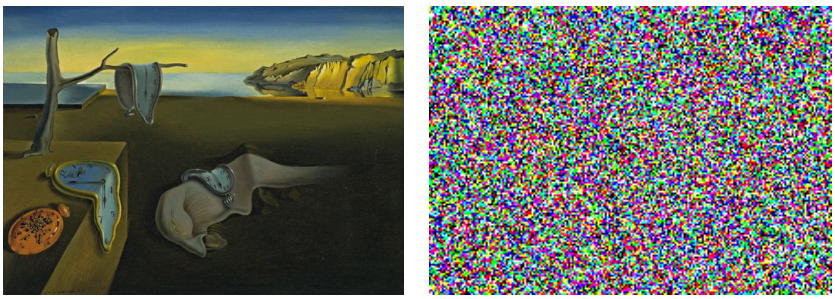
Unfortunately pseudo-random number generators are per definition not truly random at all. Like most RNGs supplied in compilers and programming languages they are highly predictable, and a skilled observer could predict the next output by studying past output. As a consequence, all RNGs are not suitable for cryptographic applications. In addition to the pseudo-random property we want cryptographically secure RNGs to be unpredictable, so that it is impossible to predict the next bit in the sequence based on the previous bits. We call this the *cryptographically secure pseudo-random property*.

To verify this property of a RNG, rigorous testing is performed with the generator to build confidence that the output it is undistinguishable from a truly random output. We will cover some of these tests in the following sections.

[2]Whether or not the stock market is a stochastic process or not is a debatable issue, a stock broker would probably oppose this idea.

[3]Without a truly random source connected to the computer.

Entropy



(a) *The Persistence of Memory* by Salvador Dalí (b) Random Noise

Figure 2.3: Which JPEG image contains the most information?

One of the most widely used measures for *information content* and randomness is entropy. The information content tells us how much information one *symbol* gives us, when we view the information stream as a continuos stream of *stochastic nature*; that is, that the next symbol to be read are unknown to us, and that the information we receive may look arbitrary and chaotic.

Streams of bits and bytes in digital media are such stochastic streams, their readability depending on the granularity in which we look upon them. A stream of bits may look totally random and without patterns, but by grouping these symbols into higher level and predefined symbols like bytes or words, patterns may emerge. The interpretation of the patterns thus depends on the symbols used at the machine reading the stream, and a stream may be interpreted differentially at different machines. This would of course not yield any sensible information transfer, since without a properly defined alphabet it is hard (but not necessarily impossible) to decipher what the stream really should be interpreted as.

The entropy of a message M with an alphabet size of ω is defined by Shannon [63] as:

$$H(M) = E\{I\} = \sum_{i=1}^{\omega} p_i I_i = \sum_{i=1}^{\omega} p_i log\left(\frac{1}{p_i}\right), 0 \leq p_i \leq 1 \tag{2.9}$$

Here, $E\{.\}$ is the statistical expectation operator and I is the information content, while parameter p_i is the probability of encountering that symbol i. We easily see that in an uniform distribution, all these probabilities will assume the same value, namely $1/\omega$.

When we are working with information transfer (which, essentially all digital media and computers are all about) we have to treat the signals like stochastic information. This is indeed the core of information theory; true information transfer happens when the receiver does not know what the next piece of information will be before he has received it. The logicality of this statement should be quite clear; there's no sense in transferring information that the receiver already know. Thus entropy also tells us something about the *uncertainty* of a message or stream, that is, how many bits that are needed to be recovered to discover the meaning of the message. In cryptographic terms, the uncertainty

is how many bits of the plaintext that need to be discovered to infer the whole message. If a message can be represented by a single bit, like a typical boolean relationship `"true"`/`"false"`, a cryptanalyst needs only do discover one carefully selected bit to recover the whole plaintext. If the ciphertext `"lal"` is either `"true"` or `"false"`, one one bit plaintext could reveal the whole plaintext since the entropy of the message is 1.

Random sequences of symbols has entropy approaching the maximum value for the alphabet, and thus mimics the properties of a uniform distribution. This is quite logical, since we want each symbol in the alphabet to appear with the same probability as the others, so that no one can predict the next symbol accurately. A random sequence of bytes M_{bytes} will approach a entropy value of 8 bits per byte when a large enough sample size is used, since the alphabet size $\omega = 2^8 = 256$ and each p_i in $W = \{p_1, p_2, ..., p_\omega\}$ equals 1/256. We may express this as (using 2.9):

$$\begin{aligned} H(M_{bytes}) &= \sum_{i=1}^{256} p_i \log_2 \left(\frac{1}{p_i}\right) \\ &= \frac{1}{256} \sum_{i=1}^{256} \log_2 (256) \\ &= \log_2(256) \\ &= 8 \; bits/byte \end{aligned}$$

Since we are measuring the information content (entropy) of bytes and using the base 2 logarithm, the information content in each symbol (e.g., byte) is measured in *bits*, and expressed by *bits per symbol* or *bits per byte*. The choice of base for the logarithm is essentially free, but base 2 is commonly used for digital information content.

It also follows from the above that random data cannot be significantly compressed, since it is already approaching its maximum entropy value $\sqrt{\omega}$, depending on the sample size n as explained above. We can express this as

$$\lim_{n \to \infty} H(M_{bytes}) = \sqrt{\omega} \tag{2.10}$$

Therefore, given a large enough sample size, random data will approach its maximum entropy value. That is, it is not possible to express the information any more efficiently using bits, and the random data representation is therefore a *minimal representation*. Consequently, if you could express the information more efficiently, it would be an indication of non-random data.

As inferred from the above there exists many other types of data that shares the property of high entropy with encrypted data. Compressed files have high entropy; JPEG images has typically an entropy value of 7.9-8 bits per byte. This does not mean that compressed data is random, it is usually highly correlated, but still has high entropy values.

In Figure 2.4, a 1800-byte segment of the physical memory of a Ubuntu Server 7.10 using OpenSSL is visualized by sliding a 256-byte window over it and calculating the entropy of each window. A 512-bit RSA key is located at offset `0x460`, and as we can see, it has a distinctly higher entropy value (around 7.0) than its surroundings. Unfortunately, this is not always the case.

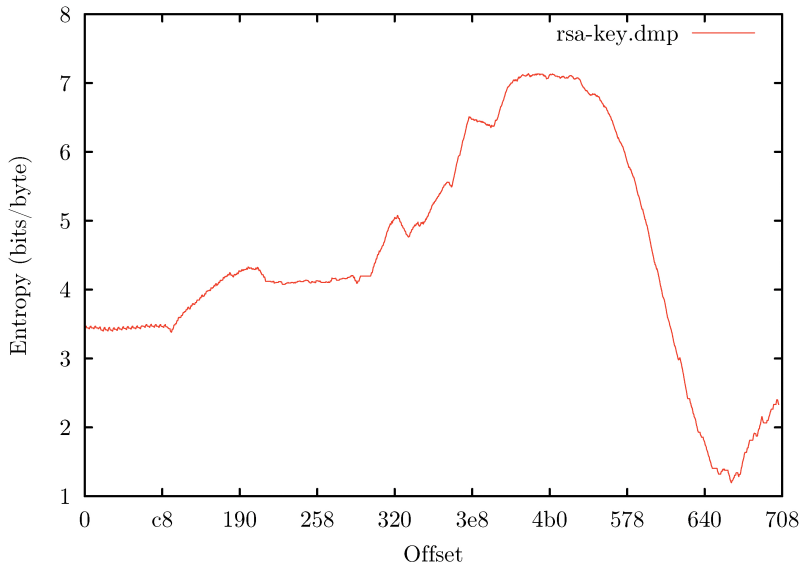


Figure 2.4: Entropy-graph for 1800 bytes of memory containing a 512-bit RSA key. The key is located at offset `0x460`.

The surrounding data may very well have high entropy values, as shown in Figure 2.5.

Figure 2.5 shows a typical non-key high entropy region taken from an image of the physical memory of a Windows system. The region is clearly not random, as it is simply a sequential string of bytes. For a pure entropy search, this region would probably be counted as a search hit. If we are to reduce the number of false positives when searching for high-entropy regions like suggested by Shannon, we need to distinguish between compressed, non-random data like the above and (pseudo-)random data.

Other Statistical Methods for Evaluating Randomness

Fortunately, there exists a myriad of statistical methods for evaluating the randomness of data [61]. Many of these methods are needed by the cryptanalysts that designs RNGs, so that they may evaluate the randomness of the output of these. χ^2-distributions (Chi-square), poker tests and run lengths can be used to accurately estimate whether the data analyzed is random or not. These methods can only say if the data is *statistically random*, but several of these tests are sensitive to correlation and other factors that indicate non-random data.

An idea is to utilize these test to analyze key structures in memory, and generate signatures and methods to identify random data (e.g., keys or ciphertext). We will briefly go through some of these tests here; the simple statistics tests like counting, poker and runs are usually able to identify pseudo-randomness, while the more advanced like χ^2 and arithmetic mean are more sensitive to the predictability of the data. Therefore they can indicate the quality of the pseudo-randomness, and if it is cryptographically secure. All test data in this section is assumed to be 20 000 bits, and all tests has a error probability of

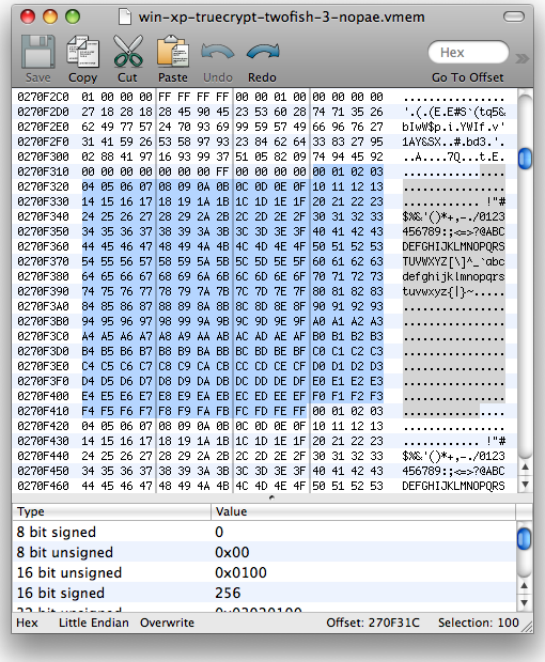


Figure 2.5: Example of a (non-random) high-entropy region in memory.

10^{-6} unless another value is mentioned. While the tests are described as applicable for bit-level granularity, the tests can usually be performed using bytes or `DWORD`s instead.

Runs Test A run is a sequence of bits of the same value, either "`0`"s or "`1`"s. The runs lengths tested are normally from 1-6 bits, and the test is passed if the counted number of such runs falls within an acceptable interval. National Institute of Standards and Technology (NIST) has among other specified some acceptable intervals [64] for runs testing, which are reproduced in Table 2.1.

Run Length	Interval
1	[2267, 2733]
2	[1079, 1421]
3	[502, 748]
4	[233, 402]
5	[90, 223]
6	[90, 233]

Table 2.1: Tolerance intervals for runs of various lengths.

Long Runs Test The long runs test are a variation of the runs test, testing for runs with lengths of 34 or longer. These runs should not exist, and the test

fails if any is encountered.

Monobit Test This test simply counts the number of "`0`"s and "`1`"s, and as we discussed earlier, these numbers should be roughly the same. NIST recommends that the test should pass if the number of "`1`"s falls within $[9654, 10346]$ for 20000 bits.

Poker Test The poker test divides the sequence of bits into four-bit segments and counts the frequency (denoted f_i) of each of the $2^4 = 16$ possible values. The test is passed if the value

$$X = \frac{16}{5000} \sum_{i=0}^{15} f_i^2 - 5000 \tag{2.11}$$

lies within the interval $[1.03, 57.14]$ according to NIST.

The χ^2 Test Pearson's χ^2 (chi-square) test is probably the most used test for the randomness of data, and it is extremely sensitive to small variations in its input. Mathematically, it can be described as the statistic

$$\chi^2 = \sum_{i=1}^{t} \frac{(H(X_i) - E(X_i))^2}{E(X_i)} \tag{2.12}$$

for t distinct events X_i where the expected value $E(X_i)$ of the event X_i may be expressed as

$$E(X_i) = nP(X_i) \tag{2.13}$$

where n is the number of observations and $P(X_i)$ is the probability for the occurrence of X_i. In essence, the test gives information on *goodness of fit*, that is, how well an empirically collected probability distribution corresponds to a theoretically expected distribution. As mentioned before, a random sequence should be uniformly distributed, which means that the expected value of all $P(X_i)$ will be the same (where ω is the alphabet size):

$$P(X_i) = 1/\omega \Rightarrow E(X_i) = n/\omega \tag{2.14}$$

Whether this actually occurs can then be calculated using the χ^2 test (using 2.12 and 2.14):

$$\chi^2 = \sum_{i=0}^{\omega-1} \frac{(H(X_i) - n/\omega)^2}{n/\omega} = \frac{\omega}{n} \sum_{i=0}^{\omega-1} H(X_i)^2 - n \tag{2.15}$$

This test is repeated for samples of data (essentially partial sequences of X_i, in our case memory bytes), and the test passes if the result is within the interval $[\omega - 2\sqrt{\omega}, \omega + 2\sqrt{\omega}]$. In reality, the distribution is estimated using numerical methods, and several libraries and implementations exist to do this.

Based on the above assumptions, the error probability (that a "good" random sequence is interpreted as a "bad" one) is around two percent. We can thus relate the output of the test to a percentage of how often a truly random sequence would exceed the result value, like Walker does in his tool ENT [Wal08].

This will give an indication of how "suspect" the supposedly random sequence is of being non-random. Randomness is as mentioned hard to measure.

We will use the ENT tool to measure the randomness of several data types during this thesis, and we can already now establish that compressed data fails the chi-square test as expected (see Section 2.3.3).

Arithmetic Mean Test The *arithmetic mean* of a sequence of symbols is simply the the sum of the symbols divided by the length of the sequence. When the symbols are bytes, this value should be around $\frac{\omega}{2} = 127.5$ This is equivalent to the Monobit test for bytes.

Serial Correlation Coefficient This test measures the possible existing dependencies of the symbols in the information measured. For C code, the test will give values approaching ± 0.5, while totally uncorrelated data will have values near ± 0.0. Uncompressed bitmaps and other highly uncompressed and correlated data will give values approaching ± 1. The test passes if data is sufficiently uncorrelated. NIST specifies the following test value for a range of 10000 bits, $b_1, ..., b_{10000}$, and for a t in the range $1 \leq t \leq 5000$:

$$Z_t = \sum_{i=1}^{5000} b_i \oplus b_{i+1} \tag{2.16}$$

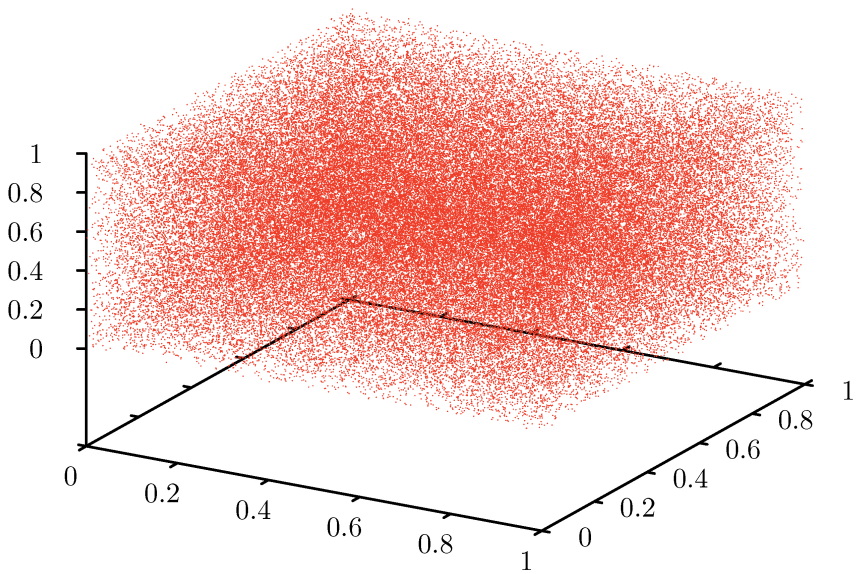


Figure 2.6: Lattice test for Unix function `rand()`.

Visual Tests In addition to the above tests, it is possible to visually spot non-uniformness by plotting the output of the function investigated (e.g., the RNG) on a 2D or 3D graph. For example, the *lattice test* is formed by plotting the output of three different instances of an RNG on a 3D map, and visually

confirming the uniform distribution. The output should take the shape of a cube, like in Figure 2.6.

The ENT Tool

John Walker's ENT tool can be used to measure the randomness of a given input. To illustrate the former discussion about randomness, we will use the program to measure the randomness of a compressed JPEG file with high entropy. Using the image in Figure 2.3(a) as input, the output of the command is as follows:

```
$ ent persistence_memory.jpg
Entropy = 7.940680 bits per byte.

Optimum compression would reduce the size
of this 7611 byte file by 0 percent.

Chi square distribution for 7611 samples is 707.44, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 126.5066 (127.5 = random).
Monte Carlo value for Pi is 3.063091483 (error 2.50 percent).
Serial correlation coefficient is 0.100115 (totally uncorrelated = 0.0).
```

The chi-square value is interpreted according to the tool description [Wal08]:

> *"The chi-square distribution is calculated for the stream of bytes in the file and expressed as an absolute number and a percentage which indicates how frequently a truly random sequence would exceed the value calculated. We interpret the percentage as the degree to which the sequence tested is suspected of being non-random. If the percentage is greater than 99% or less than 1%, the sequence is almost certainly not random. If the percentage is between 99% and 95% or between 1% and 5%, the sequence is suspect. Percentages between 90% and 95% and 5% and 10% indicate the sequence is "almost suspect"."*

As we can see the image has a high entropy (7.940680 bits per byte), but the chi-square value clearly indicate non-randomness by being "almost certainly not random".

To put this in perspective, we extracted the private exponent from a 4096-bit (512-byte) RSA key, and ran the output through ENT:

```
$ ent private_exp
Entropy = 7.601792 bits per byte.

Optimum compression would reduce the size
of this 512 byte file by 4 percent.

Chi square distribution for 512 samples is 266.00, and randomly
would exceed this value 30.51 percent of the times.

Arithmetic mean value of data bytes is 128.9434 (127.5 = random).
Monte Carlo value for Pi is 3.058823529 (error 2.63 percent).
Serial correlation coefficient is -0.011168 (totally uncorrelated = 0.0).
```

The exponent appears random to all of the tests, except the Monte Carlo value that has and error of 2.63 percent. This is however due to the slow

Reference	Magnitude
One million (10^6)	2^{20}
Seconds in a year	2^{25}
Global population	2^{32}
Age of universe	2^{34} years
1 MIPS Year (MY)	2^{45} operations
1 Sony PlayStation 3 Year (230400 MIPS)	2^{63} operations
Estimated number of protons in the universe	2^{256}

Table 2.2: Reference for large numbers.

convergence nature of the Monte Carlo test (and is also why it is not covered in this thesis). Thus the sequence is essentially random, which is as expected. In contrast, if we input the whole DER-encoded (see Section 2.3.5) private key, we see that the tests are extremely sensitive to non-randomness, and that the small tags from the DER encoding affect the chi-square output of ENT:

```
$ ent private_key.der
Entropy = 7.895050 bits per byte.

Optimum compression would reduce the size
of this 2349 byte file by 1 percent.

Chi square distribution for 2349 samples is 370.66, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 124.6607 (127.5 = random).
Monte Carlo value for Pi is 3.058823529 (error 2.63 percent).
Serial correlation coefficient is 0.042607 (totally uncorrelated = 0.0).
```

Consequently we need to adopt advanced techniques to use entropy and randomness evaluation methods to find such keys in memory.

2.3.4 Key Length

While the randomness of crypto keys are important, key lengths are also vital to the security of the cipher. Since any attacker can launch a brute-force key-guessing attack on a cipher key, it should be long enough to make this approach computationally infeasible.

The numbers in Table 2.2 represent some examples of large numbers, provided to present some context to the discussion around key lengths. The numbers treated in cryptography are extremely big, often beyond human comprehension. As we can see, brute-forcing a 256-bit AES key is roughly equivalent to searching for one particular proton in the entire universe. This strongly imply that a symmetric key of this size likely will withstand all foreseeable increases in computing power.

Symmetric Key Length

Symmetric keys are in general only susceptible to brute-force attacks, as they does not have any internal structure and mostly can be interpreted as random sequence of bytes. To mount such an attack, the attacker must have access to a

small amount of ciphertext and its corresponding plaintext. The attacker does not need large amount of plaintext/ciphertext pairs, often a known header in a Transmission Control Protocol (TCP) packet or the header of a known filetype is enough. As a result, the space complexity of the attack is virtually zero. The time complexity of the attack is directly dependent on the key size; a 56 bit DES key has 2^{56} possible permutations, and the expected time used searching for the key is $\frac{1}{2}(2^{56}) = 2^{55}$ since the key statistically will be found on the half way.

To infer anything usable from the above, we need to compare these numbers with the numbers in Table 2.2. First of all, to say anything about the security of the cipher, we assume that the adversary knows every detail of the cipher we use, and that he has access to vast amounts of plaintext/ciphertext pairs. Now recall an important principle from Section 1.2: The security of the cipher should rest in the key, and not the algorithm design. Clearly this implies that for the cipher to be secure, we need a key size that will withstand such a brute-force attack.

So how long will it take to mount a successful brute-force attack on a 56-bit key? The answer is that it depends on the approach; the search does not need to be sequential. The attacker may divide the key range in segments and assign each to a devoted chip or distributed computer. With enough money, dedicated hardware may crack such keys far faster than a software-based approach. Notably, the COPACOBANA project [Cop07] and the Electronic Frontier Foundation (EFF) DES Cracker [65] are able to crack 56-bit DES in 6.4 days and 22 hours, respectively (COPACOBANA has a much lower cost associated). Other approaches uses more easily available hardware to perform similar attacks. For example, recent research suggest that 52 PlayStation 3 consoles can be used to crack DES in 9 days using 30,056 Euro as a one-time cost [66].

But the time complexity raises exponentially, and not everyone has resources or wits to construct their own hardware cracker. Diffie argues in his paper *Ultimate Cryptography* that even with a breakthrough i quantum computing, key sizes up to 250-400 bits should suffice in the future [67]. Brute-forcing these has a time complexity way beyond our apprehension, and even if we were able to harvest all the power from the sun and other stars and channelize all this energy into the task of breaking the keys, we would be faced with a mind-boggling number of years of waiting [7].

So faced with 56, 128, 192 or 256-bit keys and a strong cipher, what should one chose? Of course, this depends on the value of the data, available resources and performance demands. 56-bit keys are insufficient for general security, while 128-bit is the clear choice for now. This gives 2^{128} different keys and an expected brute-force time complexity of $2^{127} \approx 1.70*10^{38}$, which makes it 2^{71} times harder to guess than the 56-bit key. If the system is designed to stand the test of time, a 256-bit key would be a wise choice.

Public-Key Key Length

Public keys are vulnerable to other attacks than symmetric keys. Since the public modulus of an RSA key is the product of two large primes p and q, one way of attacking the key is to try to factor the modulus and recover the private key d. This is a *hard* problem, given large enough modulus. Most public-key system security rely on *one-way functions* that are easy to perform one way,

Year / Cipher	**AES-128**	**AES-192**	**AES-256**
2001	2644 3224	6897 7918	13840 15387
2010	2942 3560	7426 8493	14645 16246
2020	3296 3956	8042 9160	15574 17235
2030	3675 4379	8689 9860	16538 18260

Table 2.3: "AES-security"-matching RSA modulus sizes. All sizes in bits.

but computationally infeasible the other way. It is therefore interesting to see how fast one can solve such problems of different sizes.

Factoring is hard, but it is getting easier. The General Number Field Sieve (GNFS) [68, 69], the currently fastest factoring algorithm for large numbers [70], is constantly improving its performance. At the time of writing, the current record is factoring a 200 digit (corresponding to 663 bits) number in 3 months on a cluster of 80 2.2 GHz Opterons [71], achieved by Bahr, Boehm, Franke and Kleinjung May 2005. Clearly, this makes 512 bit RSA insufficient for security.

Lenstra points out that to mach AES-128 security, a RSA key length in the interval $[2942, 3560]$ bits is needed in 2010 [72]. A summary adapted from his paper is shown in Table 2.3, where the predicted sizes are represented in two columns for each AES key length: One minimum and one conservative value. It is of course not sensible to make predictions in this field of study, since history has shown that new methods for factorization very well may be invented. This implies that to select a key size, one have to know the resource level of an adversary, and choose large enough key sizes accordingly.

However, when searching for keys in memory, we face the opposite problem (at least when using the entropy approach): The longer the key, the easier it is to locate; large amounts of data are needed to accurately estimate the entropy. We will discuss this further in Section 2.3.5.

2.3.5 Key Management

When users are given the option to choose their own passwords (and thereby keys), the tend to choose strings based on birthdays, pets, football teams or anything else that helps remembering them. This can undermine the security of even the most secure cryptosystem.

Instead, users should be encouraged to choose passphrases or abbreviated passwords that contain as many different types of ASCII characters as possible. Words from a dictionary should be kept out, and also information that can be tied to the person owning the key. Of course, the user wants 'Bob83' as his password, it is his or her choice.

In addition, a proper *key churning* process like the one described in the following section should be undertaken by the cryptographic application, so that the password selected by the user not is used directly in the encipherment process. A good cryptographic practice is to never use the master key for any encryption tasks at all, but instead derive keys from it or encrypt the encryption keys used with the master key.

Key management is therefore a difficult task. Keeping the key secret is what the security of the system depends on, and keeping it secret often depend on

external and less controllable entities like users, operating systems and applications. In this section we will consider the different usage and storage options for keys that are relevant to this thesis and memory forensics.

Generating Keys

Key generation or churning is not something you do to create easy-to-remember keys, but rather a random byte-string of a given length. The U.S. Department of Defense recommends DES in Output FeedBack (OFB)-mode, with an Initialization Vector (IV) created from an amount of state indicators at the generating computer (e.g., registers, system clock and counters) [73]. The plaintext can be typed in by hand by an administrator, for example a 8-character password. The output of the cipher is then used as key.

There exist many ways of doing key generation, and we will not cover them here. The important lesson is that the key should essentially be as random as possible, to prevent better-than-brute-force attacks. In the case where parts of the key has mathematical properties, these should of course be tested accordingly.

Key Storage

Usually a cryptographic key can be stored in two distinct[4] places on a computer; in physical memory (RAM) during usage, and on secondary memory or other non-volatile memory at all other times. We focus on the former scenario, but cannot exclude the possibility of encountering keys on the hard drive. In a real investigation, effort should be laid down to search in all available storage mediums for keys.

The storage of keys on secondary memory is clearly a security versus usability tradeoff, ideally no such storage should be performed, and the user should be required to present the password or key every time it is needed. This is obviously not feasible in many forms of operation, for example consider a SSL web server that needs its private key every time it receives a HTTPS request.

Instead many applications encrypt their keys using a user-supplied password, decrypts it at startup and keeps the decrypted key in memory while the cryptographic application is running. The storage format and encoding on disk varies from application to application (and even within applications), but in memory the processor is usually dependent on a raw byte representation to make quick use of the key.

Trusted Computing and the TPM Chip Another approach to key storage is to store them at a tamper-resistant device or a token. This is one of the ideas behind the Trusted Platform Module (TPM) chip, that comes pre-installed at most new computers today. The initiative lobbying the chip, the Trusted Platform Group (TPG), assess that the TPM can among other things be used for key storage [74]. The TPM keeps track of the state of the computer, and will only release the key if the computer is in a "trusted state"; this is to prevent unauthorized use and malware modifications of the OS.

[4]Keys may be subject to swapping, and hence may be residual at secondary storage. In that case, the borders between secondary and primary memory is not as clear (from our viewpoint).

The TPM thus aims to counter the untrusted nature of computers today, by tracking the different states during boot and operation. The chip itself is tamper-resistant, which means it is hard to desolder or read the chip without destroying it. Thus, the confidentiality and integrity of the data encrypted with this key is protected not only by the state of the computer, but also by a hardware layer of security.

Unfortunately, recent research shows that in some cases the TPM makes whole-disk encryption systems like Microsoft BitLocker less secure, because the key is loaded into memory at boot time, before login. This allows an attacker to perform a coldboot attack even if the computer is powered off at the point in time where it is seized [30].

As an example on how keys are stored and handled, we will discuss RSA, AES, Serpent and Twofish keys and their representation in memory, and how the representation may be utilized to generate search signatures that are able to accurately extract keys from memory dumps.

RSA Key Storage Private RSA keys are mostly stored represented as .PEM files, where the key data resides, base64 encoded for portability, between two textual delimiters:


```
-----BEGIN RSA PRIVATE KEY-----
MIIBPAIBAAJBAJnAyV66JVf2EGyIf7xzqVSwPVWD912gOiO2UiHlUXmXkY9b4Nbp
4kpUyRhliaDWSV2yuOpq3EAJMEbPLb2pYL8CAwEAAQJBAJVwTcimQDmYTipPGWg/
Oqu3iEWfuEPlweXDOFxlmKUGbKTGdgzwixkoD4GCyODlQqJ9vhkaSgYOGISVBoVK
hNECIQDIDQTY1ALDqAND/5OCFHRVI7nmgqVLblME2UkNdsxwFQIhAMTA/8zlEEov
TI7O+Yp2nzR1YwixplmSt8ZhWzklF26DAiEAlpaeQG4PiqLNmoEn07J8A576EFfl
/4sTuUGrKRR1PiUCIQC6sAYHfDGAsnCJ8ImGgBd/xwI49ZdJ1pTZfvb3ueIJOQIg
QkOyt4ZWMOSjYCOKe6ZnQ+5IXeh2dfldfE9qBCLzyDO=
-----END RSA PRIVATE KEY-----
```


When the key is stored at disk, it is usually encrypted using a password. OpenSSL uses DES, 3DES or AES encryption for this. When the key is used however, the key is decrypted, base-64 decoded and used in its plain form, which is ASN.1 [75] DER[5] encoding as specified in PKCS #8 [76, 14]. The ASN.1 encoding specifies several data types markers that are used in the encoding process of the key, among others `SEQUENCE` (represented by the hexadecimal value `0x3082`) and `INTEGER` (which corresponds to `0x02`). These identifiers are used to identify different instances of data values and their properties (size, etc.) in a DER-encoded file. We will discuss how to leverage these properties for cryptographic key search in Section 6.1.5.

The AES Key Schedule

In the paper *Lest We Remember: Cold Boot Attacks on Encryption Keys* [30], Halderman et al. uses the properties of the *AES key schedule* to search for AES keys in memory. The key schedule (sometimes called round key or key expansion) is an array of keys derived from the master key, each key used in the separate rounds of the cipher. This key schedule is often computed ahead of time, in what appears to be a security-performance tradeoff. We will briefly explain the AES 128-bit key schedule here, the approach for 198 and 256 bit

[5] *Distinguished Encoding Rules*, a subset of *Basic Encoding Rules* (BER), set by the ASN.1 standard.

keys is in principal the same, albeit with slightly modified key schedules. For more on the AES key schedule, see [12, 13] or [38].

The key schedule uses some of the common operation in Rijndael's Galois field:

- The `rotate` operation, a 8-bit circular rotate on a 32-bit word
- The `rcon` operation; 2 exponentiated to a user supplied value in the Galois field
- The S-boxes, `sbox`
- A key schedule routine `schedule_core`

Basically, the inner loop of the key schedule routine `schedule_core` performs the following operations (for a 128-bit key):

1. Take in an input of a 32-bit word and an iteration number `i`
2. Copy the input over to the output
3. Use `rotate` on the output
4. Apply `sbox` on all four individual bytes in the output word
5. On the leftmost byte of the output, XOR the byte with $2^{\texttt{rcon(i)}}$

The actual key expansion, the `expand_key` operation, uses these operations to expand the 128-bit (16 bytes) to a full 176 bytes key schedule:

1. The first 16 bytes is the master key
2. The iteration value `i` is set to 1
3. Until we have 176 bytes of key schedule, do:
   (a) To create the first four bytes, do:
      i. Create `t`, a four-byte temporary value
      ii. Give `t` the value of the proceeding four bytes
      iii. Perform `schedule_core` on `t`, with `i` as iterator value
      iv. Increment `i` by one
      v. Xor `t` with for four-byte block 16 bytes before the new expanded key. This becomes the next four bytes in the key schedule
   (b) To create the next 12 bytes of the key schedule, do the following three times:
      i. Assign the value of the proceeding four bytes to `t`
      ii. Xor `t` with for four-byte block 16 bytes before the new expanded key. This becomes the next four bytes in the key schedule

There exists several test vectors for this operation, for example, the *empty key* `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00` generates the following key schedule:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63
9b 98 98 c9 f9 fb fb aa 9b 98 98 c9 f9 fb fb aa
90 97 34 50 69 6c cf fa f2 f4 57 33 0b 0f ac 99
ee 06 da 7b 87 6a 15 81 75 9e 42 b2 7e 91 ee 2b
7f 2e 2b 88 f8 44 3e 09 8d da 7c bb f3 4b 92 90
ec 61 4b 85 14 25 75 8c 99 ff 09 37 6a b4 9b a7
21 75 17 87 35 50 62 0b ac af 6b 3c c6 1b f0 9b
0e f9 03 33 3b a9 61 38 97 06 0a 04 51 1d fa 9f
b1 d4 d8 e2 8a 7d b9 da 1d 7b b3 de 4c 66 49 41
b4 ef 5b cb 3e 92 e2 11 23 e9 51 cf 6f 8f 18 8e
```

As a comparison, here is a hexadecimal dump representation of a real 128-bit key (`b6 e4 48 2d c1 bd 00 89 3f 02 f9 dd 5d a5 10 22`) found in the memory of Windows Vista:

```
$ hexdump -C vista-simp-key-2
00000000  b6 e4 48 2d c1 bd 00 89  3f 02 f9 dd 5d a5 10 22  |..H-....?...]..."|
00000010  b1 2e db 61 70 93 db e8  4f 91 22 35 12 34 32 17  |...ap...O."5.42.|
00000020  ab 0d 2b a8 db 9e f0 40  94 0f d2 75 86 3b e0 62  |..+....@...u.;.b|
00000030  4d ec 81 ec 96 72 71 ac  02 7d a3 d9 84 46 43 bb  |M....rq..}...FC.|
00000040  1f f6 6b b3 89 84 1a 1f  8b f9 b9 c6 0f bf fa 7d  |..k............}|
00000050  07 db 94 c5 8e 5f 8e da  05 a6 37 1c 0a 19 cd 61  |....._....7....a|
00000060  f3 66 7b a2 7d 39 f5 78  78 9f c2 64 72 86 0f 05  |.f{.}9.xx..dr...|
00000070  f7 10 10 e2 8a 29 e5 9a  f2 b6 27 fe 80 30 28 fb  |.....)....'..0(.|
00000080  73 24 1f 2f f9 0d fa b5  0b bb dd 4b 8b 8b f5 b0  |s$./.......K....|
00000090  55 c2 f8 12 ac cf 02 a7  a7 74 df ec 2c ff 2a 5c  |U........t..,.*\|
000000a0  75 27 b2 63 d9 e8 b0 c4  7e 9c 6f 28 52 63 45 74  |u'.c....~.o(RcEt|
000000b0
```

The Serpent Key Schedule

Serpent's key schedule has a format similar to the AES key schedule; it uses its user supplied key as the first round key, with the following round keys derived from this master key. It also uses functions from the cipher to calculate the round keys, by utilizing its S-boxes.

If the master key supplied is smaller than 256 bits, the key is padded by appending a "`1`" bit to the Most Significant Byte (MSB) end, followed by as many "`0`" bits as necessary to make up 256 bits. The cipher needs 132 32-bit words of key material, hence we need to derive 33 128-bit sub keys $K_0, ..., K_{32}$ from the master key. The derivation process can be described as follows, based on the discussion in [48, 44]:

1. Set the value of the first two sub keys, K_1 and K_2, to each half of the user-supplied master key

2. Expand the key up to 256 bits if necessary as explained above

3. Treat the key as 8 32-bit words $w_{-8}, ..., w_{-1}$ and expand it to a *prekey* $w_0, ..., w_{131}$ by the following transformation:

$$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) << 11 \qquad (2.17)$$

   where ϕ is the fractional part of the golden ratio $(\sqrt{5}+1)/2$.

4. Calculate the round keys using the eight S-boxes. The words of the final key schedule, $k_0, ..., k_{131}$ are calculated in the following way:

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &= S_3\{w_0, w_1, w_2, w_3\} \\
\{k_4, k_5, k_6, k_7\} &= S_2\{w_4, w_5, w_6, w_7\} \\
\{k_8, k_9, k_{10}, k_{11}\} &= S_1\{w_8, w_9, w_{10}, w_{11}\} \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &= S_0\{w_{12}, w_{13}, w_{14}, w_{15}\} \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &= S_7\{w_{16}, w_{17}, w_{18}, w_{19}\} \\
&\ldots \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &= S_3\{w_{128}, w_{129}, w_{130}, w_{131}\}
\end{aligned}$$

5. Renumber the 32-bit words into 128-bit keys of the form

$$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$$

The result is a 560-byte array of the master key together with 33 derived round keys. As an example, the following is a key schedule found in a Windows XP memory image using Truecrypt Serpent with key
`6f 69 97 c5 40 ff ff d3 c0 22 ce f8 6e c4 3c 54`
`41 5d 26 95 95 5e 2d b5 fc 5a 1a ee 57 dd 95 3d`:

```
6f 69 97 c5 40 ff ff d3 c0 22 ce f8 6e c4 3c 54
41 5d 26 95 95 5e 2d b5 fc 5a 1a ee 57 dd 95 3d
9b 56 94 5c 21 f0 c0 09 a7 9a 2d 56 24 8e 46 2d
57 c0 f1 d6 61 ee 5f 60 ed 72 f7 63 e4 be 53 ae
86 d7 cc e8 96 09 eb 59 e6 23 e4 10 4b d0 63 14
5c c4 76 75 56 83 9d be 21 d5 79 cd c0 af a5 b4
7e 9f 1e 43 af c2 34 78 e0 ce 3c 39 2d ff 69 a7
b1 91 d2 86 1e fb d4 7d f8 a8 70 19 cb 35 36 8e
ca 99 78 14 40 6c 3e 0f 59 58 d3 df f8 fd 9a 58
79 94 99 13 9d d8 4d 7a 03 86 e5 6a 16 8d 06 01
85 8e f3 9c 97 bf f5 d2 79 ec 9a b7 35 bf 09 6b
03 23 ed 0f e3 8b 3e e2 6c af 31 71 71 59 d6 80
5a 75 c7 c4 d2 28 88 91 42 16 b5 04 97 99 9a 9d
ac 82 94 71 3e 50 cd 33 6b 94 9a e4 cf 33 a8 5e
3a 87 93 f5 8b 94 f4 4a e0 8a fe f4 3d f0 26 bc
61 89 8e af 13 63 42 6a 91 64 ad 40 aa dc ec 1c
79 c4 44 c4 e0 b0 3c 46 c1 f9 1e a9 b9 1c ec db
dc b1 21 78 29 77 a6 81 d6 fb 83 5c db 8c 55 76
bb 33 52 6d 66 41 65 f2 4c e1 9f 6a 4c 25 c1 2a
7f ac 71 b8 e4 18 b2 6b 8d 87 92 38 1c 86 61 98
2e 6b 0b f2 02 f0 71 38 eb 6e 36 fb 81 4d ca fc
17 d3 f8 ef f6 fc 97 a4 3b 47 a0 e6 f6 5a e5 e9
29 ae 85 d2 95 3b c2 50 e4 74 8a 0d c1 b1 6c dd
88 11 b3 5d eb 24 4e b2 b0 70 b1 99 68 4d c3 77
61 9c 9d 7b 97 7f 58 46 24 ca 18 16 15 87 f8 bb
8a 66 a3 17 5a cc 6a 55 6e eb de d0 27 91 b1 bb
b8 25 8c c7 57 4b bc 80 a3 cb 67 95 f0 2e 8a b3
d5 c7 b7 7a a9 67 a7 6a 1a 25 14 d9 32 dd fb 37
8a e4 34 e3 69 d5 1d 18 b5 7a 0e fc 2e 5a ee 87
72 5d 1b 6d b7 cc 19 82 00 c1 14 6c 7c 83 8a a4
0d a8 33 77 48 3c 88 21 64 88 fa 53 19 ae 7f 89
67 1d 84 66 3e 17 c4 11 8e 92 1f b3 45 14 6b ed
a7 db f5 35 c5 1a 67 8c 11 a2 8c c8 38 6c 58 e5
e8 bb 2b eb 68 ec 6e e6 e6 86 57 d2 09 23 c1 dc
80 87 7c 0a d7 71 24 39 c1 84 0e bf 12 ae db 3b
```

Here the two first 16-byte vectors are the 256-bit master key, and the 33 remaining rows the 128-bit sub keys.

The Twofish Key Schedule

Twofish uses a slightly different approach than AES and Serpent, by utilizing key-dependent S-boxes together with round keys in the encryption process [45]. Twofish is a 16-round feistel-based structure with additional input and output whitening, where the keyed S-boxes are combined with a Maximum Distance Separable (MDS) matrix and a Pseudo-Hadamard Transform (PHT) to form the core of each round, resulting in a far more complex key schedule than the last two examples. The cipher can operate with keys of length $N = \{128, 192, 256\}$ bits.

If the algorithm is compiled for a modern-day computing device with sufficient amounts of memory, it also combines several of the operations and represents them as a 4 KB table in memory. This is mainly done because of performance reasons, and the resulting encryption operation reduces itself to only four table lookups and three XORs [47].

The size of this table both makes Twofish keys both easier and harder locate. Easier since the 4 KB table makes an excellent search signature, and harder because the size of the whole key schedule exceeds 4096 bytes, which is the usual size of a page in memory. The key schedule may therefore be scattered over several pages at different locations in the physical memory. We will treat this problem more in-depth in Section 6.1, for now we assume that the whole key schedule occupies a continuous address space in the memory investigated.

The full key schedule consists of 40 32-bit words of expanded key $K_0, ..., K_{39}$, the keys for the S-boxes and the optional 4 KB table that merges the S-box lookup and MDS matrix multiplication. Before discussing the key schedule generation, we briefly describe the MDS matrix, an error correcting code matrix RS, the function h and permutations q_0 and q_1 which are all needed to calculate the schedule.

MDS Matrix The MDS matrix is given by:

$$MDS = \begin{bmatrix} \texttt{01} & \texttt{ef} & \texttt{5b} & \texttt{5b} \\ \texttt{5b} & \texttt{ef} & \texttt{ef} & \texttt{01} \\ \texttt{ef} & \texttt{5b} & \texttt{01} & \texttt{ef} \\ \texttt{ef} & \texttt{01} & \texttt{ef} & \texttt{5b} \end{bmatrix}$$

where the notation is in hexadecimal form.

RS Matrix This matrix (abbreviated from Reed-Solomon, an error-correcting code) is defined in [45] as:

$$RS = \begin{bmatrix} \texttt{01} & \texttt{a4} & \texttt{55} & \texttt{87} & \texttt{5a} & \texttt{58} & \texttt{db} & \texttt{9e} \\ \texttt{a4} & \texttt{56} & \texttt{82} & \texttt{f3} & \texttt{1e} & \texttt{c6} & \texttt{68} & \texttt{e5} \\ \texttt{02} & \texttt{a1} & \texttt{fc} & \texttt{c1} & \texttt{47} & \texttt{ae} & \texttt{3d} & \texttt{19} \\ \texttt{a4} & \texttt{55} & \texttt{87} & \texttt{5a} & \texttt{58} & \texttt{db} & \texttt{9e} & \texttt{03} \end{bmatrix}$$

The Function *h* The function h takes a 32-bit word X and a list $L = \{L_0, L_{k-1}\}$ of 32-bit words as input (where k is defined as $k = N/64$), and produces a single word of output. The function works in k stages. In each stage, one must perform the following operations:

1. Split X and L into bytes:

$$\begin{aligned} l_{i,j} &= \lfloor L_i/2^{8j} \rfloor \mod 2^8 \\ x_j &= \lfloor X/2^{8j} \rfloor \mod 2^8 \end{aligned}$$

   for $i = 0, .., k-1$ and $j = 0, ..., 3$.

2. While $y_{k,j} = x_j$, apply the following sequence of substitutions and XORs:

   (a) If $k = 4$, do:

$$\begin{aligned} y_{3,0} &= q_1\left[y_{4,0}\right] \oplus l_{3,0} \\ y_{3,1} &= q_0\left[y_{4,1}\right] \oplus l_{3,1} \\ y_{3,2} &= q_0\left[y_{4,2}\right] \oplus l_{3,2} \\ y_{3,3} &= q_1\left[y_{4,3}\right] \oplus l_{3,3} \end{aligned}$$

   (b) If $k \geq 4$, do:

$$\begin{aligned} y_{2,0} &= q_1\left[y_{3,0}\right] \oplus l_{2,0} \\ y_{2,1} &= q_1\left[y_{3,1}\right] \oplus l_{2,1} \\ y_{2,2} &= q_0\left[y_{3,2}\right] \oplus l_{2,2} \\ y_{2,3} &= q_0\left[y_{3,3}\right] \oplus l_{2,3} \end{aligned}$$

   (c) For all cases, do:

$$\begin{aligned} y_0 &= q_1\left[q_0\left[q_0\left[y_{2,0}\right] \oplus l_{1,0}\right] \oplus l_{0,0}\right] \\ y_1 &= q_0\left[q_0\left[q_1\left[y_{2,1}\right] \oplus l_{1,1}\right] \oplus l_{0,1}\right] \\ y_2 &= q_1\left[q_1\left[q_0\left[y_{2,2}\right] \oplus l_{1,2}\right] \oplus l_{0,2}\right] \\ y_3 &= q_0\left[q_1\left[q_1\left[y_{2,3}\right] \oplus l_{1,3}\right] \oplus l_{0,3}\right] \end{aligned}$$

   where permutations q_0 and q_1 will be explained in the next paragraph.

3. Multiply the resulting vector $Y = [y_0, ..., y_3]$ with the MDS matrix:

$$Z = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} \cdot & \cdots & \cdot \\ \vdots & MDS & \vdots \\ \cdot & \cdots & \cdot \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

4. Return Z

The Permutations *q_0* and *q_1* The functions q_0 and q_1 are fixed 8-bit permutations. For input value x, output value y is defined as follows in [45]:

$$\begin{aligned}
a_0, b_0 &= \lfloor x/16 \rfloor, x \mod 16 \\
a_1 &= a_0 \oplus b_0 \\
b_1 &= a_0 \oplus (4 >> b_0) \oplus 8a_0 \mod 16 \\
a_2, b_2 &= t_0 [a_1], t_1 [b_1] \\
a_3 &= a_2 \oplus b_2 \\
b_3 &= a_2 \oplus (4 >> b_2) \oplus 8a_2 \mod 16 \\
a_4, b_4 &= t_2 [a_3], t_3 [b_3] \\
y &= 16b_4 + a_4
\end{aligned}$$

where $t_0, ..., t_3$ are 4-bit S-boxes, different ones for q_0 and q_1, respectively.

Generating the Key Schedule Finally, the creation of the key schedule can be defined. To expand the key into 40 32-bit words, perform the following steps:

1. Split the key M into vectors with its even and odd bytes M_e and M_o:

$$\begin{aligned}
M_e &= \{M_0, M_2, ..., M_{2k-2}\} \\
M_o &= \{M_1, M_3, ..., M_{2k-1}\}
\end{aligned}$$

2. Derive vector S, by by taking the key bytes in groups of 8, interpreting them as a vector over $GF(2^8)$, and multiplying them with the RS matrix:

$$S_i = \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{bmatrix} = \begin{bmatrix} \cdot & \cdots & \cdot \\ \vdots & RS & \vdots \\ \cdot & \cdots & \cdot \end{bmatrix} \cdot \begin{bmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{bmatrix}$$

3. Interpret each result of 4 bytes as a 32-bit word of the vector S for $i = 0, ..., k-1$. These are the S-box keys, and the vector is used in the "reverse" order, e.g.:

$$S = \{S_{k-1}, ...S_0\}$$

4. Then expand the key and form expanded keywords K_j:

$$\begin{aligned}
\rho &= 2^{24} + 2^{16} + 2^8 + 2^0 \\
A_i &= h(2i\rho, M_e) \\
B_i &= (h((2i+1)\rho, M_o)) << 8 \\
K_{2i} &= (A_i + B_i) \mod 2^{32} \\
K_{2i+1} &= ((A_i + 2B_i) \mod 2^{32}) << 9
\end{aligned}$$

Finally, if specified, the 4 KB table is generated based on the S keys. This quite complicated generation procedure generates a large amount of keying material. Worse, for the sake of our research, Twofish does not use its master key as part of this material. Thus we cannot use the similar procedure as in AES and Serpent to search for keys.

Notes on the Twofish Key Schedule Early in the AES selection process, certain notes were made on the Twofish key schedule by the authors of the algorithm [77]. Furthermore, Mirza and Murphy published some interesting properties of the key schedule, namely that the S-box keys did not have an uniform distribution, but rather would seem to follow a Poisson distribution with mean 1 [78].

The Twofish team quickly researched the matter, and later proved that the properties did not affect the security of the cipher [79]. However, in Section 6.1.5 we will leverage these properties to locate Twofish keys in volatile memory.

2.4 Implementing Cryptography

Implementation of cryptography is not a task for the faint-hearted. In addition to the security of the underlying algorithm, other properties like good coding practice, code verification, key storage, large numbers arithmetics and key handling all have large impact on the overall security of the implementation of a cipher.

It is therefore generally not recommended to implement own ciphers, or even implement own versions of scrutinized, existing ciphers. If one after these words of warning still has to use own code and not use existing libraries, it is important to be aware of certain cryptography-specific issues. On software implementations, the cipher is at mercy of the underlying software—often closed-source OSes that offer limited security guarantee for precious data like keys. These issues may have an impact on our research, because failing to address them creates a larger window of opportunity when searching for keys in memory.

2.4.1 Purging Keys From Memory

Good cryptographic applications should purge or wipe keys and plaintexts from memory as soon as they are no longer needed. For some applications, keys must reside in memory while the applications is running, and in that case they should be purged the moment it terminates.

2.4.2 Compiler Optimizations

Welschenbach notes that even if data is deleted in the code of the application, compiler optimizations may still thwart the effort [38]. Consequently, is not sufficient to dereference a pointer to the data or set the value of the data to zero, since we have no guarantee that the data will be overwritten. The behavior of code like:

```
void a_function() {
    unsigned char *secret;
    struct key_info *key;
    ...
    /* Overwrite varibles (Not compiler-safe)*/
    secret = 0;
    memset(key, 0, sizeof(key));
}
```

are entirely compiler- and OS-dependent, and thus cannot be trusted.

Both the `memset()` function and the zero assignment above is simply ignored by most compilers if optimization switches like GCC `-O2` are used, which are often true for release binaries. This is perfectly reasonable for optimization purposes, since there is no need to explicitly overwrite memory with zeroes if the data is not to be referenced again. This leads to the possibility of sensitive data residing in memory or pagefiles even after the termination of the crypto application.

Therefore, compiler-safe purging methods are essential to any cryptographic implementation. To make the above method compiler safe, one would attempt to explicitly overwrite the content of the data before freeing any memory, by calling a dedicated purging function:

```
void purge_keys(unsigned char *secret, struct key_info *key) {
    *secret = 0;
    *key = memset(key, 0, sizeof(key));
}

void a_function() {
    unsigned char *secret;
    struct key_info *key;
    ...
    /* Overwrite varibles (Compiler-safe)*/
    purge_keys(secret, key);
}
```

The purging function `purge_keys()` accepts the secrets as arguments, and sets them to zero. This call cannot be ignored by the compiler on the principle of optimization strategy.

2.4.3 String Handling in Auxiliary Applications

Just like keys, plaintexts should also be purged from memory using a compiler-safe operation. Unfortunately, many cryptographic applications act as a proxy for other applications, encrypting plaintext from the other program before submitting it over an unsecured channel. The encrypting application has no control over string- and memory handling of these applications, and they can therefore not guarantee that plaintext won't be present in pagefiles or memory.

In the author's minor thesis [18], the poor string handling of applications often undermined the strong ciphers of the crypto applications. It is therefore not always easy to secure generic applications with the use of cryptography.

2.4.4 Prevention of Swapping or Paging

Software-based cryptographic software need to make sure that sensitive content like keys never are written to disk as a result of virtual memory management (see Chapter 3). There are several ways of doing this, but they are all dependent on the good behavior of the underlying OS. This is a security risk that has to be assessed when designing a software encryption tool.

2.4.5 Hardware Encryption versus Software Encryption

In addition to the notes above, hardware encryption has several benefits in terms of security when compared to software implementations. Tamper-resistant devices may secure cryptographic credentials so that they cannot be read unencrypted, and if the algorithm is hard-coded, it cannon be altered. Furthermore, the algorithm is not dependent on underlying systems in the same way as software.

While the security properties of hardware encryption are strong, its portability, cost and distribution properties are not. Hardware encryption is expensive, and if the algorithm is broken, one may be forced to buy new equipment, as opposed to download new software from an update site.

It is no secret that hardware encryption both performs faster and more secure than its software counterparts, but today the momentum of software-driven encryption is huge. To a forensic investigator, the weakness of software encryption provides an opportunity to break cryptographically secure ciphers by uncovering their keys.

Chapter 3

Windows Memory Management

The memory management system is an essential part of any modern OS. When searching for cryptographic keys in a raw memory dump, it is important to realize that the memory management procedures of the underlying OS can be exploited to identify interesting objects, pages or regions in the memory.

By using all available metadata of the memory management system to lessen the haystack, an investigator may save resources, increase hit rates and reduce false positives. Indeed, we will show that it can leverage searches for keys that would not have been possible with a brute-force approach.

The memory management systems available are as many as there are different flavors of operating systems, and a treatment of all of them is beyond the scope of this thesis. Instead, we will focus our attention towards Microsoft Windows systems, specifically Windows XP. Secondly, we will cover the relevant essentials of its memory management system, and present an approach for locating interesting data by using our knowledge of the Windows memory internals.

3.1 The Memory Manager

I order to provide memory to the multitude of processes on modern computers, the OS needs to provide a virtualization layer of memory called the virtual address space. Microsoft Windows uses such a virtual memory layout. For the rest of this discussion, the term *virtual address* will refer to an address in the virtual address space, while *physical address* will be the physical location of the data in main memory (RAM). The virtual address space facilitates sharing of the scarce resource of RAM modules between a large number of processes. As a result of this, some sort of address translation or mapping between a virtual address and a physical address is needed.

The inner workings and structures of Windows are largely undocumented, and most of the structures mentioned in this chapter are the result of reverse engineering or debugging. Many of the data structure definitions are simply examined using the Windows Debugging Tools (WDT) package with the proper symbols [Mic08a]. In Appendix B, we give a complete overview of the structures

related to memory management, as reference for specially interested individuals. For more on how to interpret these outputs, please be referred to *Microsoft Windows Internals, Fourth (or fifth) Edition* [80].

3.1.1 Introduction

Windows' memory working horse is the *Memory Manager*. The memory manager is responsible for managing the virtual and physical address space of the OS, including tasks like paging[1], memory allocation and de-allocation. Throughout this thesis we will treat the memory management system as if Windows is booted with no Page Address Extension (PAE) [80] [Mic07, Mic08b] and no /3GB switch [Old04] and the standard 4 KB page size. This is done for simplicity and coherence; the methods described could very well be implemented without these constraints[2].

Each process on 32-bit Windows has per default a 2 GB virtual address space called the *process address space*. The rest of the 4GB maximum virtual memory size is reserved for system use. The virtual address space is further divided into smaller 4 KB units called *pages*. Each page is owned by a given running process, and may be referenced using its virtual address. The memory of a process that is resident in the physical memory is called the current *working set*. When the memory becomes *overcommitted*, that is, when the applications running is trying to use more memory than is available, the memory manager is responsible for paging pages out to a *pagefile* at secondary storage, and bringing them in again when needed. The memory manager also provides services that are beyond the scope of this thesis, like execution protection and locking of memory.

The memory of a running process is thus split into pages that may or may not be resident in the physical memory. A graphical representation of this can be seen in Figure 3.1. Here, the pages of the process that were allocated at the same time (by for example calling `malloc()`) are continuous in the virtual address space, but may be scattered or not even present in the physical memory. This can have large effects on what that is to be found when considering forensics investigations of memory dumps.

3.1.2 Memory Structure

Physical memory is simply divided into 4 KB chunks of called *frames*. In Figure 3.2, we have visualized the physical memory of a Windows XP Professional computer with 256 MB of RAM by interpreting each of the memory bytes as a color value (giving a possible 256 different colors for each byte). Zero (`0x00`) is represented as the color black, and `0xff` is white. In the figure, the pattern formed by the 4 KB standard pages can be seen as vertical lines throughout the figure.

[1] Paging and Swapping is in effect the same.

[2] When the /3GB switch is set, Windows provides a 3 GB virtual address space per process instead of the default 2 GB. This results in a 1 GB system virtual address space. The /PAE switch enables Page Address Extension, a feature provided by later Intel and AMD processors. The effective result of having this switch set is several page directories per process, and hence the virtual address translation is slightly more complicated. Both these switches are set at boot time. For further reading, see [Old04] and [Mic07, Mic08b].

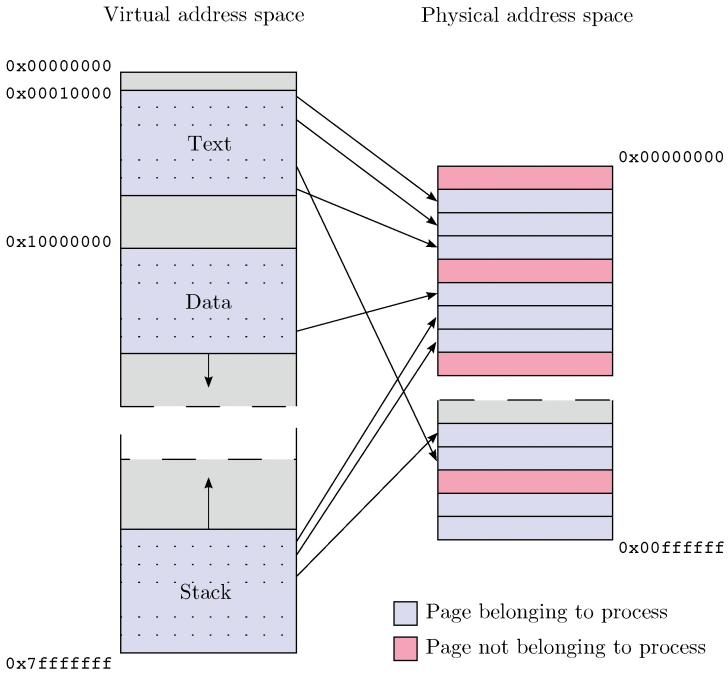


Figure 3.1: Virtual and physical address space relation.

In the virtual memory is divided into a system space and a user space, both 2 GB in size. All addresses from and above `0x80000000` are reserved for system use, and pointers to this area cannot be referenced by a non-system process thread unless they map to shared memory section. Likewise, a process thread cannot access another process' address space unless it is shared and/or the thread uses cross-process memory functions allowing access to the memory. The system space contains among other kernel code, page tables, drivers and special memory areas like the Nonpaged pool (see Section 3.1.3).

3.1.3 Paging

By the means of *paging*, the memory manager swaps pages in and out of physical memory as they are needed. The memory manager marks each page in the process address space with either free, reserved or committed. A process may reserve pages for future committing, or reserve and commit in a single call. The former is analogous to reserving space with `malloc()` or the Windows equivalent `VirtualAlloc()` in C, and later assign a value to the reserved space.

A committed page is a page that when referenced actually translates to a valid page in main memory. Any references to uncommitted pages will cause an access violation; the page is not mapped to any physical storage and the reference cannot be resolved.

When a process tries to reserve a memory range larger than the currently available physical memory, the memory manager needs to page out other pages to make space. These pages are written to a file called `PAGEFILE.SYS`, usually located in the boot partition of the system. If the pages are needed at a later

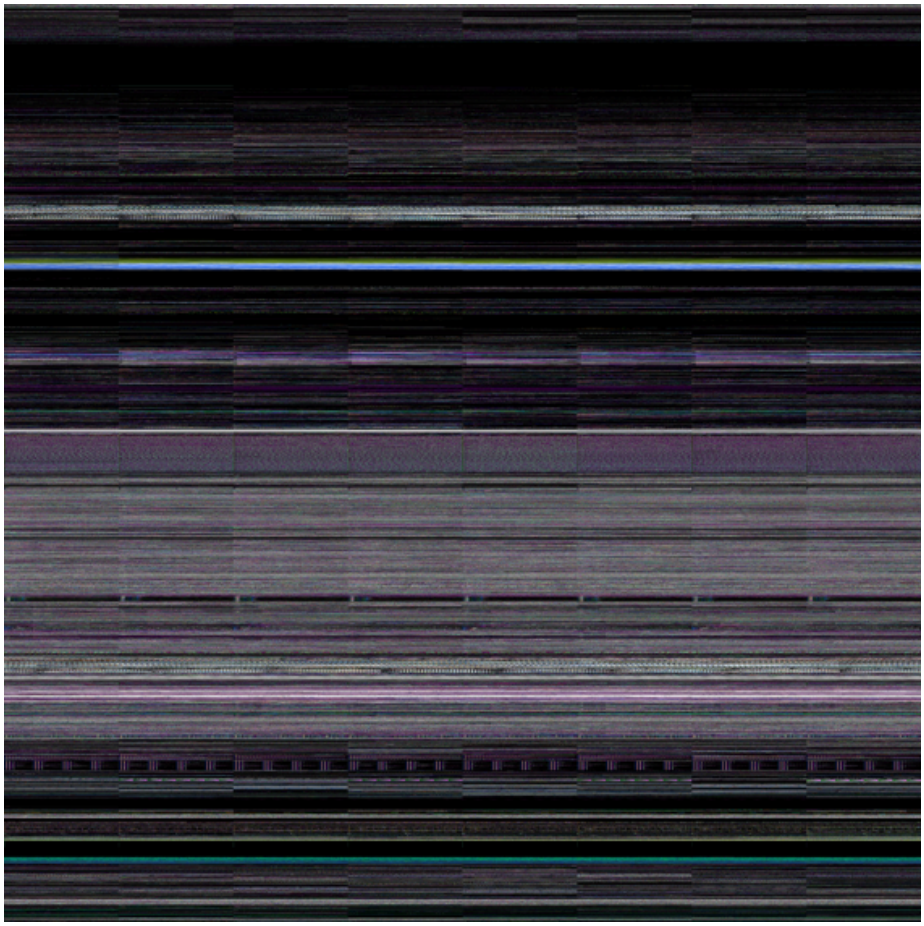


Figure 3.2: 256 MB of RAM Memory from Windows XP (running Truecrypt) visualized by interpreting each byte as a 256-color palette color. The image can be "read" from the upper left corner, row by row. The image has 8192 rows, and is 8 pages wide (8192 x 8 x 4096 = 256 MB). The border of the pages can be seen as vertical stripes in the image.

point in time, the memory manager fetches them into physical memory again.

Finally, when a process terminates or explicitly frees address space, the memory manager marks the corresponding pages as free.

The Virtual Address Descriptor (VAD) Tree

To keep track of the virtual addresses in use, the memory manager keeps a structure of Virtual Address Descriptors (VADs) to facilitate *lazy evaluation* of page tables—waiting to perform page table creation until required. A VAD describes an allocation of virtual memory, so that when the address is referenced, page tables and PTEs can be created as needed.

By keeping a self-balancing binary tree of VADs (The VAD tree), the memory manager can locate the VAD for each virtual address quickly, and perform the necessary operations when referenced. Therefore, the memory manager waits to create a page table until a page fault occurs, and then it creates a page table for that page. This method significantly improves performance for high-committing processes.

The Nonpaged Pool

Since applications often tries to allocate memory smaller than the page size, Windows provides a pool of pages that are reserved for such reservations. If no such pool was provided, a one-byte reservation would potentially lead to the waste of 4095 bytes, since the rest of the page cannot be used by any other process. This is an unacceptable waste of precious resources.

A subset of these system memory pools is the *Nonpaged pool*. Processes that need to ensure that some of their data never is paged out for performance or security reasons may request allocations from this pool. The memory manager asserts that allocations in this memory area never will be paged out, and always will be resident in the physical memory.

In Windows, applications can request memory from the Nonpaged pool by calling the API method `ExAllocatePoolWithTag`. The method takes three parameters, as seen in Listing 3.1.

Listing 3.1: Windows method ExAllocatePoolWithTag

```
PVOID
    ExAllocatePoolWithTag(
        IN POOL_TYPE PoolType,
        IN SIZE_T NumberOfBytes,
        IN ULONG Tag
        );
```

The `Tag` parameter is user selectable, and saved in a "reverse" little endian fashion. A call like:

```
char *pointer = ExAllocatePoolWithTag(NonPagedPool,4096,'GATa');
```

would therefore return a result in a 4096-byte allocation in the nonpaged pool with the tag "aTAG". Also note that it is possible to allocate memory blocks larger than or equal the page size, if so, a page-aligned buffer is allocated in the virtual address space [MSD08b].

Cryptographic applications are encouraged to use this feature for storage of sensitive information, including keys and plaintexts. Since the available memory in the nonpaged pool per process are small, careful consideration is needed when assessing whether or not to use the feature.

As an example, the Truecrypt device driver `truecrypt.sys` allocates memory from the NonPaged pool to ensure that no pages are written to disk. This can be observed on a system running Truecrypt using the command `pstat`, and the output from such a command can be seen in Figure 3.3. Here, column 3, 4 and 5 represent code, data and paged memory, respectively. As we can see, no memory is paged for the Truecrypt device driver.

```
Command Prompt
   Dxapi.sys F9449000    6272     384     640 Fri Aug 17 22:53:19 2001
watchdog.sys F9D14000    3712     128    8576 Wed Aug 04 08:07:32 2004
     dxg.sys BF9C3000   61312     896       0 Wed Aug 04 08:00:51 2004
  dxgthk.sys F9F75000     128       0       0 Fri Aug 17 22:53:12 2001
  vmx_fb.dll BF9D5000   75776    8064       0 Tue Oct 30 08:15:13 2007
 ndisuio.sys F7F3D000    7168     256     768 Wed Aug 04 08:03:10 2004
    hgfs.sys F7CE5000   38016   48128       0 Thu Jul 12 22:11:50 2007
  mrxdav.sys F7CB8000   26368    5504  129024 Wed Aug 04 08:00:49 2004
lgtosync.sys F9D54000   22656    1408       0 Fri Mar 10 02:47:08 2006
  ParVdm.SYS F9F12000    1408     128       0 Fri Aug 17 22:49:49 2001
vmmemctl.sys F9D5C000       0       0       0
     srv.sys F7B26000   54144    8192  237824 Mon Aug 14 12:34:39 2006
  wdmaud.sys F7A49000    8704    2048   64512 Wed Jun 14 11:00:44 2006
sysaudio.sys F7ACE000    3072     128   47360 Wed Aug 04 08:15:54 2004
    HTTP.sys F775A000   94592   26624   97664 Fri Mar 17 01:33:09 2006
truecrypt.sys F74F5000  116992   39424       0 Sun Mar 16 20:35:31 2008
 Fastfat.SYS F74A7000    8960     768  114304 Wed Aug 04 08:14:15 2004
dump_diskdump.sys F7646000       0       0       0
dump_vmscsi.sys F7726000       0       0       0
  kmixer.sys F73DC000   14592   35840  105600 Wed Jun 14 10:47:45 2006
   ntdll.dll 7C900000  503808   20480       0 Wed Aug 04 09:56:36 2004
-----------------------------------------------------------------------------
       Total          5430784  698240 4844416

C:\Program Files\Support Tools>
```

Figure 3.3: Output from `pstat` on a system running Truecrypt.

3.1.4 Address Translation

To translate a 32-bit virtual address into a physical, the memory manager needs to perform two lookups, in the *page directory* and a *page table*. This operation is pictured in Figure 3.4 and described below.

A virtual address is interpreted as three distinct components–the *page directory index*, the *page table index* and the *byte index*. This structure is shown in Figure 3.5.

To translate a virtual address, the memory manager uses the page directory index to perform a lookup in the page directory (one per process). Each executive process structure (`EPROCESS`) contains a pointer to the kernel process block (`KPROCESS`). The `KPROCESS` contains a pointer to the process page directory that together with the CR3 processor register form the physical address of that process. The value of the CR3 register is loaded from the `EPROCESS` at each context switch. The entry in the directory (the Page Directory Entry (PDE), a 32-bit structure) points to a particular page table; each process may have up to 512 page tables.

The PDE structure is isomorphic to the Page Table Entry (PTE), and can be seen in Figure 3.6. The Page Frame Number (PFN) points to the frame

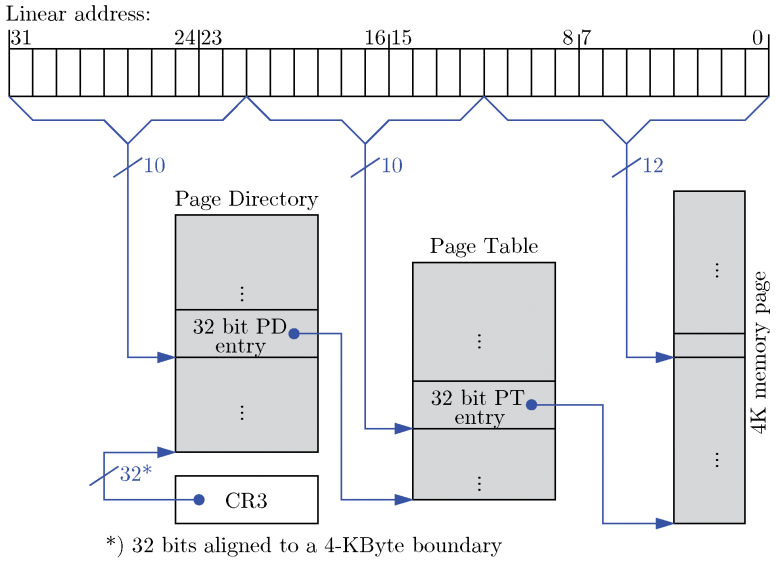


Figure 3.4: Address translation on a x86 computer using 4 KB page size and no PAE. Figure adapted from Wikipedia (see Appendix C)

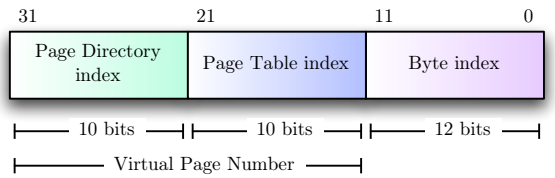


Figure 3.5: The 32-bit virtual address on x86 Windows systems.

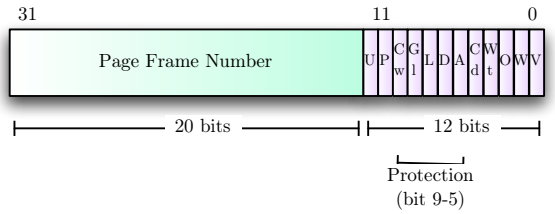


Figure 3.6: Valid x86 hardware PTE (PDE).

in physical memory where the page table can be found for PDEs, or the page PTEs. The last 12 bits describes the page and its properties.

The `EPROCESS` data structure is shown in Listing B.1 as outputted from the Windows Debugging Tools. Note the `Pcb` member pointing to the `KPROCESS` and the `VadRoot` member pointing to the VAD tree root.

Having located the page table, the memory manager uses the page table index from the virtual address to lookup the PTE. If it is valid, the PTE points to the desired page in the physical memory, and finally the desired data is found by using the byte index as a index within that page. If the page is invalid (e.g., it is paged out), the memory management fault handler locates the page and tries to make it valid by loading it (and potentially other pages) into memory.

3.2 The Physical Memory as Seen by the Digital Investigator

Applications and system code uses virtual addresses to reference its data and code, but when analyzing data from a dump, we don't know the memory management structures that the memory manager does, and cannot easily interpret the scattered pages in main memory.

Furthermore, we don't necessarily have access to registry values and other settings of the running operating system at the time of analysis. This especially a potential problem when the target system uses whole-disk encryption. Many of these parameters are needed to infer where in the physical memory the memory management structures may be located.

However, there exists searching tools that aims to combat these limitations (see Section 5.2). Some of these tools can reconstruct entire processes from memory and pagefile, resulting an an executable that may be scanned for viruses or verified against known version using fuzzy hashing like SSDeep [Kor07]. As mentioned, the memory management structures are largely undocumented by Microsoft, and therefore the results from many of these tools suffer from proof of concept nature and large amounts of false positives or negatives.

In Chapter 6 we suggest how to use our knowledge of the memory management system and its structures to perform new types of searches for cryptographic keys in physical memory dumps.

Chapter 4

Digital Forensics

In this chapter we will summarize the current paradigms related to digital memory forensics in general, and this thesis specifically. An introduction to the terminology and basic theory behind digital forensics is given, and several important forensics principles are discussed. Furthermore, a brief discussion of the different states of a system at the time of acquisition and their implications on the data available is given.

4.1 Digital Forensics Basics

To allow a brief discussion of digital forensics, a basic terminology is needed. The terminology used in this report is generally consistent with Mohay et. al [81] and Carrier and Spafford [83]. Generally the term *forensics* will in this thesis refer to *digital forensics*, as defined in Kruse and Heiser [82]:

> *"Preservation, identification, extraction, documentation, and interpretation of computer media for evidentiary and/or root cause analysis."*

Cause analysis if often performed by forming hypotheses of the course of events related to the crime. An *hypothesis* is (forensically speaking) a theory of how and in what sequence of events a digital crime or incident unfolded. To verify or refute a hypothesis one must find supporting or refuting *evidence*, and these can be physical or digital. This report focuses on the latter. In such a manner we can define *digital evidence* as [83]:

> "Digital evidence *of an* incident *is any digital data that contain* reliable *information that refutes or supports a hypothesis about the incident."*

An important aspect here is that *evidence* is not the same as *proof*. As Willassen notes on his blog on the subject [Wil08] after discussing the properties of proofs:

> *"[...] Evidence on the other hand, is an item that provides information about the sequence of events. In an investigation, there are usually many evidence items. Every single item tells its own story*

> *about the sequence of events and may confirm or refute the investigator's theory about what happened. Taken together, the evidence items may be sufficient to convince the fact finder that the investigator's theory of the sequence of events is correct. But there is no need to prove the absolute correctness of every single evidence item. Indeed, this is impossible, since proving the correctness of an empirical evidence item must necessarily have to rely on other empirical evidence items, which themselves have to be proven correct and so on ad infinitum."*

In the case of search for cryptographic keys we don't look for evidence directly; encrypted material can be metaphorically looked upon as a locked container that may or may not contain evidence, and the cryptographic key is needed to open it. Given such a "black box", it is not alway obvious how much (if any) effort that should be laid down attempting to break the container. Methods that ensure that all measures are taken to "catch" the key while it exists in digital form are thus vital to be able to decrypt encrypted data and uncover potential evidence. The worst case for an investigator is, if a strong cipher is used, to be forced to brute-force the key.

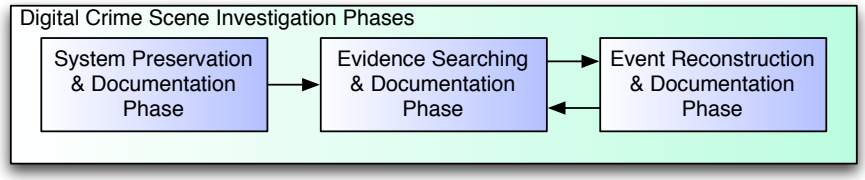


Figure 4.1: The (improved) IDIP model.

When performing a digital investigation, it is also desirable to follow a certain process model or framework for digital investigations [81, 84, 85]. Such a model allows us to relate our work to phases of a digital investigation, and think about and discuss their limitations and implications. It also promotes reproducibility and may enforce the strength of evidence in a later trial. Several such frameworks exist, notably the Integrated Digital Investigation Process model (IDIP) suggested by Carrier and Spafford [86]. This model were further improved *An Event-based Digital Forensic Investigation Framework* [83], which is the framework we have chosen to follow in this thesis.

As seen in Figure 4.1, the framework divides the digital crime scene investigation into three main phases, where the *System Preservation and Documentation* and *Evidence Search and Documentation* phases are most relevant to this thesis. The *Event Reconstruction and Documentation* phase is where the hypotheses are formed and evaluated, and this process is not considered in this thesis: We only treat crypto key discovery and interpretation, and leave the cause analysis to classical digital forensics methodology. The process model is entirely abstract, while we aim to provide a more hands-on approach for searching for cryptographic keys in memory dumps.

The system preservation phase is generally performed by documenting and preserving the crime scene as it was when first encountered. As a regular in-

vestigator would take photographs of physical objects in a regular crime scene, a digital investigator will additionally try to preserve the states of the digital artifacts found, like computers, cell phones and digital storage media. To preserve a digital crime scene, imaging tools are used to make identical copies of the components of the crime scene, while attempting to inflict as little change as possible to the overall state.

After securing the crime scene, searches for data that can be used to infer knowledge of the event chain are performed. This is the evidence search phase, and like the former phase, documentation is essential. The searches are generally performed at the preserved images from the digital crime scene, to prevent interference. The findings will be used to support or refute hypotheses, and may spawn additional searches for evidence, and even lead to new crime scenes. Both these phases relate to a few core forensic principles, namely the *Locard Principle*, the *Order of Volatility* and the *Chain of Custody*.

4.2 Digital Forensics Principles

We are concerned with finding crypto keys from crime scenes where cryptography has been or is in use. To be able to do so in a forensically sound manner, we need a procedures to guarantee that as good results as possible will be archived. We will relate our discussion to these principles that are highly relevant when evaluating the forensically soundness of such a procedure:

The Locard Principle The Locard Principle states that *Tout contact laisse des traces* – Every contact leaves a trace. This is true both in the physical and digital world, and as digital investigators we try to honor this principle by performing as little actions as possible on live digital crime scenes, and use write blockers when imaging disk drives to be certain that no unauthorized or unintended change is made to a crime scene. This is closely tied to the the idea of *atomic* data transactions, that are guaranteed to either completely occur, or have no effects. When assessing memory acquisition methods, atomicity is of great importance.

The Order of Volatility In order to gather as much data material as possible with small or no impact on the target system state, it is also wise to follow the Order of Volatility (OOV) [87]. It states that data should be collected from volatile sources first, since these are most likely to change rapidly. The idea is to preserve a digital crime scene in a particular state, so that it can be analyzed post-capture. This ground rule (together with the Locard principle) is in force when digital investigators make image copies of whole hard drives, CD-ROMS, thumb drives and all other found data for later analysis. According to the OOV, physical memory should be collected as one of the first objects at a digital crime scene, as it is highly volatile.

The Chain of Custody To create reliable evidence, it important to make sure that the evidence remains intact and in the same state as it was when it was seized. Thus, investigators use write blockers and cryptographic hash

functions[1] to verify that no change has been done to the data sources during or after investigation. To further support the Chain of Custody evidence is kept at physically secure locations, and a log is usually kept to keep track of where it has been, which individuals that have had access to it and what actions that has been performed using it since acquisition.

When considering new approaches in digital forensics, we need to look carefully and select methods that don't interfere with these three core principles.

4.2.1 Digital Forensics and Volatile Data

In a classical digital forensics investigation, the chain of custody is maintained by the fact that one usually has a original physical data source, whether it be a disk drive or a DVD-ROM. Thus, the hashes taken in the documentation phases of an investigation may later be verified against a new hash by hashing the original data source.

Volatile data is different. Because no physical representation of the data exist after powering off a computer, it is difficult to verify any data captured from it while powered on using hashes. The original data source is non-existent, and what's worse: It is quite impossible to reproduce the distinct state of it at the time of acquisition. The volatility of RAM modules are so considerable that a difference in milliseconds of the start of an acquisition procedure can influence the data in the resulting memory dump. This is in sharp contrast to less volatile media like disk drives.

4.2.2 Incident Response and the States of a Crime Scene

The state of a digital crime scene may change rapidly. For example, shutting down a computer may alter the state of the hard drive, or trigger hidden software that overwrites potential evidence. When seen in the light of the above principles, it is evident that caution should be taken to preserve the crime scene in its original state. A digital crime scene can even change state without external influence, as a result of an automated process. For example, consider a scheduled virus scan or backup procedure; both these will alter the state of the system when executed. It is also possible to alter the state of a digital crime scene from another physical location, using network access to shell or remote desktop applications. In some cases, it is therefore desirable to pull the network plug and disable wireless Network Interface Cards (NICs) when encountering a live digital crime scene.

A computer's state can be defined as the product of the states of all its software and hardware. For example, running software, present hardware, remote user interaction and scheduled tasks are all a part of the overall state of a system. The number of possible states is therefore incomprehensible, and it may be impossible to accurately evaluate a computer's state when encountered at a crime scene. Therefore, methods for generalization of states could be of help

[1] It may be interesting to note that several weaknesses has been found in the most common hash functions used today. Collisions have been found for MD5 [88] and (64-step) SHA-1 [89], and this may be used as a defense in court by claiming that a given checksum does not sufficient collision resistance and that it may have been tampered with. To preserve the chain of custody SHA-256 is used for hashing in this thesis.

to an investigator to reduce the chance of significant state alteration. Such a framework should help preserve the digital crime scene at a certain (generalized) state, so that imaging and documenting of the components that compose the crime scene may proceed.

As described earlier, the state of a computer may have great impact on the data available for analysis. When considering memory analysis, the extreme example is when a powered off computer is encountered, where simply no physical memory is available at all. Memory information may still be found in page- and hibernation files, but given whole-disk encryption, a powered off system is a "black box" case. The other extreme is the case where the computer is powered on, and cryptographic software is running. Between these extremes, there exist a countless number of states that all have different impact on the state of the physical memory. To be able to conclude anything regarding the presence of crypto keys in the different states of a computer, we need to define generic and broad states that embrace all these intermediate states, and at the same time are reproducible, reasonable and identifiable. We will return to this task in Section 6.6.

Chapter 5

Forensic Memory Acquisition and Analysis

In this chapter, we consider the volatile memory acquisition methods and frameworks presently available to forensic investigators, and assess their quality for forensic usage. Secondly, a presentation of the existing work within the field of memory analysis is given, and the selected tools and methods considered for forensics usage by applying our core forensic principles from Chapter 4.

Finally, we will summarize the present methods and discuss the need for new software and a forensically sound memory acquisition and analysis practice, and argue why new development is needed in the field of cryptographic key discovery.

The terminology within this field is somewhat confusing, probably both as a result of the maturity of it and lack of standards. Therefore, many terms like "dumping" and "acquisition" may have similar or converging meanings, depending on the context. The author has attempted to be as consistent as possible, but some overlapping of terms may occur.

5.1 Volatile Memory Acquisition

The process of seizing volatile memory on computing devices is still not a mature science, as a result of the many different OSes, versions and hardware platforms available. On Windows, there exist a myriad of strategies, few of which are forensically sound. In addition to the non-standard nature of the acquisition procedures, there exists few working frameworks or step-by-step procedures available that ensures that the principles in Section 4.2 are honored. The first individuals that encounter a "live" digital crime scene need to know what to do and perhaps more importantly, what *not* to do.

By researching how the different *states* of a digital crime scene influence the number of keys found in our investigations, we aim to provide a best practice for incident response teams, with respects to volatile collection procedures. We concentrate our research on cryptographic keys, but the procedures described for memory collection are the ones believed to have the least impact on the state of the target computer, and thus be the most forensically sound approach. They may therefore prove valuable for any forensic investigation.

Despite its maturity level, forensic dumping and examination of volatile memory is an area of great research effort, spurred by the recent activity in the field. A good summary of the existing acquisition procedures on Windows computers are provided by Nicolas Ruff [22], and we will provide a similar brief summary here. In addition, we will treat some of the methods not mentioned, and provide an assessment of their value for a digital forensics investigation.

Roughly, the existing methods on memory examination and/or dumping can be categorized into three groups: *live digital forensics*, *process memory dumping* and *full dump of physical memory*. In addition, data resources like the pagefile, secondary storage, registry entries and operating system or service pack information comes into play when analyzing the resulting data.

5.1.1 Live Digital Forensics

Performing *live analysis* or *live digital forensics* on a system can be done through the use of debugging tools, several of the tools from Sysinternals (now a part of Microsoft) and many others. By "live", we mean in the sense that the system is not halted, and the analysis is done while the system is running. By inspecting memory usage, process behavior, etc., an administrator can obtain a picture of eventual (hostile) activity. For servers and other computers demanding high uptime this may be an alternative *before* an eventual forensic investigation takes place, but one should be aware that any interaction with the system can potentially destroy evidence [90]. Some of this potential destruction can be countered by keeping detailed logs of actions during analysis, but as a forensics procedure for memory, this method is not sound because of the high volatility. Furthermore, rootkits that use hiding techniques can subvert the system into reporting false data to the investigator. A live analysis is therefore not recommended as a forensic procedure for volatile memory analysis; post-capture analysis like the ones below are advisable (if applicable).

5.1.2 Process Memory Dumping

An alternative to inspecting the live computer is to dump the address space of certain interesting processes and inspect them offline. This permits hashing of the data, and thus secures that no changes can be done to the captured data. This facilitates preservation of the Chain of Custody.

Several tools exists to dump processes from live computers, among other *pmdump* by Arne Vidström [Vid06], *Process Dumper* by Tobias Klein [Kle06] and *Userdump* by Microsoft. All these has a distinct disadvantage; that they do not pause the process while dumping, thus potentially creating a "smear" in the dumped content.

Another approach is to use the script `adplus.vbs` included in the WDT package [33]. This script is able to pause processes while dumping, creating a static image of the process at the time of capture.

However, there is one more disadvantage when considering these methods for a digital investigation: To create a complete process image, all eventually paged-out pages must be loaded into memory, potentially causing the memory manager to page other pages (potential evidence) out to disk. Furthermore, the paging operation may overwrite invalid pages that may be of interest to an investigator. The procedures does not keep the memory at its initial state, thus they interfere

with the Locard Principle. There is probably possible to implement methods that does not scramble physical memory to the same extent, but no such software was found at the time of writing. In addition, these methods needs to load the dumping software into memory, further thrashing the state of the memory. Lastly, it is a highly OS-dependent method.

5.1.3 Full Dump of Physical Memory

Full physical memory dumps follow the same paradigm that common digital investigation does: Dump first, then analyze. The whole contents of the RAM is dumped to a file, for example on external media. This method fits cleanly into the IDIP process model, by following the same steps of conduct. For forensic usage, this method is preferred to the live analysis and process dumping, as it has a firm anchorage in the Locard and OOV principles.

We will summarize the different methods for full memory dumps here, and provide an assessment of their feasibility for forensic usage.

Hardware-based Memory Acquisition

The concept of having dedicated hardware to dump memory may seem like a good idea, and there exist attempts to commercialize solutions like Tribble [19]. This method leaves no footprint in memory or on disk, and is totally OS-independent. However, hardware-based acquisition has several disadvantages that makes it unusable for the majority of digital crime scenes: It is expensive and requires pre-installation, the latter being a problem that is hard to counter. It is neither 100% foolproof, as shown by Rutkowska [91].

Direct Memory Access Through Firewire DMA

As suggested by several researchers, Firewire (IEEE 1394) may be used to dump the entire memory from a computer with the necessary hardware (e.g., a Firewire port) [24, 23, 92]. Unfortunately, the demand for such a interface is a drawback, along with the fact that it is not foolproof, and may be operating system dependent. It does neither stop the system activity, resulting in a smeared image of the physical memory.

It does however leave a minimal footprint, as no processes and software are run on the target system, thus respecting both the Locard Principle and the OOV.

DD and Other Software-based Approaches

The Unix command `dd` is a popular choice for acquisition of both memory and disk content. There even exists forensics-tailored versions of the utility to satisfy evidence and Chain of Custody demands. It is also included on many forensics toolkits and Live CDs, like Helix [Ef05]. For example, these tools (and many others) can be used together with netcat (`nc`) or `cryptcat` to stream the content of the physical memory to another computer over the network (the forensic computer has IP address `1.1.1.1`)[1]:

[1] A physical memory dump over the network is largely bottlenecked by network transfer limits, and do therefore not honor the OOV in a large degree.

```
Forensic computer: $ cryptcat -l -p 1234 | zcat > physical_memory.dmp
Target computer:   $ dd if=\\.\PhysicalMemory conv=noerror | gzip |
                   cryptcat 1.1.1.1 1234
```

Just as with process dumping, the use of a software-based approach has several drawbacks, first of all it requires software access to a special device like `/dev/mem` on Linux, `/dev/kmem` on Mac OS X (now removed) and `\Device\PhysicalMemory` on Windows (only available on systems preceding Windows 2003 SP1).

Additionally, the launch of a process will cause change in the memory on the target computer. Thirdly it is highly OS-dependent, which cannot always be trusted. Finally, it can be slow (up to several hours), and it does not normally pause the target system, potentially creating a smear in the obtained image.

Recent research attempts to counter these drawbacks by loading a specialized OS in a confined space of memory on the target computer, halt the host OS, and extract memory [93]. A proof of concept tool called BodySnatcher were also implemented, and experiments suggest large improvements over the classical `dd` method. According to the article, BodySnatcher only causes 8.4% change to the memory, compared to 46% for `dd` on a computer under normal load using 512MB of RAM. However, the approach has limitations because of the low-level nature of the software; it is highly hardware-dependent, and support for different chipsets and processors must be improved.

CrashDump Memory Dumping

By crashing a Windows system (inflicting a Blue Screen), it is possible to make the OS write all memory to disk before rebooting. This requires setting certain settings beforehand, by default Windows only performs a "minimal memory dump" when a fatal error occurs.

Inflicting such an error is easier than anticipated, either with Windows-supplied software (NotMyFault [Rus05, Mic07]) or even a keyboard shortcut [22]. However, the methods require registry editing and a reboot, and Windows writes the content by default to the boot partition, potentially overwriting evidence. These are substantial drawbacks for a forensics investigator.

Virtualization Snapshots

An ideal image of the physical memory can be archived by the use of virtualization software. In the case of VMWare Server [VMw07], a snapshot or simply suspending[2] the virtual computer will result in a full write-out of the memory of that computer to a single file at the host. This is the acquisition method used in this thesis, for a further treatment, see Section 6.4.1.

This method may be a more viable approach in the future, when virtualization software is more prevalent, at least in the server and business segment of the market.

[2] In VMware Server, a snapshot is required for the memory to be written to disk. However, in VMware Fusion, the memory is written to disk upon pushing the "suspend" button.

Hibernation Mode Memory Dumping

Many modern computer systems feature *Hibernation mode*, where the content of the memory is written to disk by the operating system before the computer goes to sleep. When the computer is waked up, the OS restores the content of the memory (and actually the processor state) from the content on disk so that the system may proceed at the point where it went into hibernation. The user usually faces a logon screen to authenticate before normal usage may commence.

Windows stores its memory content in a file called `HIBERFIL.SYS` in the root catalog of the boot partition. This file is largely proprietary, but has been reverse engineered by Suiche and Ruff [94, 95]. The content of memory is compressed within the file, and Suiche and Ruff has written a C library that allows decomposition of hibernation files, and even the construction of dd-style memory dumps from the file. When such a reconstruction has been performed, the resulting dump may be searched as a regular memory dump.

If the target system supports Hibernation, the feature may be enabled via the Control Panel. No reboot is necessary, making this method superior to crash dumping. However, the hibernation file will be written to its default destination, potentially overwriting evidence (including an eventual former hibernation file). Furthermore, if the system uses whole-disk encryption, the hibernation dump is a moot point, since the system will encrypt the file before entering hibernation mode. Whole-disk encryption systems often also feature authentication *before* any memory is loaded back in by using a pre-boot authentication screen.

Hibernation can be a good option if available, since it pauses the computer at a given state, and provides an atomic view of the memory. It is though advisable to acquire an eventual existing hibernation file first, to prevent overwriting. If a computer is encountered in its hibernation state, the hibernation file may provide invaluable information on the system state at the point in time when hibernation started, and keys may be found in the decompressed memory image.

Pulling the Plug or Power-cycling

Pulling the plug or powering off the target computer may seem like a counter-intuitive thing to do when trying to preserve the crime scene, but recent research suggests that this my be the right thing to do. By power-cycling ("coldbooting") the computer, the contents stay intact (or intact enough to correct it) because of remanence effects in DRAM modules [30].

By quickly powering the computer on again and booting from a external OS over a network or from a thumb drive, the memory may be extracted in a close to atomic fashion. It is also possible to halt the system in the BIOS until necessary equipment is present (most BIOSes feature a pause or boot menu hot-key, are fairly small and do not feature Power-On Self-Tests (POSTs) anymore). Given the presence of a "Reset"-button, it can be used to forcibly reset the computer, and halt it in the boot process while acquisition proceeds.

Halderman et al. also show that cooling down the DRAM modules with a dust remover can in an inverted position (or even using liquid nitrogen) could preserve their state for an extended period. This may permit investigators to physically remove the modules, freeze them using liquid nitrogen canisters and later extract the data when back at the lab. Another approach is to create a portable motherboard with free DRAM slots and a custom OS for dumping, so

that acquisition may be done on site. However, further research on the impact of these methods are likely needed.

This method may be a bit risky, but since powering off the computer leaves the disks and OS in an untouched state, it is honoring both the Locard and OOV principles. Compared to the methods above, it requires no pre-installation, it is atomic and not OS- or strictly hardware-dependent.

The main drawback with these methods is that they are highly hardware and BIOS-dependent, and that the risk of loosing data is imminent: If whole-disk encryption is present at the target system and the process somehow fails, the hard drive may be undecipherable as a direct result of the acquisition method. Another drawback is that the method is dependent on remanence effects that are not present in for example SRAM. In addition, certain computers (notably laptops like the MacBook Air) have soldered or physically hard-to-get RAM modules, that can be tough to extract without custom tools and previous experience.

5.1.4 Comparison of Existing Acquisition Techniques

In Figure 5.1, we have placed the above techniques in a graph comparing their feasibility as digital forensic methods, based on the previous discussion. The x-axis denotes the atomicity of the acquisition technique, that is, how well it the resulting dump matches the real physical memory. The y-axis represents the "timing" of the method; whether the method can be applied regardless of system, or if configurations or installations (both software and hardware) has to be performed on the system prior to acquisition. Please note that the methods are placed on the graph based on their estimated properties, and the placement is intended to be informal only. The coldboot technique does not really fit under either category, but we define it as a hardware technique since it ultimately depends on hardware to work.

It can quite clearly be seen that many of the methods discussed are not directly suitable for a digital forensic approach. The coldboot method could prove valuable in some cases, since it is the one with the least impact on the running system. If it fails, at least the rest of the computer is left in its original state. If a software-based method is necessary, the BodySnatcher proof of concept tool seems promising, but further research has to be performed on the feasibility and performance of the tool.

5.2 Existing Tools for Windows Memory Dump Analysis

The current set of tools available to the digital investigator when it comes to memory dumps are unfortunately not as numerous as the tools available for disk analysis. The applications are largely characterized with proof of concept nature or being in an early development stage. Furthermore, few scientific tests and experiments have been performed to evaluate their performance and accuracy. However, this is likely to change in the near future, and the time to come certainly looks promising.

To make the most of cryptographic memory forensics, tools for rebuilding the virtual address space both from process address space and pool allocations

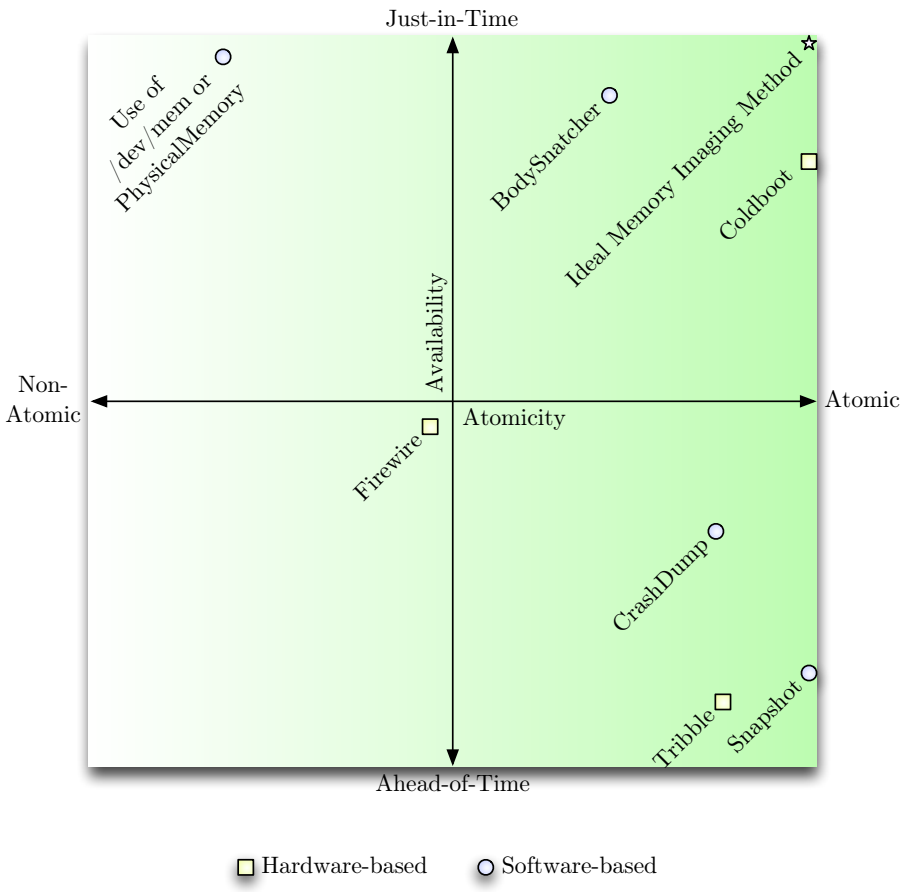


Figure 5.1: Comparison of Existing Memory Imaging Methods.

could prove useful when the data structures that are searched for are known, but potentially scattered in main memory. Like discussed earlier, this can in some cases be true when Twofish is used as cipher. Therefore, some examples of the existing analytic tools for memory dumps are warranted.

5.2.1 The PTFinder Software Tool

Andreas Schuster's PTFinder [Sch07] scans the memory dump for `EPROCESS` structures, and are able to output process graphs (or XML-structures) that even contains terminated processes. Using data from PTFinder, one can use some of his other PERL scripts (`memdump.pl`) to even manually restore executable images of processes from memory. In the related paper *Searching for Processes and Threads in Microsoft Windows Memory Dumps* [32] he also describes many of the undocumented kernel data structures like the `EPROCESS` and the `POOL_HEADER` structure. Most of these were gathered using the Windows Debugging Tools [Mic08a].

5.2.2 The PoolFinder Software Tool

In a related tool named PoolFinder [96], Schuster attempts to locate pool headers based on the `POOL_HEADER` structure. The tool outputs to a SQLLite database, which makes it possible to search for pool tags using Structured Query Language (SQL). He also includes a utility for dumping these allocations to disk. However, the tool may present large number of false hits. When testing the tool, we experienced that it frequently "lost track" of the forward and backward links in the linked list of pool headers, resulting in garbled output.

5.2.3 The Volatility Software Tool

Another approach is made by the tool suite Volatility (formerly Volatools and FATKit) [Sys07] [97], by traversing a structure known as the Virtual Address Descriptor (VAD) tree. This structure is kept in memory by the windows kernel to keep track of its virtual allocations (see Section 3.1.3). The tools are able to display a wide range of information, and even dump the process memory described by the VAD tree [98].

5.2.4 The Memparser Software Tool

Memparser [Bet05] is a direct result of the Digital Forensics Research Conference (DFRWS) 2005 challenge [DFR05], and is capable of finding processes and dumping their memory including system memory, and print loaded modules and process environment information. It may be seen as a combination of Schuster's tools, but written in C and thus much faster. However, it is only able to parse Windows 2000 dumps, cannot find terminated processes, and the upstream on the project looks quite weak.

5.2.5 The KnTTools Software Tool

Another tool that grew out of the DFRWS 2005 challenge is the now commercially available KnTTools [Inc07]. It includes KnTDD, a acquisition tool for

physical memory, and KnTList, a tool for parsing the memory dump. The latter is meant as a compliment to Schuster's work, and has a similar output.

5.2.6 Harlan Carvey's Tools

In his book Windows Forensics Analysis [33], Carvey presents tools with many of the same functions as all the above mentioned applications. His PERL scripts are free, but unfortunately they are only aimed at Windows 2000 memory dumps [Car06].

5.3 Summary

This chapter has summarized some of the current paradigm within forensic memory acquisition and analysis. As discussed, there is a great need for standardization of the fields, to support contemporary digital forensics procedures. Forming such standardization is unfortunately hard because of the wide range of hardware and software available, and the fact that physical memory's volatility makes it hard to acquire. We will return to address this issue in Section 8, where we suggest several key points to facilitate a more forensically sound approach to memory forensics.

Part II

Methodology and Practical Work

Chapter 6

Methodology

During the research of this thesis, it became clear that custom software had to be developed to facilitate search for keys in memory dumps. To the author's knowledge, no such software exist, and while the coldboot authors have promised to release their source code for public use, no timeframe for this release is given at the time of writing.

Therefore, a proof of concept tool called *Interrogate* were developed, based on a unification of the theoretic background of cryptography, memory system and volatile digital forensics. The tool is able to locate keys from several different ciphers, so that we were able to test the chances of uncovering such keys in a digital crime scene investigation.

6.1 Cryptographic Key Search Strategies

Several strategies for cryptographic key searches in memory dumps has been proposed by other researchers. As mentioned in Chapter 1, entropy, structural properties, kernel data structure and key schedules have all been used to locate crypto keys. We will discuss some of the arguments for and against each strategy here, and define existing and new search strategies we have implemented in our proof of concept tool. Among these several new approaches based on combining existing and new methods are proposed.

6.1.1 Strategy 1: Brute-Force Dictionary Attack

The ultimate naïve approach, using each sequence of bytes in memory as decryption key, is actually quite feasible. For a 2 GB memory dump, this approach has a time complexity of trying all byte offsets, e.g., around $2(1024^3) = 2147483648 = 2^{31}$. We did however not attempt this approach in our research, but a simple bash script together with OpenSSL would probably suffice.

Thus, if one knows the key size, algorithm, algorithm mode and some plaintext, this strategy is quite viable, although not very elegant. We can do better by utilizing the properties of cryptographic keys as discusses in Section 2.3.

6.1.2 Strategy 2: Compression Trial and Error

We can utilize the randomness properties discussed in Section 2.3.3 to search for random data by sampling a large enough chunk of bytes and then attempting to compress it. Compression functions rely on redundancy and inequalities in the probabilities of the symbols to form a more efficient representation of the information, so if a slight compression rate is archived, the data is probably not random. However this is not a feasible nor clever approach, since compression would put a heavy toll on performance of the search.

6.1.3 Strategy 3: Estimating Entropy

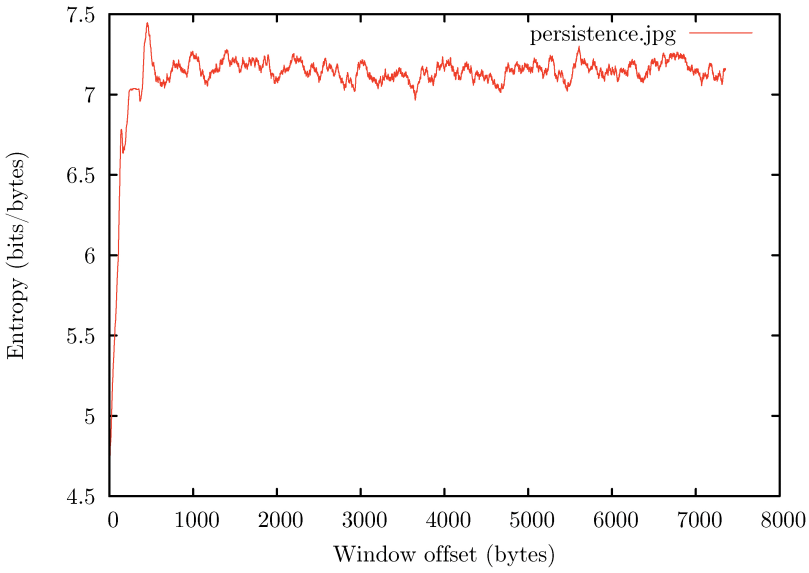
Using entropy to locate RSA keys were first proposed by Shamir and van Someren [25]. Their technique were among others to *estimate entropy* by counting unique bytes within a window corresponding to the key size. For example, a 64-byte window would match a 512-bit key. By visually inspecting regions that surpassed a heuristic threshold, they were able to pinpoint suspect areas of the memory. Using a visual confirmation method, they are able to identify 512-bit RSA keys in something they call a "lunchtime attack". Using a fairly small memory space (around 300 kB) they only get a few false positives, and no false negatives.

The algorithm can be described as follows (where the window offset is printed to the user if the counter returned is above a heuristic threshold):

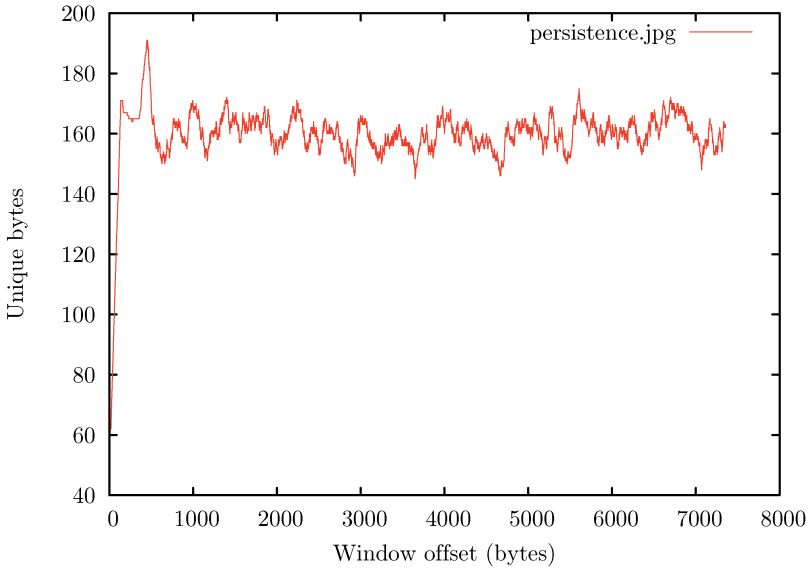
```
ENTROPY-SEARCH(DUMP, window_size, threshold)
for i ← 0 to length[DUMP] − window_size
      do BYTES[256] = NULL
         count = 0
         ▷ Iterate through window
         for j ← 0 to window_size
               do c ← DUMP[i + j]
                  ▷ Check if byte value has been counted before
                  if BYTES[c] = 0
                     then BYTES[c] ← 1
                          count ← count +1
         ▷ Return offsets where the byte count is high
         if count > threshold
            then return i
```

We used this method to estimate entropy of the image in Figure 2.3(a), and the result can be found in Figure 6.1. Here, the upper graph represent the true entropy values obtained by sliding a 256-bit window over the JPEG image. The lower graph represent the corresponding method when using the unique byte count method suggested by Shamir. By comparing the graphs, one can see that the estimate is quite good.

However, a problem with this method occurs when a too small or large window size is used. If a small window size is used, the statistical data within each window is insufficient to accurately estimate entropy, and when a large window is used, the unique byte count may approach its maximum value (that is, the alphabet size ω) regardless of the randomness of the data.



(a) *The Persistence of Memory* Entropy



(b) *The Persistence of Memory* Unique Byte Count

Figure 6.1: Entropy and estimate of entropy of a JPEG image (Figure 2.3(a)). Window size 256 bytes, values measured using the two algorithms Naive-Entropy-Search and Entropy-Search

Furthermore, the size of the memory that are available on modern computers has grown substantially since 1998 when the article was written, and our research suggests that memory images in average has numerous regions with high entropy when memory are seized from a system under normal load. Visual inspection of these would be a tedious task. Of course, location of such regions could be a great input for a future brute-force approach on the keys, by treating each offset in these regions as a key.

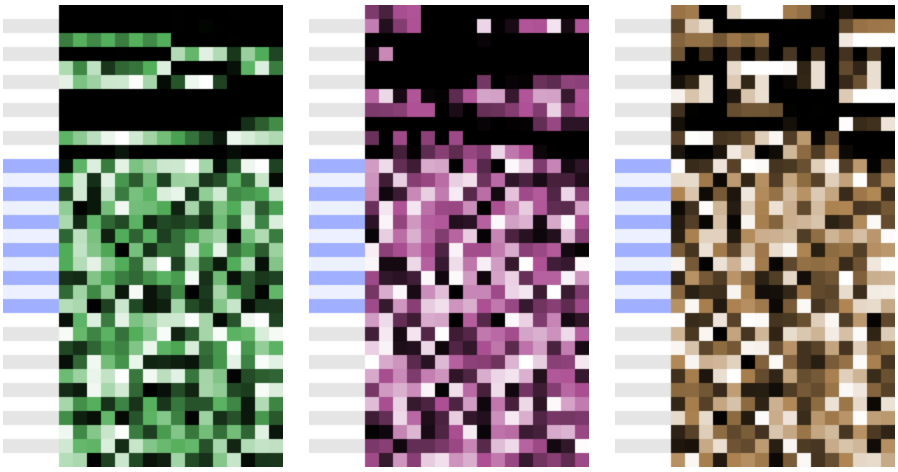
It is possible to reduce the number of false positives by performing statistical tests on the identified regions, like discussed in Section 2.3.3. One straightforward approach is to test the statistical properties of each found region, and discard the regions that are under a heuristic threshold. This could greatly reduce the number of false positives.

Unfortunately, not all keys are suitable for an entropy-based search. Symmetric keys are in general too short (in terms of bytes) to be located by this method. For example, a 256-bit (32-byte) AES key has a length that is 8 times smaller than the size of the alphabet (considering bytes with an alphabet size $\omega = 2^8$). Accurately identifying such small regions is statistically hard, since we need more data to get an precise estimate of entropy. Therefore, we also created another more naïve algorithm Naive-Entropy-Search, by calculating the *true* entropy within each window, using the original Shannon equation with he same algorithm as Entropy-Search. Although providing better accuracy, the algorithm suffers in terms of performance, and the output were largely the same. We also attempted to use the statistical methods in Section 2.3.3 to test whether regions were (pseudo) random or not, but were largely disappointed in the results. This is largely due to the size of the keys being searched for.

In Figure 6.2, we've visualized three different 128-bit AES keys found in the memory of a Windows XP system, with their expanded key schedule (a total of 176 bytes). Here, each byte value is interpreted as a each of 256 possible grey-tones, and printed out in a 16-byte wide format. We've also included the surrounding 176 bytes before and after the key schedules, and the key schedule is marked with light blue index lines. The first of these line is the 16 byte master key. Since the key schedule in most implementations will be positioned in an array or continuous memory region, measuring the entropy of this structure could be a viable approach. If we look at the figures, we can see that even if we were able to identify this region by means of entropy measurement, the key does not stand out compared to its surroundings. However, this will probably return quite a few positives, and we can do better by using the techniques from the coldboot attack.

Summarized, the drawbacks of the entropy-based search is rather substantial. The search strategy does however feature one major advantage; it is highly implementation-independent, and will, unless key obfuscation techniques are employed, locate interesting regions independent of OS, algorithm and implementation.

Entropy-based search methods combined with structural knowledge of the Twofish key schedule can be used to locate Twofish keys, as noted in Section 6.1.5.



(a) 128-bit AES key (b) 128-bit AES key (c) 128-bit AES key

Figure 6.2: Three visualized 128-bit AES keys with key schedule in memory. The whole key schedule is marked with blue lines.

6.1.4 Strategy 4: Cryptographic Key Schedule Searches

By using the method suggested in the coldboot article, AES keys can be located in memory. We implemented this method, since no implementation was available at the time of writing, and such a method was needed to answer our problem definition. The method can probably be extended to any AES implementation that pre-calculates its key schedule.

By utilizing the Serpent key schedule, we also propose a new algorithm for location of Serpent keys.

Cryptographic Key Schedule Search for AES Keys

Based on the description from Halderman et. al., we implemented our own version of the algorithm:

```
AES-Search(DUMP, key_size)
 1  for i ← 0 to length[DUMP]
 2  ▷ Treat each offset in DUMP as a key
 3        do key ← DUMP[i]
 4           ▷ Generate AES key schedule based on key size
 5           if key_size = 128
 6              then ks ← Expand-Key(key)
 7           elseif key_size = 192
 8              then ks ← Expand-Key-198(key)
 9           else ks ← Expand-Key-256(key)
10           ▷ Compare the key schedule against data at offset
11           if ks = DUMP[i]
12              then return ks
```

This straight-forward search is reported to output few false positives, and should, using a good error-correcting code against remanence effects, also be

robust against false negatives. We did not calculate the hamming distance between the key schedules like suggested above, because we are dealing with atomic memory copies created with virtualization software (see Section 6.4.1).

Cryptographic Key Schedule Search for Serpent Keys

We may utilize the Serpent key schedule structure discussed in Section 2.3.5 in the same way as we utilized the AES keys schedule, by treating each 256-bit string in the memory as a key, calculate its key schedule, and compare it to the 560 bytes at the current offset within memory.

We hereby propose a new algorithm for locating Serpent keys, based on the same procedure for locating AES keys. The search can be described with the following algorithm, where we iterate through each byte of memory, generate key schedules based on the data at the offsets and compare the results against the data found after the offsets:

```
SERPENT-SEARCH(DUMP)
for i ← 0 to length[DUMP]
▷ Treat each offset in DUMP as a key
      do key ← DUMP[i]
         ▷ Generate Serpent key schedule (560 bytes)
         ks ← SERPENT-SET-KEY(key)
         ▷ Compare the key schedule against data at offset
         if ks = DUMP[i]
            then return ks
```

The key schedule routines for Serpent are less computationally heavy than the AES key schedule generation, resulting in a slightly faster implementation.

6.1.5 Strategy 5: Structural Searches

Both Pettersson [26] and Waters and Petroni [29] suggests structural searches based on the analysis of open-source cryptographic tools and kernels. Pettersson suggests a direct search for the C data structures (e.g., `struct`) holding the keys, by interpreting each region in the memory dump as a potential structure, and testing certain heuristics like pointer addresses.

Waters and Petroni has a rather different approach, where they utilize several kernel data structures related to drive management to locate a chain of structures ultimately leading to the data structure holding the master key. They do however seem to neglect the fact that the pointers between these structures are pointers in the virtual address space, and that some sort of address translation is required. The pages containing these structures may also be paged out, further complicating the task of finding them. However, their work is hypothetical and related to their tool Volatility [Sys07], that has the ability to extract process memory from memory dumps, and their attack may therefore be feasible.

Structural search is highly implementation-specific, and in the case of Pettersson, based on several assumptions that are not always true. A decent approach could nevertheless be to utilize these methods, and try the outputs as keys. Even with a substantial magnitude of false positives, the approach would have a relatively small time complexity.

Structural Search for RSA keys

To search for RSA keys in memory, we can use these field types mentioned in Section 2.3.5 together with some basic knowledge about the keys to develop a search pattern. This search strategy has been suggested by several researchers, notably Halderman et. al. [30], Ptacek [Pta08] and Klein [27]. We add some structural checkups to provide a higher degree of accuracy an fewer false positives.

First, that according to the RSA Cryptography Standard and PKCS #8 all private keys are values of type `RSAPrivateKey` in DER encoding with the following structure:

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER, -- n
    publicExponent    INTEGER, -- e
    privateExponent   INTEGER, -- d
    prime1            INTEGER, -- p
    prime2            INTEGER, -- q
    exponent1         INTEGER, -- d mod (p-1)
    exponent2         INTEGER, -- d mod (q-1)
    coefficient       INTEGER, -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

We see that the structure starts with the field `SEQUENCE`, and then a two-byte value indicating the length of the blob. According to PKCS #8 the value of the version will always be "`0`", unless multi-prime RSA is used. We assume from now on that this value is "`0`". If we base-64 decode a private-key .PEM file, we can observe the DER encoded file in its raw (hexadecimal) format:

```
$ openssl rsa -inform PEM -outform DER -in private.key -out private.der
$ hexdump -C testdata/private.der
00000000  30 82 01 3a 02 01 00 02  41 00 b6 16 cc 12 6a 56  |0..:....A.....jV|
00000010  e1 b8 84 59 91 7d 4b 90  d2 54 02 f2 42 f6 c1 c3  |...Y.}K..T..B...|
00000020  54 96 04 c3 8a 5a 8b ee  4d de a3 0c 0f 01 50 a9  |T....Z..M.....P.|
00000030  a0 6e bb 9e bc 43 41 b8  0c 0a 88 29 68 12 2d 53  |.n...CA....)h.-S|
00000040  8e e9 03 2a d6 16 cd 01  ee 5d 02 03 01 00 01 02  |...*.....]......|
00000050  40 6e 94 b9 aa 15 5a 5e  0a 28 96 1c 7c f2 ff 28  |@n....Z^.(..|..(|
00000060  3c 4c ed c3 2d 07 cf 0f  f7 6b 3d 35 30 77 fa 68  |<L..-....k=50w.h|
00000070  de dd e6 c2 86 22 5b d2  03 e8 56 5b 6c a7 1c 7f  |.....")[...V[l...|
00000080  d4 55 4e ae e7 e3 67 a6  b7 45 bf 7a 9b 3e 80 12  |.UN...g..E.z.>..|
00000090  21 02 21 00 dd 00 0a 7e  78 bd ed 7d fa c0 cf e6  |!.!....~x..}....|
000000a0  14 d3 98 84 fe 9d e4 ce  9f 01 7c e5 a0 44 56 2f  |..........|..DV/|
000000b0  4e 8a ec ef 02 21 00 d2  ed 31 25 01 61 f8 9d 88  |N....!...1%.a...|
000000c0  25 79 e7 52 65 b8 01 84  2c 8c 05 54 47 9c 63 88  |%y.Re...,..TG.c.|
000000d0  02 55 f2 f6 3c 71 73 02  20 17 d9 e6 48 09 fd ed  |.U..<qs. ...H...|
000000e0  80 b8 2c 51 03 b2 e1 b7  47 3b 37 8d 37 23 80 04  |..,Q....G;7.7#..|
000000f0  9b bf b5 40 5b f0 ad 1b  af 02 20 48 6b 01 75 88  |...@[..... Hk.u.|
00000100  1d 00 03 ee 2b 97 c8 11  25 35 60 e7 e5 77 89 98  |....+...%5`..w..|
00000110  df 21 55 96 eb de 60 95  a4 38 fb 02 21 00 bc b8  |.!U...`..8..!...|
00000120  ca 9f 12 a7 4e be 68 d6  f7 13 48 5e 9c c0 35 4d  |....N.h...H^..5M|
00000130  02 95 74 8a 6d bf 53 ff  f7 35 04 ab 6c 71        |..t.m.S..5..lq|
0000013e
```

We observe the `SEQUENCE` value (`0x3082`) and the length (pointing at the end of the file (`0x013a + 0x4 = 0x013e`), followed by the `INTEGER` version field (`0x02`). Every `INTEGER` field is based on the following syntax (where `length` and `value` may be several bytes):

```
marker(0x02) length value
```

The `length` byte can take on a long or short form. In the short form, bit 8 of the byte has value "`0`", and bits 7-1 indicate the length of the integer (in bytes). If the length of the integer is over 127 bytes, the long form is used. Here bit 8 of the byte has value "`1`" and bits 7-1 indicate the number of additional lengths bytes. For example, if we have a 2048-bit RSA key, the first byte of the modulus `length` field would be `0x82` (`10000010` in binary), and the second and third byte `0x0101` (which is the length of the modulus, 257 bytes) [Kal93].

We can use these values and search for them in a memory dump (or any other blob of data for that matter) by performing raw string matching and some structural checkups, for example by controlling that the public exponent is either 1 or 65537 (`0x01001`):

RSA-Search($DUMP$)
1 **for** $i \leftarrow 0$ **to** *length*[$DUMP$]
2 ▷ Search for ASN.1 signature
3 **do if** $DUMP[i]$ = `0x3082`
4 **then** Parse-DER($DUMP[i]$)

The method Parse-DER implements the structural checkups, and writes the full DER-encoded keys to files on the disk.

Structural Searches for Twofish Keys

Based on the structural properties of the Twofish key schedule discussed in Section 2.3.5, we suggest several methods for locating Twofish key schedules generated from 256-bit keys. It is important to remember that even if we were to find such a key schedule, there seems to be no straight-forward way of deducing the key from the data obtained; the MDS matrix multiply and the h function is to the author's knowledge not reversible. Thus, even if we were able to identify the S keys, we cannot deduce the master key, and hence not verify whether or not the found round keys K_j matches the S vector, and vice versa.

However, the master key is not needed to decrypt content encrypted under it, the S vector and the round keys would suffice. Therefore, a simple brute-force attempt using all bytes in memory as these, could be a viable approach. This would require a modification of the original source code to permit input of the S and K vectors instead of a regular key.

We could do even better if we were able to identify certain properties of these vectors and search for them in the memory. Fortunately, the Twofish key schedule has just the properties we need to form such a search signature.

For a large number of Twofish key schedules, we clearly see that the entropy value of the S keys (Figure 6.3) take on distinct values, not a uniformly distributed high entropy value. This is congruent with the notes on the Twofish key schedule in [79, 77, 78]. We conducted a large number of experiments generating such key schedules, and found that the entropy values falls within the values in Table 6.1. The values between 3.0000 and 2.0000 are omitted for the sake of space, as these have an extremely low probability and are tested as a range in our proof of concept tool (e.g., with code like `((entropy <= 3.0000) && (entropy >= 2.0000))`). The table can be compared to Figure 6.3, where a lower number of samples were used.

Entropy Values					
4.0000	3.8750	3.7500	3.7028	3.6250	3.5778
3.5000	3.4528	3.4056	3.3750	3.3278	3.2806
3.2744	3.2500	3.2028	3.1556	3.1494	3.1250
3.0778	3.0306	3.0244	3.0000	$\longrightarrow$	2.0000

Table 6.1: Measured entropy values for the S-box keys of a 256-bit Twofish key schedule. $1 * 10^{12}$ samples were used, and the entropy value rounded off to four decimals. The arrow indicates that there exist many values in the interval $[3.0000, 2.0000]$.

Furthermore, we have conducted experiments using large number of key schedules[1], that indicate that the sub keys K_j has entropy values in the relaxed range $[6.1, 7.4]$, as seen in Figure 6.4.

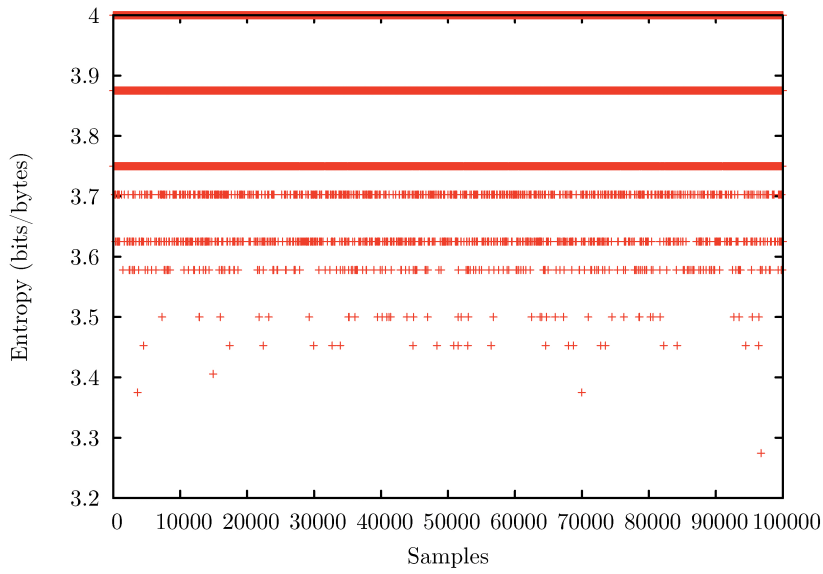


Figure 6.3: Plot of entropy from the Twofish S key vectors of 256-bit keys.

If we look at the 4 KB table, we see that it can only take on one distinct entropy value, namely the maximum possible 8 bits per byte (Figure 6.5).

As an example, a Twofish key schedule (without the large table that would not fit on this page) found in a Windows XP running Truecrypt with Twofish is presented here:

```
4aa9faa2 c00f0e9e 6cd17283 b12ac515
5ef3944a a9296b94 1a450617 66deaefc
72e068d4 0e9b7a91 e321a47e af9da9e0
7caaaf0f 98ebeac4 17538a58 2e91ec60
```

[1]The graphs in this thesis is formed using 100.000 sample key schedules although we conducted experiments using up to 1.000.000.000.000 samples when verifying these properties. The graphs from these experiments has a size that are not suitable for inclusion in a PDF document.

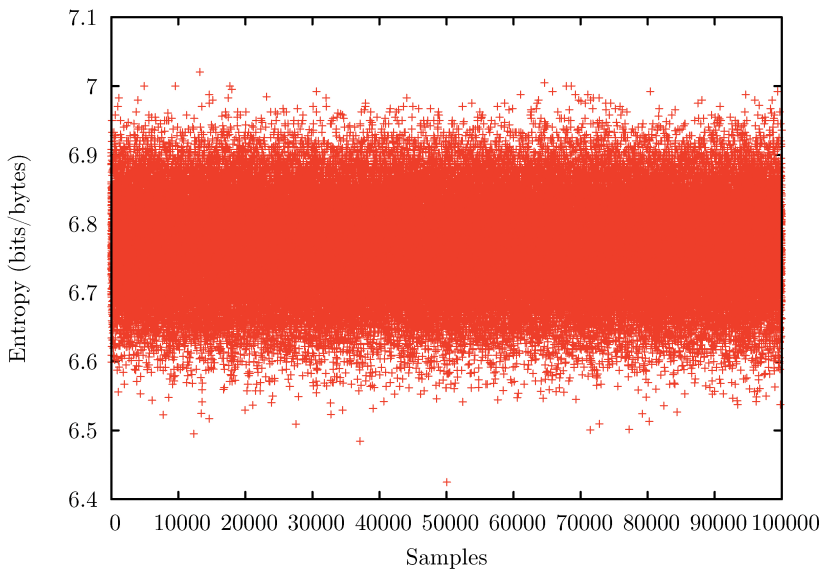


Figure 6.4: Plot of entropy from the Twofish K key vectors.

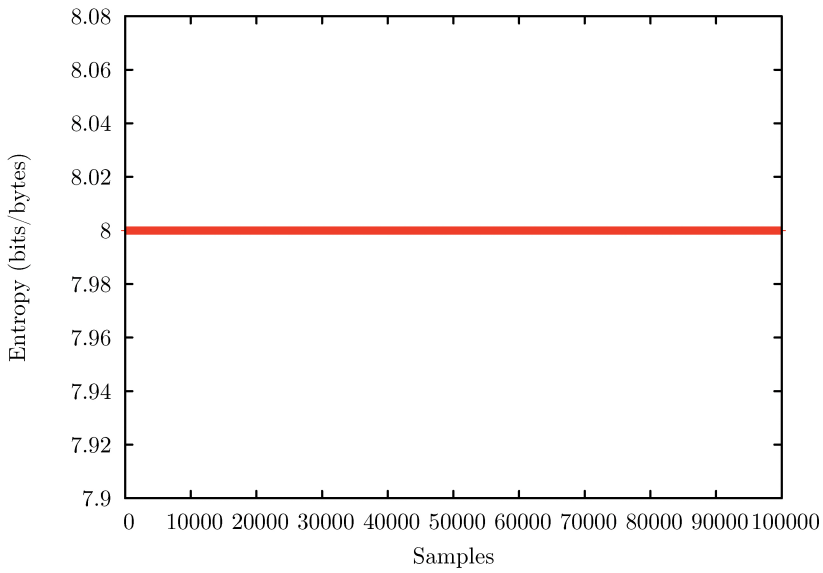


Figure 6.5: Plot of entropy from 4 KB full keying tables from Twofish.

```
dae09381 c9bf4322 1c64263f c7370026
75d29686 4b21fdbc 5f710b05 e941147d
22044248 e91468a1 042495ea 504e3746
4ffee71a 00a84644 d2870d27 55ed855a
3a748153 3b8a8150 87d4772c b7824076
296a3807 261a476b 02f2854f 645f1f42
effe9aa2 9602ff2c 21c85355 73662510
eeeeaa88 04050707 560b56e7 a93453a9
```

The first ten 16-byte keys are the round keys K_j, while the last four words are the S vector (read from right to left). Please note that the 32-bit words are printed in Litte-Endian, so the Least Significant Byte (LSB) is the leftmost byte of each word. The representation above is implementation-specific for Truecrypt [Fou08b], other applications may use other structures for managing their key material. However, adjusting search signatures to other implementations is a straight-forward task.

Listing 6.1: Truecrypt Twofish key schedule struct

```
/* Twofish key structure, taken from TrueCrypt implementation */
typedef struct {
    unsigned int l_key[40];
    unsigned int s_key[4];
    unsigned int mk_tab[4 * 256];
    unsigned int k_len;
}
twofish_tc;
```

If we look at the Truecrypt source code, it uses a C structure to store the fully expanded key schedule, as seen in Listing 6.1. Here, the `l_key` vector is the sub- and whitening keys, the `s_key` vector is the four S-box keys, the `mk_tab` is the 4 KB table and the `k_len` integer is the number $k = N/64$ as treated in Section 2.3.5, which in the case of Truecrypt always is 4 because it only uses 256-bit keys ($N = 256$). Using this information, we propose the following algorithm (where w, x, y and z are heuristic entropy thresholds):

TRUECRYPT-TWOFISH-SEARCH($DUMP$)

for $i \leftarrow 0$ **to** $length[DUMP]$
▷ Treat each offset in $DUMP$ as a key schedule struct
    do $ks \leftarrow (twofish\_tc)DUMP[i]$
        if $k\_len = 4$
            then $e\_mk \leftarrow$ ENTROPY(mk_tab) ▷ Entropy of 4 KB table
                $e\_s \leftarrow$ ENTROPY(s_key) ▷ Entropy of S-box keys
                $e\_l \leftarrow$ ENTROPY(l_key) ▷ Entropy of sub keys
                ▷ Check heuristic entropy thresholds
                if $e\_mk = 8$ and $w < e\_s < x$ and $y < e\_l < z$
                then return ks

A Less Implementation-dependent Search

To counter the drawback of only being able to search for Truecrypt keys, we propose another method of locating Twofish key schedules by means of counting runs. In addition to being highly entropic, the 4 KB table also has a quite constant number of runs (see Section 2.3.3). By evaluating a large number of

key schedules, we have set a heuristic threshold for such runs of length from one to six, as seen in Table 6.2. By counting runs in each 4 KB window of the memory dump, we can locate probable 4 KB tables. To verify these tables, we perform the same checkups as with the Truecrypt Twofish key schedule, using data structures taken from both the Truecrypt implementation and the other implementations of Twofish, including the SSH, Linux/GPG and reference implementation [Sch98]. The `structs` from these implementations can be found in Listing 6.2. This facilitates finding more than one type of key schedule data structures.

Run Length	Interval
1	[485, 520]
2	[0, 0]
3	[1, 12]
4	[0, 0]
5	[0, 0]
6	[0, 1]

Table 6.2: Intervals of measured runs of different lengths in the Twofish key schedule. Runs of 6 or more are all counted in the '6'-bin.

Listing 6.2: Twofish key schedule structures

```
/* Twofish key sructure from Linux and GPG implementations
 * Isomorphic with SSH impelentation below as far as we are concered. */
typedef struct {
    unsigned int s[4][256], w[8], k[32];
}
twofish_gpg;

/* SSH twofish key schedule */
typedef struct {
    unsigned int s[4][256];                /* Key-dependant S-Boxes */
    unsigned int k[40];                    /* Expanded key words    */
    int for_encryption;                    /* encrypt / decrypt     */
}
twofish_ssh;

/* Twofish key structure taken from Nettle */
typedef struct {
    unsigned int k[40];
    unsigned int s[4][256];
}
twofish_nettle;

/* Twofish optimized implementation */
typedef struct {
    unsigned int K[40];
    unsigned int k_len;
    unsigned int QF[4][256];
}
twofish_opt;
```

Counting runs can be optimized for sequential searches, and is thus significantly faster than measuring entropy with a large window size like 4 KB. By just keeping track of the runs that "fall out" and enter the searching window, we can reduce the runtime of our algorithm significantly. The implementation of

this algorithm can be found in the source code in Appendix A. The pseudocode for the search algorithm can be described as follows:

```
OPTIMIZED-TWOFISH-SEARCH(DUMP, window_size)
run[6] ← NULL
firstrun, lastrun, i ← 0
▷ Calculate runs for first window
RUNS(DUMP[i], run, firstrun, lastrun)
if IS-MK-TAB(run)
    then VALIDATE-TF-KS(DUMP[i])
i ← i + 1
▷ Iterate through rest of data using optimized runs method
for i to length[DUMP] − window_size
        do RUNS-OPTIMIZED(DUMP[i], run, firstrun, lastrun)
           if IS-MK-TAB(run)
              then VALIDATE-TF-KS(DUMP[i])
```

Here, the RUNS method counts the number of runs in a window, while the RUNS-OPTIMIZED method uses the numbers from the preceding runs to keep track of the runs within each sequential window. The IS-MK-TAB function simply performs a heuristic check on the number of runs, using the data from Table 6.2. Finally, the VALIDATE-TF-KS uses the structures from Listing 6.2 to perform heuristic checkups on the S-box keys and sub keys, and outputs the full key schedule if enough tests are passed.

6.2 Preprocessing: Rebuild Virtual Memory

As discussed in Section 3.2, a digital investigator may face keys that are distributed over several non-contiguous pages in memory. To counter this, we wrote a simple virtual address reconstructor. Memory reserved with an instance of a system call (e.g., `malloc` or any equivalent) are generally given contiguous virtual memory. Therefore, if we could fetch pages from the physical memory via virtual addresses and address translation, we could rebuild the virtual address space of a process, and search the reconstructed data for keys as opposed of the original memory dump. This also facilitates a significant reduction of search data.

To reconstruct the virtual address space of a process, we only need to know the location of its Page Directory Base (PDB). Using this, the reconstruction procedure greedily iterates through all virtual addresses, one page at a time, and looks them up in the process page directory and page tables. To locate the page directory base for the target process, a tool like PTFinder or Volatility can be used. This search method requires extensive knowledge about the cryptographic application, its processes and threads; specifically we need to know what thread and process that handles the cryptographic keys. For transparent cryptosystems, these threads usually operates in the process `System.exe`, which has its PDB at `0x00039000`.

This reconstruction method is not complete, as we do not fetch pages that are paged out to the pagefile. It is also prone to fetch pages that are not a part of the process, since we iterate through the whole address space of the process

(`0x00000000 - 0xffffffff`), and many addresses may not be in use. Our implementation does however permit specification of memory range to reconstruct, to facilitate selection of only interesting memory regions like the NonPaged Pool.

The reconstruction method can be used as a preprocessing step to lessen the haystack for all the above search strategies, and hence significantly improve the performance of the search.

6.3 Proof of Concept Tool: Interrogate

Implementing most of the above search strategies, a proof of concept search tool called *Interrogate* was developed. The application is able to identify and locate 128, 192 and 256-bit AES keys, 256-bit Serpent and Twofish keys and arbitrary-length RSA keys encoded with ASN.1. RSA keys are written to disk in a DER format, and it is also able to greedily reconstruct process memory given the location of that process' PDB like discussed in Section 6.2. Furthermore, the tool can indicate location of high entropy regions in memory, and therefore indicate location of interest to a forensic investigator that is looking for crypto keys.

The source code of Interrogate is presented in Appendix A under the GNU Public License (GPL). The author chose to release the code under this license to permit further development in the field, and as a consequence of the belief that open source is beneficial to both the IT security community and the public in general.

The tool makes the some assumptions about the target computer (e.g., the system that the memory dump were acquired from), namely that it is a 32-bit Little Endian system. In addition, to utilize the virtual memory reconstruction features, the target system must be Windows XP, with no PAE or /3GB switch. All the other functions, including the key search, are OS-independent in regards of target system.

Being a proof of concept tool, it is not guaranteed to locate keys, nor correctness of the keys found. It performs very few (if any) checkups on the dump image, and it is up to the user to verify checksums and maintain the integrity of the input. Nevertheless, the tool can be used as it is to forensically locate keys in real investigation cases. The input file(s) need not be a memory dump at all, any digital file could be searched for keys.

The tool is downloadable from the Interrogate SourceForge site at `http://interrogate.sourceforge.net`. Please be referred to this site for the latest release.

6.3.1 Choice of Programming Language

The tool was implemented in ANSI C, mainly because of performance considerations. C is fast, and it simplifies several of the structural searches due to the fact that most operating systems and cryptographic software are implemented using C or C++.

6.3.2 Usage

Interrogate is a command-line tool, and may be compiled and executed using a command sequence similar to this:

```
$ make
$ ./interrogate
```

Type `./interrogate -h` for further help on the different options for running the application. The tool should compile nicely[2] with GCC 3.x and 4.x on Linux and Mac OS X, and probably lower versions too (not tested). In Windows XP, the tool can be compiled with Eclipse IDE for C/C++ Developers (CDT) version 3.3.2, GCC and Make for Windows.

6.3.3 Sample Output

Interrogate is here used to search for Serpent keys in a Windows XP SP2 memory dump. One single key is found:

```
$ ./interrogate -a serpent win-xp-sp2.vmem
Interrogate  Copyright (C) 2008  Carsten Maartmann-Moe <carmaa@gmail.com>
This program comes with ABSOLUTELY NO WARRANTY; for details use '-h'.
This is free software, and you are welcome to redistribute it
under certain conditions; see bundled file licence.txt for details.

Attempting to load entire file into memory, please stand by...
Success, starting search.

--------------------------------------------------------------------------------
Found (probable) SERPENT key at offset 017ba008:

f7 7e 33 ed 83 16 e9 00 c3 81 0a d3 29 33 b3 65
a0 65 35 e6 37 c6 30 30 a5 0c 61 1b e8 b7 29 8a

Expanded key:

f7 7e 33 ed 83 16 e9 00 c3 81 0a d3 29 33 b3 65
a0 65 35 e6 37 c6 30 30 a5 0c 61 1b e8 b7 29 8a
e3 1d 37 70 a9 4a 9e 3e 2a d0 0c 82 e6 4e 9f c2
b8 4b 85 c5 99 4c d2 87 3c 99 d6 0d 73 62 71 bf
04 93 cc 86 d2 05 dc aa ea bd 8f 60 e0 32 83 ac
a6 29 1b 1c e3 73 21 f3 25 ff f1 29 82 cb 52 40
b0 9b 10 84 0f 91 e8 0e c9 70 b1 54 33 f4 1c 5b
2d 2d e5 ab 01 42 2c fc e1 a5 5d 2a 0c 23 88 aa
e3 fc 3b c2 53 15 eb ef 19 9e 3e 8b ff 13 d0 4e
0d 61 a9 f7 6f 1f 45 f4 50 e1 a6 80 0e 19 92 82
e2 11 23 2d a8 5e b8 18 c6 f0 d2 f1 4b cd e5 29
97 5f 7f 3d 74 3d 86 22 2d 40 e9 08 02 58 d4 29
99 1d c6 51 fd 05 5a db 5a 4a d9 b9 09 b0 33 f8
37 f3 d5 5e 96 a0 92 e7 6f a4 72 62 26 4b 81 fe
9a ad 80 c9 96 84 bd 88 7e 25 a7 2a da ea 84 d8
23 46 d0 96 ef 86 2b b0 37 37 00 0b be 0b fc bb
c0 62 c1 ef 6d 66 5b 46 ce 9c d1 f3 01 80 ea 97
53 d1 d7 f9 ae 25 62 6a e4 a7 13 2d 9b e1 2a 13
5f 38 f6 04 7e 2c 4d 1f 7a 08 54 ee 91 d1 73 ed
40 60 8e bd cb 7b 93 32 0c 77 76 bb 94 ac ac 81
1d bf 8d 72 38 4b 51 99 44 3e 77 8a b7 a9 8d 7f
```

[2] As mentioned in the source code, compiler optimizations using any of the `-O` switches with GCC will result in a non-functioning Twofish search.

```
60 96 14 9f 86 08 71 60 39 6b 13 a4 f4 f2 22 82
a9 e7 2a 06 15 15 b8 47 ec 81 d2 6c 12 44 3a a5
4a 31 5c 4d f9 f1 7d db 27 d0 76 a0 0c 63 6c cf
19 ae 06 8f e2 ba aa b8 c4 dd da 1d c0 3b 25 9d
52 9d 82 b4 06 d0 da f5 2d 8c 7d 39 b6 df 09 0f
24 40 d5 84 31 97 b6 c1 48 15 ad 40 0b 3c 34 fa
4a fb be a4 50 70 09 96 ca c5 31 29 16 11 bc b7
a4 f5 43 04 5d 03 b9 8f c5 a9 fd e5 f8 83 53 11
72 f4 a9 20 76 e6 f9 eb 13 94 cc 9e 07 19 21 9b
f6 92 b3 5a 3f 9d 54 8a bf b5 c4 ba e2 03 8e 88
bb 0b c1 92 b2 af c3 79 aa 76 bb 39 b2 ad 49 25
4a ad 71 f7 dd fb d3 7c d9 94 e3 c9 d4 f1 83 67
a1 e8 45 e0 bb 36 39 bb 66 e8 c4 60 95 2b 38 19
dd a2 19 f8 7e a8 be e1 47 29 85 f8 d1 d6 d1 9f

A total of 1 keys found.
Spent 1090 seconds of your day looking for the key.
```

6.4 The Testbed and Environment

To facilitate our research, we needed a testbed that had a low cost associated and was able to perform atomic memory dumps. Furthermore, we desired an as easy as possible acquisition procedure for the memory dump.

We chose to utilize *virtualization software* to generate memory dumps for testing and development. Virtualization software permits running several instances of different OSes on virtual hardware on a host computer. The hard drives of the virtual machines exist as files on the host hard drive, and the virtualization software runs a virtualization layer ("hypervisor") between the virtual hardware and the host hardware and OS.

Using virtualization has several advantages for research purposes:

- Atomicity of memory dump (see Section 5.1)
- "Snapshot" functionality provide reversibility to a "clean" state
- Easy to maintain several copies of the same system without extra hardware
- Easy to adjust hardware to the given application (for example, adjusting the amount of physical memory with a sliding bar)
- Multitasking: Several tests can be performed in parallel on different OSes on the same computer at the same time

The flexibility of a virtualized environment allows us to study the different states of a system more thoroughly and efficient, and with a far less cost associated compared to a hardware-based approach. Additionally, the acquisition procedure for physical memory is somewhat simplified compared to the real-life alternative, as discussed in Section 5.1.

6.4.1 VMware Server

VMware Server version 1.0.5 [VMw07] was chosen as the virtualization solution for this thesis. The software runs on hosts using both Linux and Windows, and is able to support a large number of *Virtual Machines* (abbreviated VM or

clients) running within the application. It is free of charge, and supports a wide range of operating systems, both on the client and host side. It also features "snapshot"-functionality, allowing preserving the state of a virtual machine for later use. Furthermore, the virtual machine can be suspended, and exported into other VMware applications like Player, Fusion or Workstation. It is also possible to download virtual machines directly from the VMWare site, allowing download-and-play OSes.

The "snapshot"-functionality of VMware server was utilized in this report to make atomic images of the physical memory at the virtual machine. To reduce the impact on the state of the target OS, all networking functions were turned off after all the necessary software had been installed. In addition, the "shared folder"-functionality that allows the client machines access to designated host folders was turned off to prevent cross-contamination between the host and the target.

VMware and Forensic Research

VMware has successfully been utilized for forensic purposes in previous research. Notably, the Virtual Security Testbed ViSe [99] were used for forensic reconstruction by Årnes et. al. [100, 101], where the virtual testbed was utilized to study the effects of computer attacks as a part of a computer crime reconstruction.

The VMware `.vmem` Format and Acquisition

Pressing the snapshot button in vmware server triggers a process that saves the full information of the target machine to the host disk. The contents of the RAM is written to a file named `VMname-SnapshotXX.vmem` within the working directory of the VM on the host. The format of the `.vmem` files is simply a plain, binary representation of the physical memory, thus providing us with an atomic view of the memory state. For acquisition, the memory file is simply hashed using SHA-256 for integrity measurement, and copied into a working directory for further analysis.

Hibernation file Acquisition

The hibernation memory was processed in two different ways, either by restarting the target machine and take a snapshot at the password prompt if no pre-boot authentication is present, or by using the hibernation file (`HIBERFIL.SYS`). To grab this file, we mounted the `.vmdk` virtual disks at the host using the `vmware-mount.pl` script included in VMware Server. Force mode is necessary when mounting since the VM filesystem is marked as "in use" when in hibernation mode:

```
$ mkdir /media/vm/
$ vmware-mount.pl Windows.vmdk 1 -o ro, force /media/vm/

And in another terminal:
$ cp /media/vm/hiberfil.sys .
```

The hibernation file was then converted to a memory dump file using Sandman [94], hashed using SHA-256 and analyzed as a memory dump using Interrogate.

6.4.2 Case Generation Procedure

The following procedure was utilized to generate data memory dumps for cryptographic key searches:

1. Two general users, Bob Internetuser `<bob.internetuser@gmail.com>` and Alice Internetuser `<alice.internetuser@gmail.com>` were created, with email addresses and accounts for messaging services if needed.
2. A fresh copy of Windows XP SP2 was was installed, and updated with all security patches.
3. A snapshot of the clean OS (disconnected from the network) was taken using VMware Server built-in snapshot function. This was stored at an external drive due to VMware Server's inability to have more than one snapshot stored at a time.
4. Software were installed, passwords and keys generated for the specific tools.
5. General usage (browsing, mail correspondence) of the OS was initiated to remove the pristine condition of the OS.
6. Another snapshot was taken. This snapshot is the basis of our analysis of each cryptographic tool.
7. One or more of the cryptographic tools were used together with general usage.
8. A snapshot was taken, the resulting `.vmem` memory image was seized, hashed using SHA-256 and analyzed according to forensic methodology and the respective case using the Interrogate tool.
9. The `.vmem` memory image was after analysis verified towards the image pre-analysis by hashing it with SHA-256 again and comparing the hashes. This ensures the integrity of the target file, and maintains the Chain of Custody.
10. The system was reverted to the snapshot taken in step 6, and then the procedure was *repeated from step 7* to facilitate the several states of each case, for example **Screensaver** or **Reboot** state, etc.
11. Finally we restored the snapshot from the external hard drive, and repeated from step 4 for each piece of software.

6.5 Cryptographic Software Classes

For the sake of clarity and simplicity in the remaining discussion, we define three main *software classes* that each of the cryptosystems tested fall into. These classes are broad and not intended for any use outside this thesis; the crypto applications are classified according to the expected presence and lifetime of their keys in memory.

6.5.1 The Whole-disk Encryption Class

Full Disk Encryption (FDE, from now on denoted "Whole-disk encryption") and other cryptosystems that need to keep their keys in memory while the system is powered on falls within this class. Whole-disk cryptosystems should feature pre-boot authentication, and of course, not load any keys into memory before *after* authentication. Good cryptographic practice also suggest that the applications should detect shutdowns, screensaver activation or hibernation, in time to wipe the keys from memory.

6.5.2 The Virtual Disk (Container) Encryption Class

This class contains crypto applications that feature standalone file containers that can be mounted as disks or read/written using any other method. The common denominator here is that these applications need to keep the keys in memory while active, but should immediately upon dismounting or closing wipe its keys. Just like in the **Whole-disk** class, cryptographic best practice suggest that keys should be wiped at shutdowns, screensaver activation or hibernation. Note that Apple's FileVault falls within both this class and the former because it only encrypts the home folder of the user.

6.5.3 The Session-based Encryption Class

These applications generates session or short-lived keys to encrypt session-based information. Some of them may indeed generate a new key for each cryptogram. Nevertheless, these applications should wipe the key from memory as soon as the session is closed or the one-time key is used. Typical cryptosystems that falls within this category includes e-mail and IM encryption.

6.6 Definition of Target Operating System States

In this section we predefine the states that are tested using VMware and Interrogate. Recall that modern computer has a finite number of states it can be in at any point in time. The real number of states is of course quite much larger than the eight states defined in this thesis, but by simplifying and merging we aim to provide states that are decipherable and clarifying to any person encountering a system where cryptography has been or is in use.

The states defined here is thus not exhaustive, but common and generic states that have impact on the chances of finding keys in volatile memory.

6.6.1 The Live State

In this state both system and cryptosystem are in a logged in state, and the cryptosystem is in use. If the cryptographic application uses virtual disks these are mounted. For **Session-based** cryptography of data like IM messages or zipped file containers, the encryption is in progress.

6.6.2 The Screensaver State

The system is in the same state as **Live**, but has been left alone until the screensaver is activated. We use the default Windows screen saver, with a one-minute delay and password protection. Since the screen saver actually may affect the state of the computer by running scheduled tasks in the background, the acquisition is performed immediately after screen saver activation. For some systems, it is impossible to guarantee that encryption still is in progress.

6.6.3 The Dismounted State

Only applicable to file-container or **Virtual Disk** cryptosystems. All disks mounted through the cryptosystem are dismounted, and the system is suspended immediately afterwards.

6.6.4 The Hibernation State

Not applicable for **Whole-disk** cryptosystems. System is set to hibernate, suspended, the `.vmdk` virtual disk mounted at the host and the hibernation file is extracted for analysis.

6.6.5 The Terminated State

Not applicable for **Whole-disk** cryptosystems. In this state, the cryptographic application is terminated and virtual machine immediately suspended. Beyond that identical to the **Live** state.

6.6.6 The Logged out State

The user is logged out, after recent activity on the system using the target cryptographic application. Note that this is not identical to a freshly booted system; the system will typically present a logon screen.

6.6.7 The Reboot State

The user has rebooted the system, but not performed any action since reboot. This may leave the system in several different sub-states: Boot prompt, cryptographic boot prompt (for example, a PGP whole-disk encryption logon screen or other pre-boot authentication mechanism) or XP logon screen.

6.6.8 The Boot State

Fresh boot. System has been powered off for an extended period of time, enough for any DRAM remanence effects to be ineffectual. VMware automatically clears the virtual RAM at a complete shutdown, so in our case the machine was restarted immediately. No user action has been performed, as in the **Reboot** state.

6.7 Cryptographic Applications

Over the next pages, we briefly present the cryptographic software tested with Interrogate. The inner details of their encryption methods will in general not be discussed, because as we will later discover, knowledge of operation and encryption modes are in general not be needed to locate keys in memory.

In addition to Windows XP applications, certain selected applications (BitLocker, FileVault and OpenSSL) that may have great impact on future digital investigations was included in the set. As both BitLocker and FileVault is bundled out of the box in Windows Vista and OS X, and integrates seamless in these, they have a greater chance of being used than their standalone rivals. OpenSSL is the common cryptographic key generator for the popular Apache web server, and prevalent in most Linux distributions. We have also chosen quantity applications over depth of search; most negative search results were not investigated furthers, even if expecting to find keys in the specific state. The reason for this is to be able to conclude broadly on the chances of finding keys in volatile memory, and not be limited to certain applications.

6.7.1 Truecrypt

Name	Truecrypt
Version	5.1a
Author	Truecrypt Foundation
Licensing	Open Source
URL	`http://www.truecrypt.org`

Truecrypt [Fou08a] is an open-source, free of charge encryption suite licensed under the GNU Public License. It features strong 256-bit encryption using either of the three AES finalists Rijndael, Serpent and Twofish, or two of them together in cascade mode. As of version 5.1, it is able to encrypt the system disk, as well as independent file containers that may be mounted as virtual drives using the Truecrypt device driver and USB or flash disks. The former falls within the **Whole-disk** software class, the latter within **Virtual Disk**.

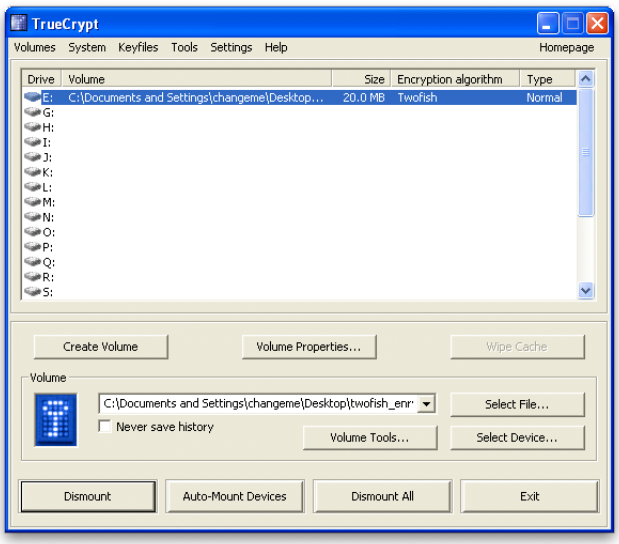


Figure 6.6: The Truecrypt main window with a Twofish-encrypted virtual disk mounted.

The whole encryption/decryption process is entirely transparent, and except from a small Truecrypt icon in the task bar of Windows, there is no visual sign of encryption. If system disk encryption is used, the system presents the user with a authentication screen pre-boot. Encrypted virtual disks exist as normal files on the host filesystem, and must be opened and mounted in the Truecrypt main window (see Figure 6.6).

In addition to its encryption feats, Truecrypt claims it provides two levels of *plausible deniability*, by the use of *hidden volumes* and the fact that no Truecrypt volume is identifiable. This feature is likely not to hide the volumes from a seasoned investigator, as there will be disk space that cannot be accounted for, unless they are sufficiently small.

6.7.2 BitLocker

Name	BitLocker
Version	N/A
Author	Microsoft
Licensing	Commercial, bundled with Windows Vista
URL	`http://www.microsoft.com/windows/products/windowsvista/features/details/bitlocker.mspx)`

BitLocker Drive Encryption is included in Windows Vista Ultimate Edition, and is able to encrypt the entire disk(s) of the system. The applications uses AES-256 in CBC mode with a custom diffuser called *Elephant* to mitigate the risk of manipulation attacks [15].

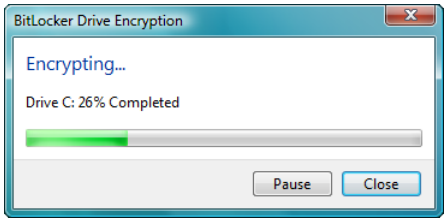


Figure 6.7: BitLocker in progress.

Vista requires per default a TPM chip to activate BitLocker, in addition to a certain partitioning scheme of the drive that is to be protected by encryption. The TPM chip ships out with newer computers, but are still not prevalent at all manufacturers. It is used for storage of the keys used for encryption, and as noted in the coldboot article [30], defaults to load the keys into RAM *before* authentication. This permits the extraction of keys from powered off machines, and makes the default BitLocker configuration insecure.

It is however also possible to use BitLocker without a TPM, by editing certain group policies. This procedure is described in Section 7.2. Since BitLocker already is known to be vulnerable in the **Boot** state when the TPM is used for key storage, BitLocker was tested without TPM support in this thesis.

6.7.3 FileVault

Name	FileVault
Version	N/A
Author	Apple Inc.
Licensing	Commercial, bundled with OS X since version 10.3
URL	`http://docs.info.apple.com/`

FileVault is a 128/256-bit AES home directory encryption tool that is included in OS X releases as of version 10.3 "Panther". It uses a key derived from the users password as master key, and encrypts and mounts the user's home directory as an image. Thus, the image is mounted and dismounted each time the user logs on or off, and no boot-time logon is necessary. There can exist several such encrypted containers at one system, one for each user that has enabled FileVault. As of the latest version of OS X, 10.5 "Leopard", FileVault uses 256-bit encryption and sparse bundles of 8 MB size instead one big image and 128 bits.

Figure 6.8: FileVault preferences pane.

FileVault has received some criticism for not encrypting the whole system drive, but this is a conscious choice from the designers, and not a flaw. This do however result in the possibility of sensitive material existing outside the container, and in the fact that FileVault does not cleanly fit into any of our cryptographic software classes.

6.7.4 DriveCrypt

Name	DriveCrypt
Version	4.61 (Demo Version)
Author	Secustar
Licensing	Commercial
URL	`http://www.secustar.com`

DriveCrypt is a commercial **Whole-disk** encryption system that boasts 256-bit AES, Blowfish, CAST and Triple DES (3DES) among its ciphers. The system is able to encrypt the boot disk of the system, featuring pre-boot authentication. It also supports standalone virtual disks that can be assigned drive letters and mounted as needed.

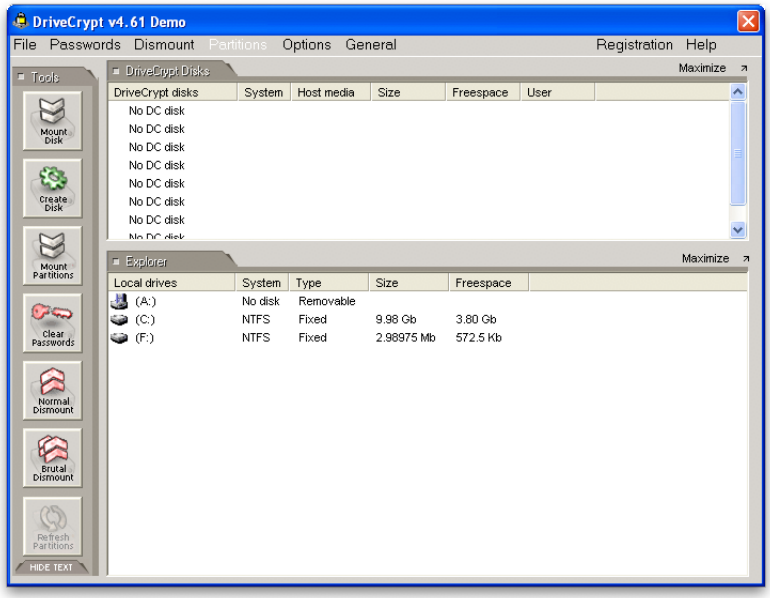


Figure 6.9: The DriveCrypt Demo main window.

The tool can encrypt CD-ROMs, DVDs and other data containers. Similarly to Truecrypt, it supports steganographic techniques to hide encrypted containers in music files or hidden partitions. In addition it supports creation of "fake" passwords that can be used to reveal "fake" content if someone is forcing the user to reveal a password. Like many of the other whole-disk encryption systems, it is completely transparent except from a small system tray icon (that can be disabled).

6.7.5 BestCrypt

Name	BestCrypt
Version	8.04.4
Author	Jetico
Licensing	Freeware
URL	`http://www.jetico.com`

BestCrypt is a freeware **Virtual Disk** container and **Whole-disk** encryption system capable of using several ciphers, among them AES, Serpent and Twofish. According to the developer Jetico, several countermeasures has been implemented in the latest release following the Coldboot article, among others crash detection, and wiping of keys at shutdown and restart [Jet08]. We tested the virtual drive encryption, which is supported by a custom BestCrypt device driver that handles on-the-fly encryption similarly to the Truecrypt driver.

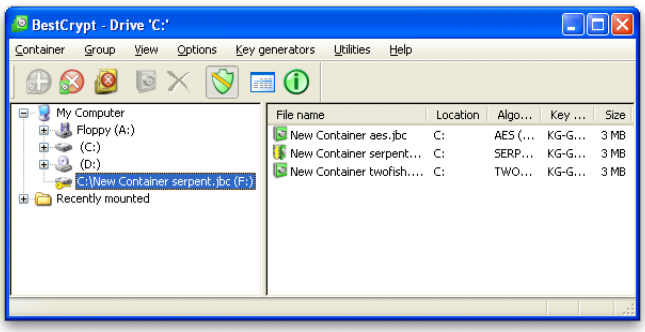


Figure 6.10: BestCrypt main window with a Serpent virtual disk mounted.

The encryption is performed using all the largest key sizes specified in the algorithm's specification, using LWR Encryption mode. BestCrypt may in addition create self-extracting archives, and the encrypted data may be visible as virtual drives, folders or NTFS partitions.

6.7.6 PGP

Name	PGP Desktop
Version	5.1a
Author	PGP Corporation
Licensing	Commercial & Open Source
URL	`http://www.pgp.com`

While originally used for e-mail encryption, Pretty Good Privacy (PGP) products has been diversified into a full set of cryptographic applications by the PGP Corporation. One of these encryption suites is called PGP Desktop, and features whole-disk encryption in addition to virtual file containers, e-mail and Instant Messaging (IM) encryption. The tool is capable of using many types of ciphers in addition to RSA, among these AES, Twofish and ElGamal.

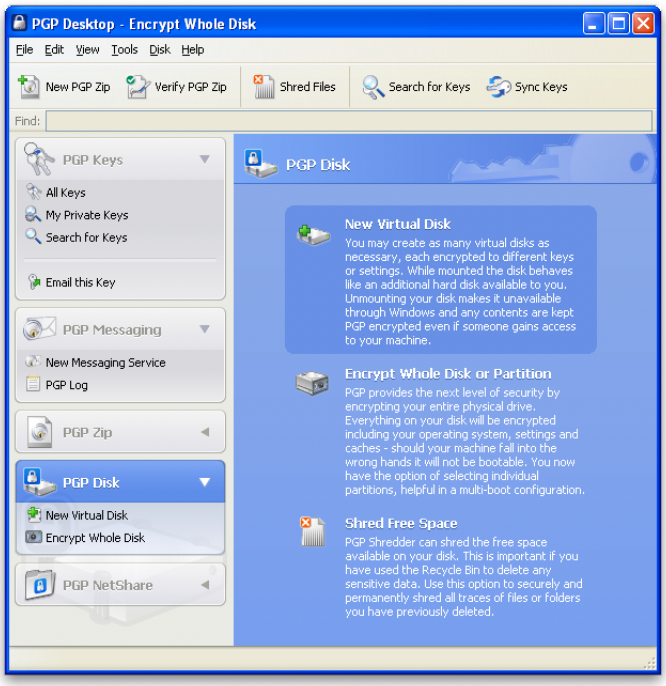


Figure 6.11: PGP Desktop Control panel.

Like the other **Whole-disk** encryption systems, when system-disk encryption is in use, it presents the user with a pre-boot authentication screen. Figure 6.11 shows the main application window; from here the user can manage his or hers encrypted devices and files.

The encryption suite acts as a proxy for e-mail and IM messages, encrypting/decrypting messages on-the-fly before handing them over to the network or requesting application. Thus, PGP Desktop falls within all our cryptographic software classes.

6.7.7 ProtectDrive

Name	ProtectDrive
Version	8.2
Author	SafeNet Inc
Licensing	Commercial
URL	`http://www.safenet-inc.com/products/data_at_rest_protection/protectdrive.asp`

ProtectDrive is a **Whole-disk** encryption system designed to encrypt system disks and Universal Serial Bus (USB)/Firewire external drives. It features several ciphers, among them AES. Like most of the whole-disk encryption systems, it features pre-boot authentication that can be token-based. It does however stand out because it uses the current Windows password as base for the encryption key. A subsequent change of Windows password will not change the encryption key however, but the new password used for authentication. The USB encryption key is derived from a user selected password, and the design of both these key derivation processes are undocumented and likely proprietary.

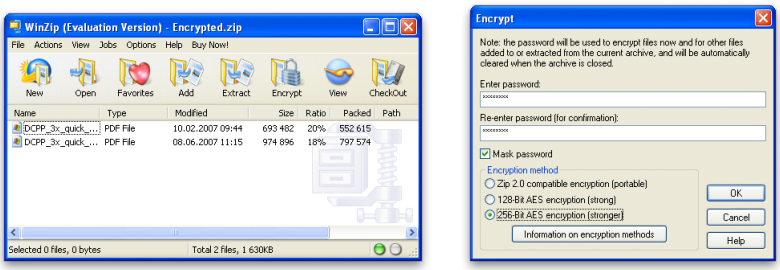
Figure 6.12: The ProtectDrive pre-boot authentication screen.

By integrating with existing Windows directory services, ProtectDrive supports ease of deployment in large organizations. It boasts features like remote management and software pushing to numerous users at the same time. In addition, token-based two-factor authentication is supported at boot via a authentication screen (see Figure 6.12).

6.7.8 WinZip Encryption

Name	WinZip
Version	11.2
Author	WinZip International LLC
Licensing	Commercial
URL	`http://www.winzip.com`

WinZip is a commercial file compression tool that features both proprietary and state of the art encryption, namely *Zip 2.0 encryption* and both 128 and 256 bit AES, respectively. Keys are derived using a password as authentication method. The files are thus stored both compressed and encrypted in a single Zip file on the hard drive. We assign this tool to the **Session-based** software class because of the short time interval where the keys presumably are in memory.



(a) WinZip main window. (b) WinZip Encryption dialog.

Figure 6.13: WinZip screenshots.

Zip 2.0 encryption is flawed and has been broken [102], while the AES implementation is based on Gladman's open source implementation [Win06, Gla06]. However, the WinZip implementation has been found to have several weaknesses [103]. The implementation has later been FIPS certified, but it is still fairly easy to brute-force the password protection[3].

[3]This is of course dependent on your resources in terms of computing power and the quality of the user-selected password.

6.7.9 WinRAR Encryption

Name	WinRAR
Version	3.71
Author	RARLAB
Licensing	FreeWare/Commercial
URL	`http://www.rarlab.com`

WinRAR is an alternative to WinZip, also featuring compression and encryption using AES-128. The solution mainly boasts the same features and formats as its commercial counterpart, and the same type of password protection authentication method. WinRAR compresses and encrypts files into single containers, and it is assigned to the **Session-based** software class, based on the same reasoning as WinZip.

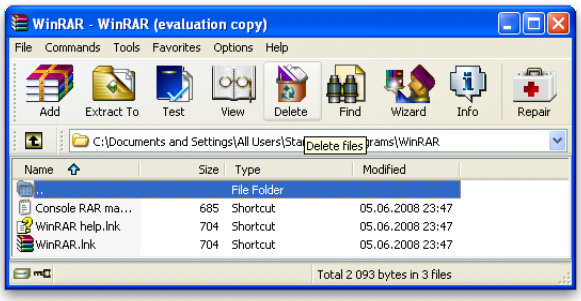


Figure 6.14: WinRAR main window.

Due to the popularity and penetration of compression software like WinZip and WinRAR, they tend to be more widely used for encryption than standalone encryption tools [104].

The encryption feature of WinRAR has received much scrutiny, just like WinZip. In a paper, Yeo and Phan describes several attacks against the feature, and summarizes by describing it as appearing to *"offer slightly better security features [than WinZip]"* [104].

6.7.10 Skype

Name	Skype
Version	3.8.0.115
Author	Skype
Licensing	Freeware
URL	`http://www.skype.com`

Skype is an internet phone communications tool that allows friends to call for free online and for low rates from computer to computer or to the Plain Old Telephone System (POTS). Skype is a **Session-based** tool.

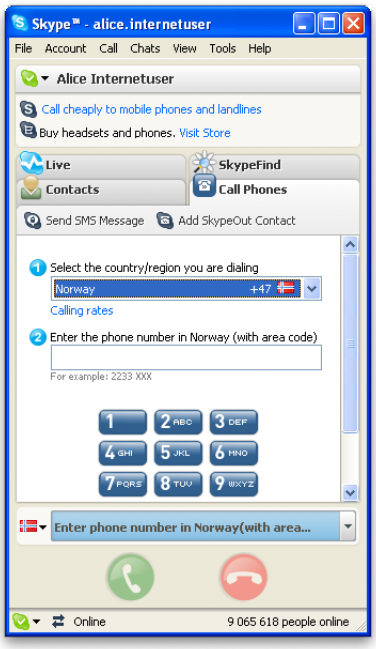


Figure 6.15: Skype main window.

The Skype protocol and its cryptographic procedures are kept in the dark by a strict closed source regime at Skype. It uses RSA and AES-256 in combination to secure its communications, resulting in a complicated proprietary protocol that only recently has been (partly) reverse engineered [105]. The protocol and the cryptographic implementation has in addition been analyzed by a Skype-hired, but external computer security expert [106]. It is no secret that Skype uses advanced methods to conceal its secrets, including obfuscation techniques and encryption of code.

6.7.11 Simp Lite MSN

Name	Simp Lite MSN
Version	2.2.11
Author	Secway
Licensing	Commercial/Freeware
URL	`http://www.secway.fr/us/products/simplite_msn`

The Simp family of encryption tools provides encryption for IM protocols like MSN and ICQ, and it is thus a **Session-based** application. It uses AES with 128 bit keys and RSA for authentication of users, and acts like a proxy for the messaging application; the chat messages are sent to a port at `localhost` where Simp Lite encrypts the message before transmitting it over the network. At the other end, Simp Lite decrypts the message before handing it over to the receiving IM client.

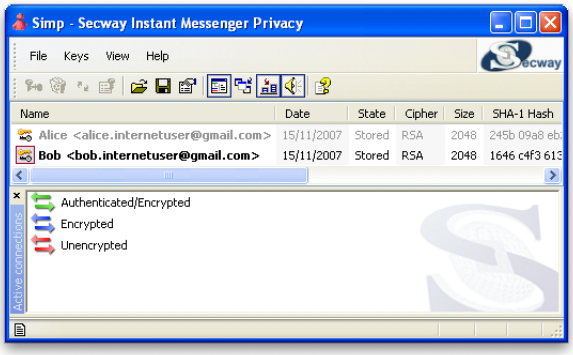


Figure 6.16: Simp Lite MSN main window.

Simp has several different modes depending on previous communications and key exchanges between the users. Upon receiving a text chat from a person using Simp software for the first time, one must approve of the other's public RSA key for future use. This key should be verified using a different and preferable secure channel, and future chats between these two entities will be automatically authenticated *and* encrypted.

The keys are stored in encrypted form in the Windows registry using an unknown algorithm and a key derived from a user-selected password.

6.7.12 OpenSSL and Apache

Name	OpenSSL
Version	0.9.8g
Author	The OpenSSL Project
Licensing	Open Source
URL	`http://www.openssl.org`
Name	Apache
Version	2.2.8
Author	The Apache Software Foundation
Licensing	Open Source
URL	`http://www.apache.org`

OpenSSL is a cryptographic suite, primarily used in cooperation with the HTTP server Apache to generate SSL certificates and perform other cryptographic duties like certificate signing. All SSL certificates consists of a private/public key pair, usually RSA keys. The SSL server uses its private key to encrypt/decrypt communications between itself and the clients, and to perform this operation it needs the private key to be resident in memory.

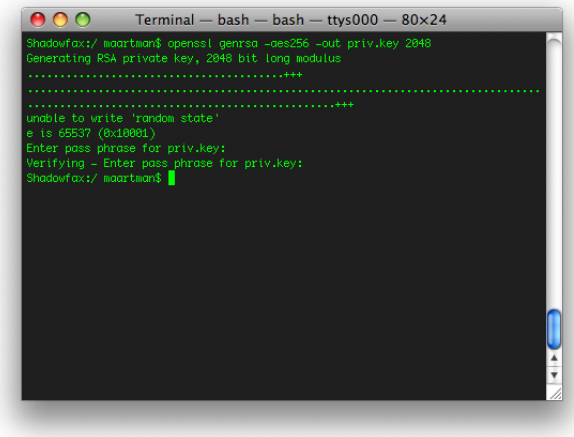


Figure 6.17: Creating a private RSA key with OpenSSL.

The keys are kept in memory at all times, mainly because of the performance degradation that would follow from decryption of the key at each HTTPS-request.

6.8 Expected Results

Generally, we expect to find encryption keys for **Whole-disk** of **Virtual Disk** cryptosystems while the disks are mounted. Implementation-specific quirks or key obfuscation techniques could thwart our search attempts, and no reverse engineering is performed if a key that are expected to be in memory is not found.

While in operation, cryptosystems are required to keep the key in some form in memory, and thus may be vulnerable to a coldboot attack. However, good cryptographic practice recommends wiping of keys when they are not in use [38]. We do therefore not expect to find keys when the cryptosystem is terminated, containers dismounted, or in any other state where there are no need for the keys to be resident in memory. We have summarized the expected results for each cryptographic software class (see Section 6.5) in Table 6.3. See explanation of this type of table in Chapter 7.

We also suspect that not all cryptographic applications pre-compute the key schedules, thereby decreasing the timeframe the full key schedule is stored in RAM. This may especially be true for the **Session-based** class of applications, and may drastically reduce the window of opportunity when the key is in memory.

State / Software Class	**Whole-disk**	**Virtual Disk**	**Session-based**
Live	Yes	Yes	Yes
Screensaver	Yes	Yes	No
Dismounted	N/A	No	N/A
Hibernation	N/A	Yes	No
Terminated	N/A	No	No
Logged out	Yes	No	No
Reboot	No	No	N/A
Boot	No	No	N/A

Table 6.3: Software classes and their expected results.

Chapter 7

Results

This chapter contains the results of the research performed during the writing of this thesis. The findings were derived using the methodology described in the preceding chapters together with the theoretical background in Chapters 2, 4, 3 and 5.

The structure of the rest of this chapter is as follows: Each application is introduced together with its findings, and a brief discussion of the results and specific search methods used. For the sake of clarity, the application specific results are discussed here rather than in Chapter 8. A simplified representation of the findings can be found in a table similar in format to the table below.

State / Cipher	**Cipher Name**
Live	Key found?
Screensaver	Key found?
Dismounted	Key found?
Hibernation	Key found?
Terminated	Key found?
Logged out	Key found?
Reboot	Key found?
Boot	Key found?

Here, each row in the table is simply answered by a "Yes"/"No" value, indicating if keys were found in that state, using the particular software with the cipher in that column. If the state was not tested or is unavailable for the cipher (for example, it is hardly recommendable to dismount a system disk used in whole-disk encryption), a value of "N/A" is inserted. All applications were tested using their default settings unless otherwise noted, on a Windows XP build `2600.xpsp_sp2_gdr.070227-2254`.

7.1 Truecrypt Results

We tested version 5.1a of Truecrypt, using both the independent system disk and virtual disk encryption and all its available ciphers. The full results can be found in Table 7.1.

	Whole-disk			**Virtual Disk**		
State / Cipher	**AES**	**Serpent**	**Twofish**	**AES**	**Serpent**	**Twofish**
Live	Yes	Yes	Yes	Yes	Yes	Yes
Screensaver	Yes	Yes	Yes	Yes	Yes	Yes
Dismounted	N/A	N/A	N/A	No	No	No
Hibernation	N/A	N/A	N/A	No	No	No
Terminated	N/A	N/A	N/A	No	No	No
Logged out	Yes	Yes	Yes	No	No	No
Reboot	No	No	No	No	No	No
Boot	No	No	No	No	No	No

Table 7.1: Truecrypt disk encryption key search results.

To safely extract Twofish keys, we cannot rely upon sequential pages in dumped memory, as explained earlier. Instead, we used the reconstruct method in Interrogate to rebuild the virtual memory of the `System.exe` process, which runs the Truecrypt device driver thread. The mounting of a virtual disk drive creates four new Truecrypt threads in `System.exe`, who are responsible for on-the-fly encryption/decryption during the time the drive is mounted (See Figure 7.1). These threads allocate memory using the previously discussed method ExAllocatePoolWithTag [Fou08b].

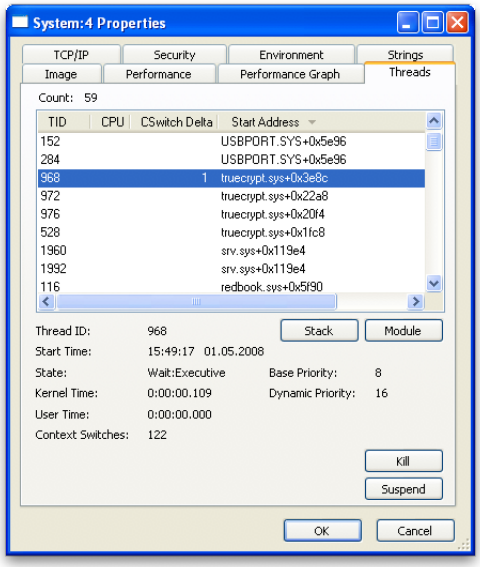


Figure 7.1: The Truecrypt driver (`truecrypt.sys`) running in the `System.exe` process. Screenshot from Sysinternals Process Explorer.

To reconstruct this part of memory, we first used PTFinder (any tool able to find the value of the PDB of a process would suffice, see Section 5.2) to find the CR3/PDB value of the `System.exe` process, and used this as part of the input to Interrogate (output from PTFinder truncated):

```
$ ./ptfinder_xpsp2 --nothreads Truecrypt-Image.vmem
No.  Type PID    TID    [...] Offset     PDB        Remarks
---- ---- ------ ------ [...] ---------- ---------- ----------------
   1 Proc      0        [...] 0x00559080 0x00039000 Idle
[...]
  33 Proc      4        [...] 0x01bcc830 0x00039000 System
$ ./interrogate -r 00039000 -a twofish Truecrypt-Image.vmem
```

In general, we had no trouble finding keys when Truecrypt was running and disks mounted. Truecrypt uses a header key to encrypt the master key in the header section of the virtual or physical drives [Fou08a], and both this and the master key were found during normal operation. We were also able to verify these keys by modifying the Truecrypt source, compiling it as a command-line tool and mounting the disks using this tool.

Truecrypt seem to be purging keys as soon as they are not needed (e.g., at the the point of dismounting of the disk), as good cryptographic practice suggest. We found no traces of the keys in the hibernation file, and in the case of the system partition encryption, this file would also be encrypted making such a search impossible.

We did however on some of the tests encounter a third Twofish key in addition to the master and the header key, also after dismounting the encrypted drive. We are however unsure of the origin of this key, and no further research were conducted on the matter. For completeness, the key's sub-, whitening- and S-box keys are presented here in the format of the Truecrypt Twofish data structure (see Listing 6.1):

```
a354793d 6e1a33f0 6ab01a83 7df52a97
12fbc877 d427152a cf9eb934 69f6e699
3ce0b947 7c5ab06d 66d41ad8 4e9f86dd
d25f6999 a3a35380 5c7cc0e2 27517ce9
9c43a538 b58d216a 49136074 4053fa28
8dd37cac 2d732874 725e993f 3f874a31
c06b1d66 b3045d42 69a78bf1 318e9035
795d6178 7692a11c cf239ae9 bafeb974
8926908b fffc400d 16a21cf1 ec65cfb2
22ad4541 01a0f21f 08fe84ab ef282332
```

7.2 BitLocker Results

Vista with BitLocker Drive Encryption was tested without the default TPM support[1]. The full results can be found in Table 7.2.

State / Cipher	**AES-128 with Elephant Diffuser**
Live	Yes
Screensaver	Yes
Dismounted	N/A
Hibernation	N/A
Terminated	N/A
Logged out	Yes
Reboot	No
Boot	No

Table 7.2: BitLocker key search results.

To be able to use BitLocker without a TPM in VMware, some adjustments had to be done. It is possible to run BitLocker without a TPM by utilizing an USB drive as authentication, by modifying a group policy in Vista before running the BitLocker initialization wizard, as shown in Figure 7.2.

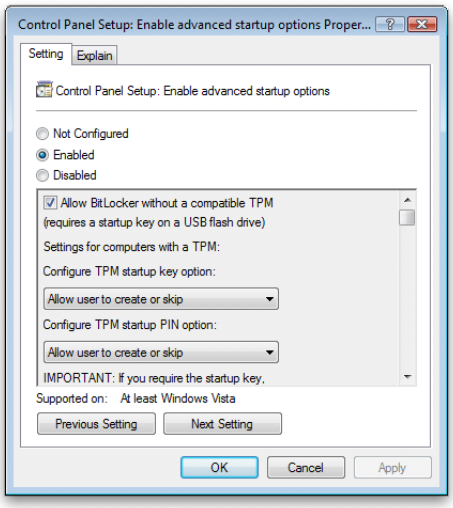


Figure 7.2: Enabling BitLocker for use without a TPM.

According to Microsoft however, BitLocker does not support running in a virtualized environment by design. Because a virtualized TPM does not exist for VMware, and USB support at boot is dismal, we had to create a virtual floppy disk[2] with the authentication info on it to be able to pass the BitLocker self-test before encryption. Furthermore, to initiate this test, we had to modify

[1] This default configuration is known to be vulnerable in *all* states, even **Boot** because the key is loaded from the TPM before authentication [30].

[2] The fact that floppy disk booting is permitted by BitLocker is undocumented by Microsoft.

the VM's `.vmx` file to accept USB devices, since the experimental support for Vista in VMware did not support USB out of the box. Because the USB support did not work in VMware Server, we were forced to use VMware Fusion for the BitLocker experiments. The following steps was performed to enable BitLocker in VMware Fusion:

1. First, to enable USB support in Vista under VMware, we added the following lines in the `.vmx` file:

   ```
   usb.present = "TRUE"
   usb.generic.autoconnect = "TRUE"
   ```

2. To be able to enter the BIOS setup by pressing F2 at boot, we extended the boot delay in the `.vmx` file (time in milliseconds):

   ```
   bios.bootDelay = "5000"
   ```

3. From the run menu, we ran `gpedit.msc`, and enabled *Local Computer Policy → Computer Configuration → Administrative Templates → Windows Components → BitLocker Drive Encryption → Control Panel Setup: Enable advanced startup options* (see Figure 7.2).

4. We created a virtual floppy-disk `floppy.flp` by creating a blank FAT-formatted `.dmg` image using OS X's Disk Utility and changing its extension to `.flp`.

5. Then the floppy was added in the VMware Fusion VM settings pane and set to be always connected. We rebooted into the boot setup menu (by pressing F2 at boot), and moved the floppy disk to last in the boot order.

6. The system was then rebooted, an USB disk drive was connected and the BitLocker wizard was started via the Control Panel.

7. We chose to "Require USB key at every startup", and saved the keys to the USB drive.

8. To copy the keys over to the floppy, we opened a command prompt with administrator privileges, and issued the command (where `A:` is the floppy mount point):

   ```
   C:\Windows\System32>cscript manage-bde-wsf -on C: -rp -sk A:
   ```

9. Finally we unplugged the USB drive, and initiated the BitLocker self-test. When the system booted into Vista, the encryption begins.

This is of course not a recommendable setup, since the permanent mounting of the floppy drive would cause the system to boot every time without authentication. The setup is for research purposes only.

BitLocker keys were found in all the expected states. In general, we found eight 128-bit keys in each dump, consisting of a total of five distinct keys with three duplicates. This confirms the results from Halderman at al. However, some of the found keys may be keys that are resident in Vista memory at

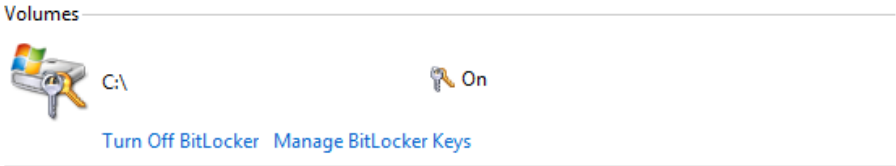


Figure 7.3: BitLocker successfully set up in VMware.

all times regardless of BitLocker, see Section 7.12. No attempt were made to confirm the keys.

In addition two keys was unexpectedly found in the **Reboot** state. They did not match any of the keys found in the **Live**, **Screensaver** and **Logged out** states, and it is therefore assumed that the keys are not used for user-mode encryption:

```
c7 9a ee ee 69 c6 df 9d 89 ca 9c 88 6b c7 41 2f
00 19 fb 91 69 df 24 0c e0 15 b8 84 8b d2 f9 ab
b7 80 99 ac de 5f bd a0 3e 4a 05 24 b5 98 fc 8f
f5 30 ea 79 2b 6f 57 d9 15 25 52 fd a0 bd ae 72
87 d4 aa 99 ac bb fd 40 b9 9e af bd 19 23 01 cf
b1 a8 20 4d 1d 13 dd 0d a4 8d 72 b0 bd ae 73 7f
75 27 f2 37 68 34 2f 3a cc b9 5d 8a 71 17 2e f5
c5 16 14 94 ad 22 3b ae 61 9b 66 24 10 8c 48 d1
21 44 2a 5e 8c 66 11 f0 ed fd 77 d4 fd 71 3f 05
99 31 41 0a 15 57 50 fa f8 aa 27 2e 05 db 18 2b
16 9c b0 61 03 cb e0 9b fb 61 c7 b5 fe ba df 9e

and

b0 a2 3f 29 86 9d 79 cc 18 60 9a c7 c1 5c 75 f0
fb 3f b3 51 7d a2 ca 9d 65 c2 50 5a a4 9e 25 aa
f2 00 1f 18 8f a2 d5 85 ea 60 85 df 4e fe a0 75
4d e0 82 37 c2 42 57 b2 28 22 d2 6d 66 dc 72 18
c3 a0 2f 04 01 e2 78 b6 29 c0 aa db 4f 1c d8 c3
4f c1 01 80 4e 23 79 36 67 e3 d3 ed 28 ff 0b 2e
79 ea 30 b4 37 c9 49 82 50 2a 9a 6f 78 d5 91 41
3a 6b b3 08 0d a2 fa 8a 5d 88 60 e5 25 5d f1 a4
f6 ca fa 37 fb 68 00 bd a6 e0 60 58 83 bd 91 fc
97 4b 4a db 6c 23 4a 66 ca c3 2a 3e 49 7e bb c2
52 a1 6f e0 3e 82 25 86 f4 41 0f b8 bd 3f b4 7a
```

These keys were *not* counted in the results for BitLocker in Table 7.2, because of the mismatch with the other keys. It is not without concern we note that these keys are present however, and the matter should be researched further. A qualified guess would be that they decrypt or are a part of the derivation of the master key from the floppy credentials, but since the format of this is unknown we were unable to conclude on this matter.

7.3 FileVault Results

FileVault was tested at a OS X 10.4 "Tiger" instance, and the key located using Interrogate. Please note that Tiger was run on a physical Macbook with 1 GB of RAM, and not within VMware. Acquisition was performed using the coldboot technique, utilizing SYSLINUX on a USB drive together with `msramdmp`. The full results can be found in Table 7.3.

State / Cipher	**AES-128**
Live	Yes
Screensaver	Yes
Dismounted	N/A
Hibernation	N/A
Terminated	N/A
Logged out	No
Reboot	No
Boot	No

Table 7.3: FileVault key search results. Note that hibernation mode does not exist on Apple OS X.

As expected, we found the AES key present when in the **Live** and **Screensaver** states. We were also able to verify the key by using an administrator account and the `hdiutil` tool supplied with OS X:

```
administrator$ su root
Password:
root# cp /Users/aliceinternetuser/aliceinternetuser.sparseimage .
root# hdiutil attach -debug aliceinternetuser.sparseimage >& out
root# grep encryption-key out
4 : <CFString 0xa7a829f4 [0xa080b1c0]>{contents = "encryption-key"} =
<CFData 0x331670 [0xa080b1c0]>{length = 16, capacity = 16, bytes =
0xa470ea89c3d4d1dca0bcbb672021752e}
```

The `hdiutil` tool mounts the encrypted sparse image, and if in debug mode, prints the key in plain form to `stderr`. By grepping for a known string ("encryption-key"), we are able to verify the key found in memory (see Figure 7.4). Mounting the encrypted sparse image at the host was not attempted, although this would have been possible by the use of VileFault [107].

FileVault seems to practice immediate wiping of keys from memory when they are not needed. Although a heavily debated feature, the fact that only the home directory is encrypted helps FileVault protect against memory analysis since it has fewer vulnerable states than whole-disk encryption systems.

Figure 7.4: Screenshot from the process of revealing the FileVault key.

It should also be noted that we found copies of the user's master key password in the memory in several of the states. This would make cryptographic key searches a moot point, since this password effectively could unlock the user's key chain and retrieve the FileVault key.

7.4 DriveCrypt Results

We tested the DriveCrypt's virtual disk encryption with AES-256 using a free demo version of the tool. Secustar claims the demo version "provides no security", but as we were unable to locate any vendor information addressing the coldboot attacks, the software were tested nevertheless. A summary of our results can be found in Table 7.4.

State / Cipher	**AES-256**
Live	Yes
Screensaver	Yes
Dismounted	Yes
Hibernation	Yes
Terminated	No
Logged out	No
Reboot	No
Boot	No

Table 7.4: DriveCrypt key search results.

Generally, we found DriveCrypt to be vulnerable to the cryptographic key searches and memory analysis attacks; three distinct AES keys were found when the cryptosystem was in the **Live**, **Screensaver** and **Hibernation** states. We also discovered an extra duplicate Twofish key of unknown origin when the system was live and the virtual disk was dismounted:

```
4c 5d 19 7a d7 4b dd b2 45 d1 2d 0c 6a ea 1d ae
90 a3 c3 b0 4d cb cc cc ea c9 00 2b 8e 06 d9 89
22 68 be 63 f5 23 63 d1 b0 f2 4e dd da 18 53 73
c7 0e 2e 3f 8a c5 e2 f3 60 0c e2 d8 ee 0a 3b 51
47 8a 6f 4b b2 a9 0c 9a 02 5b 42 47 d8 43 11 34
a6 14 ac 27 2c d1 4e d4 4c dd ac 0c a2 d7 97 5d
4d 02 23 71 ff ab 2f eb fd f0 6d ac 25 b3 7c 98
99 79 bc 61 b5 a8 f2 b5 f9 75 5e b9 5b a2 c9 e4
7f df 4a 48 80 74 65 a3 7d 84 08 0f 58 37 74 97
f3 e3 2e e9 46 4b dc 5c bf 3e 82 e5 e4 9c 4b 01
b1 6c 36 21 31 18 53 82 4c 9c 5b 8d 14 ab 2f 1a
09 81 3b 4b 4f ca e7 17 f0 f4 65 f2 14 68 2e f3
d4 5d 3b db e5 45 68 59 a9 d9 33 d4 bd 72 1c ce
73 c1 a7 c0 3c 0b 40 d7 cc ff 25 25 d8 97 0b d6
1c 76 cd ba f9 33 a5 e3 50 ea 96 37 ed 98 8a f9
```

Because DriveCrypt is closed source and we only had access to a demo version, the key was not investigated further. We were also unable to verify the keys found because of the same reasons.

7.5 BestCrypt Results

BestCrypt was tested using its virtual drive encrypting capabilities, with the AES, Serpent and Twofish ciphers. It generally seems to manage its keys well, by wiping them from memory at dismount and/or shutdown. We did on certain instances find keys probably originating from cryptographic algorithm self-tests like the key `00 01 ... 1e 1f`. Apart from that, no unexpected keys were encountered.

State / Cipher	**AES-256**	**Serpent**	**Twofish**
Live	Yes	Yes	No
Screensaver	Yes	Yes	No
Dismounted	No	No	No
Hibernation	Yes	Yes	No
Terminated	No	No	No
Logged out	No	No	No
Reboot	No	No	No
Boot	No	No	No

Table 7.5: BestCrypt key search results.

We found the countermeasures (see Section 6.7.5) to work as specified by Jetico, but unexpectedly we were not able to locate any Twofish keys. We are currently unsure of the reason for this, but suspects that the implementation in BestCrypt is slightly different from the ones we implemented in Interrogate. We hold the probability of that key obfuscation techniques are in use as low, since they are not present for AES and Serpent keys.

As Jetico mentions in its advisor on the coldboot article, BestCrypt is vulnerable when the virtual disks are mounted and on hibernation, where the keys are written to the hibernation file. Being vulnerable in the **Hibernation** state is not recommendable, since the hibernation file may exist on the boot partition of the disk drive for an extended period of time. From a digital forensics point of view, it does permit investigators to use the Hibernation acquisition method, as described in Section 5.1.3.

7.6 PGP Results

We tested PGP Desktop with virtual and full disk encryption with pre-boot authentication and e-mail messaging using both AES and Twofish. AES-256 was utilized as the cipher for the full disk encryption, RSA for the email encryption and Twofish for the virtual disks. A summary of our results is shown in Table 7.6.

	Whole-disk	**Virtual Disk**	**Session-based**
State / Cipher	**AES-256**	**Twofish**	**RSA**
Live	Yes	Yes	No
Screensaver	Yes	Yes	No
Dismounted	N/A	No	N/A
Hibernation	N/A	Yes	No
Terminated	N/A	No	No
Logged out	Yes	No	No
Reboot	Yes	No	N/A
Boot	No	No	N/A

Table 7.6: PGP key search results.

PGP whole-disk encryption is vulnerable to the same attacks as the other similar systems we have tested. However, PGP also fails to wipe its keys from memory at a reboot, resulting in keys in memory at the pre-boot authentication screen. We were repeatedly able to locate these keys, regardless of cipher. We consider this to be bad cryptographic practice, and in effect, it enables attackers that encounters PGP desktops in pre-boot authentication mode to find the keys, given that a restart has been performed. The attack is clearly an opportunistic attack, and it also depends on the boot manager to not wipe the memory at boot. Still it is a weakness that should be addressed in upcoming releases, and the author has notified PGP Corporation of this.

We were not able to locate any RSA keys in the memory dumps; this may be due to the implementation specific details of PGP, or wiping of keys. We were able to locate Twofish keys using Interrogate in the expected states when the tool used virtual disk encryption.

7.7 ProtectDrive Results

We tested both whole-disk and USB drive encryption, using AES-256 as our cipher choice. The full results can be found in Table 7.7.

	System Drive	**USB Drive**
State / Cipher	**AES-256**	**AES-256**
Live	Yes	Yes
Screensaver	Yes	No
Dismounted	N/A	No
Hibernation	N/A	No
Terminated	N/A	Yes
Logged out	Yes	Yes
Reboot	Yes	Yes
Boot	No	No

Table 7.7: ProtectDrive key search results.

ProtectDrive distinguishes itself from the other cryptosystems tested in a negative way due to the overwhelming number of keys present in memory at all times. Even freshly installed copies, not yet utilized for encryption, contained between 11 and 20 keys, all duplicates of three or more keys. These do not disappear from physical memory until the system is shut down *and* restarted (not power-cycled) or ProtectDrive is *uninstalled* using the Control Panel in Windows.

Being unable to test these keys, we attempted several re-installations of the tool to see what changed, using a new Windows password at each installation. Generally we found three distinct keys with duplicates in memory:

1. `00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f`
   `10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f`

   This key is obviously not random, and probably not used for encryption, but rather as a self-check for the encryption algorithm.

2. `a5 50 84 b1 31 5d 33 cc a1 c5 f3 33 f6 e7 d6 b7`
   `3d d7 b8 60 07 8e d5 ab 2d aa 3d 28 aa 05 3e db`

   The second key was found as many duplicates on each re-installation, and are, based on a qualified guess, believed to be some kind of application master key. The usage or need for such a key is however unclear.

3. A random-looking key that changed for each re-installation or USB drive encrypted. A qualified guess would probably suggest that this is the main encryption key.

When testing USB encryption, a fourth key was also present, presumably the USB master key. This key is wiped from memory when the USB key is removed using the "Safely remove hardware" Windows feature. If the USB drive is pulled out without using this feature, the key wiping is still performed, and the USB key is not present in memory at the time of acquisition.

The USB drive encryption dismounts and wipes keys when the screensaver activates, and our research suggests that it does so successfully. We also attempted to forcefully terminate the ProtectDrive process; this resulted in finding all the keys found while the cryptosystem was in the **Live** state. Finally, we logged the user using the USB device out without ejecting it, and surprisingly the key was *not* wiped. If the system is rebooted with the USB drive inserted, this key is also present at the pre-boot authentication screen.

Keys were also present *after* decryption of the system drive *and* reboot. As mentioned, a full uninstall of the application had to be performed to get rid of the keys. If the computer was simply rebooted, all keys were present in memory at the boot prompt. Like PGP, this is a security flaw that is worrying. However, the application does not seem to load any keys pre-authentication, and it is therefore not vulnerable in the **Boot** state. SafeNet Inc. has been notified of all of the above weaknesses.

Although the usage of the keys found is pure qualified guesswork, it is worrying that they are present in states such as **Reboot**. It is also disturbing that the keys are present in memory when they presumably are not needed at all, like when no encryption is employed. The share number of keys in memory is in addition troublesome from a security perspective, and it is unclear why and how so many instances of the keys are present. Compared to the other whole-disk encryption systems, ProtectDrive stands out as having a rather careless key management practice.

7.8 Results from WinZip and WinRAR Encryption

WinZip and WinRAR boast both compression in terms of size and security by encryption. We tested WinZip and WinRAR by creating a large encrypted archive in a virtual machine, and took snapshots both during the encryption process and afterwards. The full results can be found in Table 7.8.

	WinZip		**Winrar**
State / Cipher	**AES-128**	**AES-256**	**AES-128**
Live	No	No	No
Screensaver	No	No	No
Dismounted	N/A	N/A	N/A
Hibernation	No	No	No
Terminated	No	No	No
Logged out	No	No	No
Reboot	N/A	N/A	N/A
Boot	N/A	N/A	N/A

Table 7.8: WinZip and WinRAR key search results.

Although the Gladman implementation [Gla06] is compatible with the key schedule search method for AES keys, we were largely disappointed to not find any keys for neither WinZip or WinRAR. The results for these two applications are therefore largely inconclusive; since we did not find any keys in the expected state (e.g., **Live**) we cannot deduce if the keys are properly wiped in any of the other states.

7.9 Skype Results

We tested Skype by text and voice chatting between two instances of Windows, one Vista and one XP SP2 system on VMware. Both these systems were suspended in the separate states, and both systems analyzed using Interrogate. The results can be found in Table 7.9.

State / Cipher	**AES-256**	**RSA**
Live	No	No
Screensaver	No	No
Dismounted	N/A	N/A
Hibernation	No	No
Terminated	No	No
Logged out	No	No
Reboot	N/A	N/A
Boot	N/A	N/A

Table 7.9: Skype key search results.

Interrogate was unable to locate any Skype AES and RSA keys. We suspect that this is due to key obfuscation methods, based on the history of the Skype protocol [105]. However, it is important to note that these obfuscation methods does not contribute to the security of the application, as it could be reverse engineered and the search algorithm adjusted to the findings. Thus, the obfuscation techniques does nothing more than halt a potential attacker; it does not provide any additional cryptographic strength.

Skype is believed to use correctly implemented AES encryption. We are nevertheless not able to conclude how efficiently it operates in terms of key management and wiping.

7.10 Simp Lite MSN Results

We tested Simp Lite by running two clients on a Vista and XP SP2 system, and exchanged keys between the clients. Afterwards, some basic messages were sent back and forth between the clients, suspending the machines at different times during conversation. The resulting memory images were subsequently analyzed using Interrogate.

State / Cipher	**AES-128**	**RSA**
Live	No	No
Screensaver	No	No
Dismounted	N/A	N/A
Hibernation	No	No
Terminated	No	No
Logged out	No	No
Reboot	N/A	N/A
Boot	N/A	N/A

Table 7.10: Simp Lite key search results.

The results from the cryptographic key searches on Simp Lite memory dumps can be found in Table 7.10. No AES or RSA keys were found during operation, and consequently we were unable to locate any keys in the remaining states as well. Because of Simp Lite's commercial license and closed source, we were unable to research the matter further. However, we suspect that key obfuscation techniques are in use, or that the implementation is slightly different and therefore cannot be found using key schedule search. We have been unable to locate any information whether or not Simp Lite is implemented according to FIPS standards, and as with Skype we are unable to conclude on Simp Lite's key management procedures.

7.11 OpenSSL and Apache Results

We tested a generic Apache web server together with OpenSSL on Ubuntu 7.10 by generating a couple of key pairs, and finally installing one of them in the server. An automates script were used to subsequently issue HTTPS requests to the server over a closed network using the VMware virtual network bridge. A snapshot of the server was then taken and analyzed with Interrogate; the full results can be found in Table 7.11.

State / Cipher	**RSA**
Live	Yes
Screensaver	Yes
Dismounted	N/A
Hibernation	N/A
Terminated	N/A
Logged out	Yes
Reboot	No
Boot	No

Table 7.11: OpenSSL and Apache key search results.

Although we found RSA keys, only one of them matched the key assigned to the server. The other generated keys that were not used by OpenSSL were not present in memory, suggesting that OpenSSL wipes keys after generation. Since the server ran as root and as a daemon, it continued running after logging out of X. Keys were also found when no HTTPS requests were issued, suggesting that the server keeps its private key in memory at all times, regardless of incoming traffic. Although we found the RSA key, it seems like Apache and OpenSSL were built with security in mind, and for the SSL server to function it is clearly necessary to keep the key in memory at all times.

This opens up for an interesting attack for an adversary that has managed to get root access to a server running OpenSSL: By searching memory for private keys, he is able to extract keys that otherwise are encrypted (on disk). When the keys are extracted he may reposition to another machine, network or router, and using WireShark he may decrypt all subsequent *and* previous SSL traffic given a normal SSL ciphersuite [18].

7.12 Other Keys Found During Research

While searching for Simp Lite keys, three 128 bit AES keys were found to be present in memory of Windows Vista at all times. These differed on each installation of Vista, but remained the same for each instance at different points in time. It is not clear why these keys are present at out-of-the box Windows Vista Business and Ultimate Edition systems, but we reproduce three of them here for completeness:

1. `0d 89 e5 5b ea 9f b7 d9 da fb bd 09 3e f2 fd 31`
2. `b6 e4 48 2d c1 bd 00 89 3f 02 f9 dd 5d a5 10 22`
3. `fc 2a 2f e2 40 fd f9 33 36 4d cc f6 3c 95 04 46`

The self-test vector key `00 01 ... 1e 1f` were also located in several instances of Windows XP without any cryptographic tools installed.

Chapter 8

Discussion

In the following sections, we will discuss the results from Chapter 7 and evaluate the proof of concept tool Interrogate, its performance and cryptographic key search algorithms. First, suggestions for new features and improvements both in the existing algorithms and search strategies are proposed, and secondly we suggest how the overall performance of the tool may be significantly improved.

Third, we provide an in-depth discussion of the research results, and assess how these results can contribute to new live response methods for law enforcement and other forensics groups. Finally, we will discuss the possibility for a forensically sound approach to memory forensics, with emphasis on recovering cryptographic keys.

8.1 Evaluation of Proof of Concept Tool Interrogate

Although a proof of concept tool, Interrogate performed as better than expected, notably when looking for symmetric keys. RSA keys proved difficult to locate using the techniques proposed by other researches, and seems highly implementation and OS dependent.

8.1.1 Performance Evaluation

Interrogate is mostly not optimized in terms of time or space, in fact, most of the cryptographic key search strategies could indeed be significantly faster. We've demonstrated one such optimized approach in Optimized-Twofish-Search, on our test computer the average of these searches perform much faster in an accuracy/speed trade-off (256 MB in just over 10 seconds).

In addition to speed and accuracy, memory consumption and handling comes into play when running Interrogate at systems with lower specifications.

Speed

An overview of the average running times at two test computers, an Ubuntu 7.10 computer with Intel Core 2 Duo 6400 @ 2.13 GHz and 3.2 GB of RAM and a Macbook running OS X 10.5 with an Intel Core Duo @ 2 GHz and 2

Algorithm / Memory Dump Size	**256 MB**	**512 MB**	**1 GB**
AES-Search (256 bits)	18	57	114
Serpent-Search	19	40	84
Optimized-Twofish-Search	0.2	0.5	1
Truecrypt-Twofish-Search	0.11	0.22	0.44
RSA-Search	0.01	0.05	0.1
Entropy-Search	9	20	39
Naive-Entropy-Search	28	61	123
AES-Search (Virt. Mem. Reconstr.)	1.06	1.1	1.25

Table 8.1: Average runtimes for Interrogate (time in minutes). The entropy algorithms were tested with their default settings (window size is 256 bytes).

GB of RAM, can be found in Table 8.1. These times are an average of five measurements taken using the Unix `time` command, and are rather presented as an estimate of expected runtimes rather than statistically correct data. The disk reading operation of the memory dump has been left out by first loading the dump into memory, and then running Interrogate; the time averages in the table is measured from search algorithm start to end.

The results show the significant gap between non-optimized (AES-Search and Serpent-Search) algorithms and the optimized method for Twofish keys (Optimized-Twofish-Search) in terms of running time, from almost two hours to a couple of seconds. The Truecrypt-specific structural Twofish search (Truecrypt-Twofish-Search) is also faster and more accurate than its implementation independent counterpart. It should also be noted that the reconstruction of virtual memory space has a relatively constant runtime compared to the other algorithms, because the resulting memory to be searched for has the approximately same size regardless of memory dump size. For example, the reconstruction of `System.exe`'s virtual memory can in one instance take 10 seconds on a 1 GB dump, and reduce the search data (e.g., the reconstructed virtual memory) to only 10 MB. This significantly improves search performance, and makes it dependent on the number of pages that a process has in memory at the time of acquisition rather than physical memory size. Finally, the entropy based searches (Entropy-Search and Naive-Entropy-Search) are notoriously slow when the window size increases (the runtimes typically grow exponentially), and should be optimized.

Memory Usage and Management

Interrogate attempts to read the entire memory dump into memory at startup; this greatly enhances the performance when the host computer has enough memory compared to the size of the dump. If the memory image size approach the available physical memory at the host, excessive paging will occur, and this reduces performance.

It is also unable to handle files larger than 2 GB because of the standard 2 GB process address space. A *segmentation fault* will occur if when analysis of such large files are attempted. A dynamic file reading algorithm could probably counter both of these drawbacks.

Accuracy

Accuracy is here loosely defined as the tool's ability to correctly locate keys, in other terms, its *precision*. As we in most cases do not know the key's representation in the memory dump or if it is present at all, this term is not ideal, but it is used due to the lack of better terms.

We've generally designed the algorithms implemented in Interrogate to rather give false positives than false negatives, in order to facilitate our research on crypto key finding probabilities. While the AES-Search and Serpent-Search nearly have a 100% accuracy, the Optimized-Twofish-Search is much more greedy in terms of output. Several searches with this algorithm turned out 20 and more hits including the correct keys, many of which are duplicates. There is also a slight possibility for keys being "found" twice with the algorithm, as some structural properties may be overlapping.

The memory reconstruction method works as expected: it is crude and greedy in terms of pages fetched, but successfully reconstructs virtual memory space. The feature does however need further testing, as it was implemented in a late stage of the thesis and therefore were not scrutinized as rigorously as the rest of the code.

Interrogate's accuracy is additionally not very good when it comes to RSA keys, due to the implementation of RSA-Search and the structural differences in the representation of keys in memory in Linux and Windows. We were not able to locate any RSA keys in Windows at all using RSA-Search, while in Linux several matches were found. The accuracy on Unix platforms seems to be around 100%.

Entropy-Search and Naive-Entropy-Search are in general not accurate, and hard to use. Previous research on the key's entropy properties must have been performed to accurately pinpoint interesting regions. These regions can be quite wide in terms of bytes, further complicating the investigation. In addition, a large number of false positives are found regardless of system state. An implementation that also uses an upper threshold (e.g., search for a specific, narrow entropy value interval) would probably counter some of the false positives.

8.1.2 Limitations

Like previously mentioned, the entropy-based searches has proven to be of little value, as the memory size has grown significantly since Shamir suggested the approach. The usage of this type of search is therefore limited, and the method needs further development before these limitations can be mitigated.

We also attempted to implement a feature for checking of randomness in data, to be able to distinguish between (pseudo)random data and compressed data, etc. Both implementations of χ^2 test and the other tests mentioned in Section 2.3.3 were attempted, with poor results. It is however clear that many of these methods has properties that make them suitable for cryptographic key searches, as implemented with the Optimized-Twofish-Search algorithm.

8.1.3 Further Improvements

Based on our experience with Interrogate, we outline several improvements that would enhance its usability, accuracy and efficiency:

Large File Support To support files that has a size of 2 GB and greater, Interrogate should support dynamic reading of memory dumps. this would also improve performance for files larger than the memory available at the computer running it. In addition, *large file support* would enhance running times for host systems with large amounts of physical memory.

Reversing of the Twofish Key Schedule Generation To use a similar method to the AES key schedule on Twofish keys, we need to be able to infer the S-box and sub-keys from the 4 KB S-box table. To do this, we would have to reverse the generation of the 4 KB table and the S-box keys. Although this was not attempted, it could prove to be a viable approach. In addition, no attempts was made to deduce the master key from the key material found in memory. If such a deduction is computationally feasible, we would not need to modify current Twofish source code to apply the key material found. Both these (theoretic) improvements would greatly increase the accuracy of Twofish key schedule searches.

More Statistical Checks To improve the precision of the searches, further statistical experiments should be performed. As mentioned in Section 2.3.3, there exist several methods that could improve an entropy-based search.

Support More Ciphers and Key Sizes To expand the applicability of the tool, more algorithms and key sizes should be added to the supported ciphers list. Ciphers like CAST, DES and 3DES, ElGamal and DSA all has properties that would make them feasible for key search. For Twofish, more statistical analysis is needed to adjust the heuristics applied in the search algorithm to other key sizes, since the statistical properties of the S-boxes may change as a function of the key size.

Improve RSA-Search To improve the RSA-Search, one should consider the structure of RSA keys in Windows. One example of such a format can be found on Microsoft Developer Network (MSDN), under the name of `Private Key BLOB` [MSD08a].

Reverse Engineering To improve hit rates for closed-source applications, their key handling procedures could be reverse engineered, and search signatures formed from the obtained knowledge. Applications that this applies to, are among others Skype, Simp Lite and BestCrypt.

Integrate With Acquisition Tool To complete Interrogate as a forensic tool, it should be integrated with acquisition software and bootable software that can dump physical memory. One such small-footprint memory dumper and OS is `msramdmp` [McG08] and SYSLINUX. Furthermore, to make it capable of

handling bit-errors from coldboot-style acquisition methods, an error-correcting code functionality should be applied in Interrogate in accordance with [30].

Search For All Keys at Once Since most of our search methods are sequential and performed byte-by-byte, all of the search matching algorithms could be combined to effectively search for all key material present regardless of cipher. This would be a nice feature in the cases where the cipher used is unknown.

Check for Duplicate Keys A simple duplicate key checkup should be added to lessen output.

Include Pagefile in Memory Reconstruction In addition to improving the virtual memory reconstruction in terms of precision, the pagefile could easily be included in the process. Such an inclusion would build a complete virtual memory address space, with both valid and invalid (paged out) pages.

Support Windows Memory Modes Support for PAE and 3GB user address space should be implemented, so that the memory reconstruction method would work on target computers supporting these modes.

Support More OSes The memory reconstruction methods are highly OS-dependent, and support for other OSes like Linux, Windows Vista and Mac OS X should be implemented.

Increase Efficiency For several of our algorithms, easy improvements like checking for blank pages could greatly reduce the runtime of the application. In addition, we believe that several of the algorithms can be optimized further than the present level. The author of this thesis and the source code is no seasoned C programmer. The source code of the tool can probably, as a direct result of the above, be improved both with respects to code quality and efficiency. It is the author's hope that parts of the code can be incorporated in future memory forensics suites.

Comprehensive Documentation and Testing To further expand the usage of Interrogate, the tool needs to be fully documented and tested.

8.2 General Discussion

The *cryptographic software classes* defined in Section 6.5 mainly behaved according to our expectations. It may be interesting to notice that the notion of whole-disk encryption being vulnerable in *any* of the well-defined states were not present prior to some four months ago, when the coldboot article was published. Our expectations were formed from quite recent work in the field, and we believe that many corporations and individuals still don't know that in certain states, most software encryption solutions are broken. Although vendors claim "it's a feature [of DRAM], not a bug", the security of their applications clearly suffer from this "feature".

With that in mind, our results also indicate that most cryptographic applications feature strong key management. With some exceptions, namely PGP and ProtectDrive, keys were rarely encountered in unexpected states. These exceptions shows that key management is not trivial, and that cryptographic programming is recommendable only to the experts. Especially ProtectDrive seem to practice sloppy key management, and a full review of their code is recommended.

In Figure 8.1, the percentages of found keys per software class and system state is presented as a grouped histogram. If a state were not tested (e.g., it is represented with "N/A" in the result tables) it was not counted as a part of the data used to form this graph. A can be seen, a 100% hit rate were archived for the **Whole-disk** software class in the expected states. The **Virtual Disk** software state has slightly lower hit rates in its expected states, mostly due to our inability to locate Twofish keys with BestCrypt. We can also clearly see our failure to locate **Session-based** keys.

Even with good key management procedures, the **Whole-disk** and **Virtual Disk** systems are vulnerable to memory forensics due to the fact that they have to keep the key in memory at all times. As mentioned, this is not a design flaw, but rather a part of the design, and necessary both for performance and feasibility of on-the-fly encryption. This does however create a quite large window of opportunity for an adversary to dump and analyze memory.

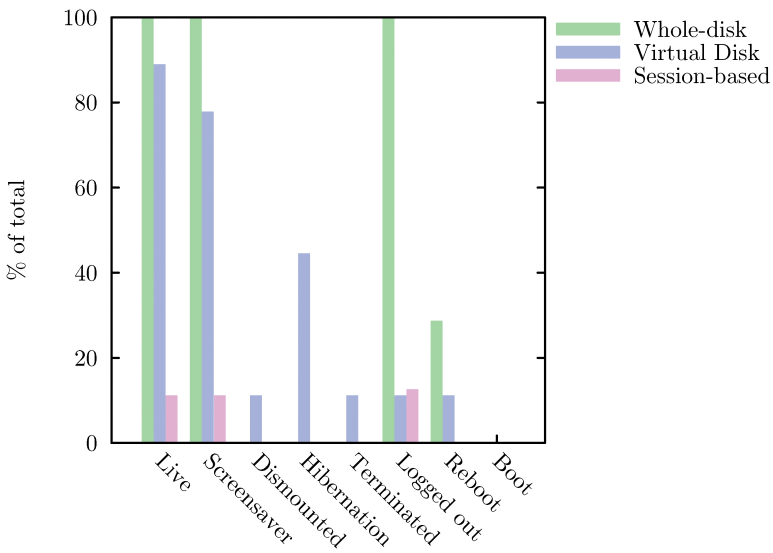


Figure 8.1: Percentages of found keys sorted by Software Class and State.

A smaller window of opportunity is present when dealing with **Session-based** cryptographic software. We were unable to locate any keys in this software class, and we suspect that the small window of opportunity combined with proprietary key structures and key obfuscation techniques is to blame. However, it is probably only a matter of time before these applications are reverse engineered, possibly making a key search feasible. In the mean time, memory analysis has proven effective against many of these applications by locating

plaintext copies in memory [108, 18].

As treated in the introduction in Chapter 1, we outlined two *cryptographic key search scenarios* to point to probable usage areas for cryptographic key recovery. When seen in the context of our results, several conclusions can be made on the feasibility of the scenarios. While finding key in memory from a whole-disk encrypted computer seem almost certain, we cannot present a reliable way of finding session-based keys, due to the fact that we were unable to locate them in our experiments (Except RSA keys from Apache). Therefore, we cannot conclude whether *post-capture decryption of communications* is feasible or not outing our methods. However, it has been shown that if a private SSL key is made available, it can be used to decrypt the dumped traffic post-capture [18]. Finding such a key is more likely to proceed through legal channels (e.g., demanding extradition of SSL private key from an internet host) than memory analysis at the moment.

We do believe that memory analysis and key extraction is a powerful addition to *secret searches*: Government agencies may in special cases perform searches that are performed prior to arrest of suspects, without the suspects knowledge [109][RIP00]. This method can be specially effective when working against organized crime and unlawful networks like pedophile rings, to secure evidence without risking alerting the more attractive suspects further up the food chain. Naturally such invasive techniques are restricted with heavy regulations based on national legislation. If such a search is successfully performed towards a suspect using whole-disk encryption, the investigators may continue their surveillance of the suspect knowing that they will be able to decrypt his disk if it is seized at a later point in time, regardless of computer state. A suspect may even feel a false sense of security using encryption, leaving evidence on his disk that he/she would not have done if encryption had not been in use. There is a slight risk that the key may be changed prior to acquisition, but all the whole-disk encryption systems we have tested change their keys rarely or never. We assume this is a security-performance trade-off, because changing the master key would require a full re-encryption of the disk. Nevertheless, not changing cryptographic keys regularly is bad cryptographic practice.

Of course, these scenarios are not exhaustive, and the applications of memory analysis are vast and yet mostly undiscovered. As the penetration of handheld devices in the business and private segments of the market increases, we're also likely to see an increase in encryption solutions suitable for these. Incidents like the one in the UK where social security numbers and sensitive information were compromised because of a stolen laptop [New06], are good catalyzers for cryptographic software. The increase will probably first be concentrated around software solutions, since these can fit on already manufactured units, and are generally cheaper than the hardware alternative. For most purposes, many would also argue that they provide sufficient security. However, handheld devices are often "always-on" devices, making them vulnerable to memory dumping attacks. In addition, they are small, easy to forget, easy to pickpocket and worst of all: full of information. The information on these devices may in many cases be extremely valuable (or sensitive) to the company or individual owning it, surpassing the value of the device itself by several magnitudes. Proper protection is therefore essential: Software encryption usually provide protection for the risk of data leakages, but to a skilled attacker, the always-on property together with memory analysis can indeed defeat this protection. Thus, we

expect to see tamper-resistant hardware encryption modules on business-class handheld devices in the future.

8.3 Towards a Forensically Sound Approach to Cryptographic Memory Forensics

The discussion so far has revolved around the proof of concept tool and the chances of uncovering keys from memory dumps. Having clarified Interrogate's weaknesses, outlined room for improvement and discussed the results in general, we will consider how it (or a similar tool) may be utilized to maximize the chances of key recovery for a digital investigator.

The results clearly indicate that the state of the system at the point of acquisition plays a vital role for an investigator. It is therefore increasingly important for a live response or forensics team to know what to do if a live system using cryptography is found.

First, upon arriving at a digital crime scene, it is desirable to be able to detect encryption. This is not always trivial. On-the-fly applications or any of the other **Whole-disk** and **Virtual Disk** encryption systems discussed in Chapter 7 can appear absent, since they only operate as a device at kernel level. Many of these systems runs as a couple of thread in a system process (like `System.exe`), and may therefore be hard to spot for an untrained eye. Except for these threads, whole-disk applications are often completely transparent, and the system may appear to not run any crypto at all.

To further complicate the matter, several cryptosystems feature hidden disk volumes/partitions, that may be invisible if not mounted [Fou08a]. This is to provide deniability, that is, to deny that the data even exits. If the volumes or partitions on the contrary *are* mounted, failing to dump their content before pulling the plug on the target computer may give an investigator an unpleasant surprise when attempting to analyze the hard drive, only to find that a large partition of it is encrypted.

The alternative to a memory dump when encountering encryption is, if possible, to extract the data while the system is live. Doing this, we do not honor the important principles of digital forensics, as mentioned in Section 4.1. Reading the content of an encrypted virtual or physical volume will potentially page out pages in memory, and therefore also effect the state of the hard drive. Thus, the risk of overwriting potential evidence is present. This risk should be assessed on a case-by-case basis, and compared against the risk of loosing data because of encryption. The comparison should then be used as a rationale to decide how and if the encrypted volumes should have their data extracted prior to imaging.

Furthermore, copying all the content of a drive using the target OS rises several trust issues. Rootkits and malware may have altered the OS, and therefore even normal disk read/write operations cannot be trusted [91] [Rut06]. The operating system or software running at the target may conceal encrypted partitions, or the fact that it is encrypted at all. In addition, copying of encrypted content is not always possible. Notably, the **Screensaver**, **Logged out** and **Hibernation** states can make such copying infeasible, and our research suggests that all these states have high success rates for key recovery given a memory dump. This all counts in favor of using imaging or dumping techniques instead

of live forensics, and it is therefore advisable that investigators should stick with the old paradigm of image first, analyze later.

If a partition or disk acquired turns out to be (partially) encrypted, the investigator may later face undecipherable data, and if no memory dump is available, he may be faced with a brute-force attempt as his only option. Consequently we can hardly recommend coldbooting a computer before documenting its internals and making reasonable sure that no encryption "booby traps" are set.

On the other side, if a memory dump *is* available, an investigator faced with an encrypted drive has more options than brute-forcing the encryption. Even the process of using each byte offset in the dump as a potential key is a significantly faster than brute-forcing modern ciphers. A memory dump may thus act as an insurance for the investigator; if encrypted material is uncovered later in the investigation, his chances of finding the key vastly improves if a memory dump was taken at the crime scene.

The feasibility of taking such a dump is unfortunately not the best, as discussed in Section 5.1. We believe that the methods will be further improved in the future, as there is great interest in the field from forensic organizations at the moment. However, today, investigators will often be faced with a troublesome acquisition procedure, that may yield results that cannot be trusted.

When a memory image has been secured, the investigation can proceed along the normal path of actions. The amount of data available for analysis will not grow substantially because of the dumping, as the physical memory versus disk space ratio decreases rapidly with new digital storage technologies[1].

The author believes that there's a substantial upside to memory dumping combined with classical digital forensics. The advantages of having memory dumps in an investigation will also likely rise, as the maturity of analyzing software increases.

From the above it is important to note that memory analysis and cryptographic key searches are not an alternative to classical digital forensics, but rather *an addition* to the existing methods. The methodologies are not mutually exclusive, as some investigators seem to think, and can easily be combined. Even though the acquisition methods are immature, they are plentiful, and the same goes for memory analysis. However, just as we needed to create our own tool to investigate the presence of crypto keys in memory, further understanding and reverse engineering of operating systems and cryptographic software is needed to fully take advantage of the potential that lies within memory analysis.

We believe that at present time, the investigator is faced with a core choice: *To dump memory or not.* As we will outline in the following section, we believe that memory dumping should be performed as routinely as disk imaging in *any* digital forensics investigation. This view is not only based on cryptographic considerations, but also on the fact that failing to dump memory effectively disregards a large portion of the digital crime scene, and hereby potential evidence. If cryptography is in use, the *consequences* of not taking a memory dump may be radically higher, as it may potentially destroy evidence.

[1] Typically, the relation between the size of physical versus secondary memory on a modern computer is approaching 1/1024, since terrabytes (TB) now are becoming increasingly common on hard drives, while physical memory still resides in the lower scale of gigabytes (1-4 GB).

8.4 A Proposal for Best Practice

Digital forensics is a highly situation-dependent field, and it is therefore hard to give recommendations set in stone on best practices, as each crime scene is different. The motive behind confiscation of data may also shift, for example consider a situation where the investigators must keep the machine powered on while searching for cryptographic keys. This would make several of the methods treated in this thesis impracticable. Therefore, some best practices are presented here as key points sorted on the two main tasks, *acquisition* and *analysis*.

8.4.1 Key Points for Best Practice Acquisition

The best practices in this section are designed to maximize the amount of data available to the investigator and the chances of uncovering cryptographic keys. The proposal fits into existing digital forensics frameworks as the IDIP model, and is intended as an addendum to existing processes. The key points mentioned here are thus to be performed together with existing digital forensics procedures.

Although not experimentally treated in this thesis, the current set of acquisition procedures for physical memory acquisition favor investigators that are technically skilled and has knowledge of what to do; and equally important: What *not* to do.

Educate Incident Response Teams It is vital to educate incident response teams and other personnel that performs normal digital acquisition procedures so that they integrate physical memory acquisition into existing procedures. Digital investigation frameworks are valuable only if they are used in real life, and to accommodate this, education is the only answer. Focus should lie on why and how physical memory should be acquired, and how to recognize that encryption is in use. For example, teams should know how to identify whole-disk cryptosystems. Many of these can be stealthy, but per default they usually use system tray icons like the ones in Figure 8.2, and this may be utilized as one of several steps for identification. It is also important to know that states like **Screensaver** and **Logged out** have high success-rates for extracting keys, especially for whole-disk encryption systems.

Figure 8.2: Taskbar Notification area icons. From left to right: DriveCrypt, Truecrypt, BestCrypt, PGP Desktop and ProtectDrive.

Prepare, Plan & Practice To be ready to acquire any computer's physical memory, sufficient preparation is needed. Acquisition software should be acquired, and incorporated in existing forensic toolkits. Forensics teams should create detailed plans, both general and specifically for each case, that have decision trees for each of the states that the target computer/device may be in. One should also consider handheld devices, as many of these have (or will have in the near future) encryption capabilities [110]. To be sure that the methods work

as expected, practice runs should be performed in a controlled environment. In this sense, it is important to remember that volatile memory is in real danger of disappearing permanently, in contrast to hard drives that are unlikely to be completely unreadable as a result of bad acquisition procedures.

Don't Pull the Plug (And if Necessary, Coldboot) The research in this thesis indicates that the chances of uncovering keys are highest while the computer is **Live**. If there is concern that the overall state of the machine may change due to automated processes, network interaction or other factors, we propose that network should first be disconnected, and if applicable, attempt to coldboot the computer. If possible, take a memory dump before coldbooting; remember that the physical memory likely will contain information about the automated process, and a lot of potential evidence. The system can ususally be halted in BIOS to preserve the state of the RAM, while preparations for extraction proceeds[2]. It is also recommendable to attempt to acquire hibernation and pagefiles before a hard reboot, but whether this is practicable or not must be assessed on a case-by-case basis. If the hibernation method is used for acquisition, eventual old hibernation files should be secured first.

Always Perform Full Memory Dumps (if Possible) It is absolutely essential to perform full memory dumps in any digital forensics investigation. In this thesis we have only touched a small part of the potential of memory forensics, and its usages are likely to rise. As mentioned before, the upside for such an acquisition is substantial, and a memory dump may provide invaluable if encrypted material is encountered at a later stage in the investigation. If full dumping is not possible, attempt to use process dumping or other means of obtaining the (parts of) the physical memory.

8.4.2 Key Points for Best Practice Analysis

Utilize Additional Resources Hibernation files, registry values and pagefiles should all be used in combination with a physical memory dump to facilitate a more comprehensive picture of the machine and software state.

Reconstruct Virtual Memory One utilization method for the additional resources addressed above, is reconstruction of virtual memory and processes. We attempted one such approach in this thesis, and several others are possible. Reconstruction of virtual address space reduce search data and facilitate searches that can locate keys that would otherwise not be possible. The method does also provide a significant performance gain.

Utilize Memory Analysis Tools As discussed in Section 5.2, there exist many freely available tools to analyze memory, any of which can be used to reduce the haystack when looking for cryptographic keys. One such utilization can be to extract process memory and use this as base for a cryptographic key search. Another approach can be to use PoolFinder (see Section 5.2.2) to find a specific application's allocated pages in the NonPaged pool.

[2]This procedure is dependent on the BIOS and the absence of a POST at boot.

Develop Methods to Preserve the Chain of Custody Different legislations have different demands for the preservation of the Chain of Custody, and analysis and documentation methods should of course be adjusted to the appropriate level. As mentioned earlier, it is usually impossible to verify volatile memory content against its original source at a later point in time. Law enforcement in countries with high demands to the Chain of Custody should develop routines to satisfy these demands. For example, the use of fuzzy hashing could be used to compare two memory dumps taken at an interval.

Utilize Virtualization Software to Experiment Virtualization software is an ideal tool to experiment with analysis techniques for volatile memory, as explained in Section 6.4. To be able to extract as much information as possible from memory dumps, forensics teams should test and reverse engineer cryptosystems and OSes, and use their findings in future investigations.

Perform Cryptographic Key Searches in Unlikely Situations Armored viruses and code obfuscation using encryption techniques are becoming more common in the malware world [111]. "Extorsion-ware", viruses and worms that hold files on unsuspecting users' hard drives as "hostages" by encrypting them, has recently emerged in the wild on the Internet [Lab08]. Cryptographic key searches can in some cases provide a solution to these situations, by deliberately infecting a constrained virtual machine, dump and analyze memory and recover the decryption key.

Last Resort: Using the Dump as Dictionary If an investigator face encrypted data and no keys are found using Interrogate or similar tools[3], it is feasible to use each byte offset in the memory dump as a potential key for a dictionary attempt. This is significantly more effective than a brute-force attempt on the encryption mechanism or password, for example, a 2 GB memory dump could possibly reduce the effort of breaking a 128-bit cipher with a factor of $2^{128}/2^{31} = 2^{97}$.

8.5 Limitations and Caveats

There are some limitations to our research. First of all, all acquisitions were performed during or immediately preceding execution of the cryptographic software, and we did not consider how long time user or kernel-level data survives in volatile memory. Research suggests that user-level data are unlikely to survive more than five minutes, even on a lightly loaded system. However, smaller segments and single pages can be found up to two hours after initial commit [34].

Secondly, because of time limitations, we also had to refrain from performing deeper analysis of many of the cryptographic applications. We fear that this has affected the results, especially in the session-based cryptographic software class.

[3]To the author's knowledge, no such tool exist freely at the time of writing.

Part III

Conclusions

Chapter 9

Conclusions

This thesis has discussed the search for cryptographic keys in the physical memory of computing devices, and how the state of the device affect the feasibility of such a search. We have explored different key types and treated digital memory acquisition, analysis and forensics. Furthermore, we have analyzed previously suggested methods and provided new algorithms for key identification and memory reconstruction, to facilitate search for keys in volatile memory. Implementations of these algorithms were used to build a proof of concept tool, that were used to search for keys in a virtual environment. By relating our research to well-defined states of computers running cryptographic software, a broad discussion regarding the feasibility and potential reward of performing memory dumping and analysis was given. We have also discussed the current paradigms of memory forensics and cryptography, and outlined how live response teams can maximize the chances of being able to extract keys from dumped physical memory.

As outlined in the problem definition, this thesis has unified memory analysis, cryptography and digital forensics in a way that will allow a higher success rate for law enforcement when encountering cryptographic applications on live digital crime scenes. We find the chances of locating encryption keys surprisingly high, to an extent where even the most brute-force approach usage of the memory dump would provide a significant performance boost compared to attempting to break the cipher itself.

Our research strongly suggests that finding cryptographic keys through a memory disclosure attack is an opportunistic approach, its success being dependent on the overall state of the target OS and cryptosystem. Particularly, the **Live**, **Screensaver** and **Logged out** states have high success rates[1], although our findings indicate that other more unexpected states may be vulnerable as well: Several of the cryptographic tools tested failed to properly wipe their keys after usage. Cryptographic systems that pre-compute cipher key schedules have all been found to be vulnerable to key schedule searches, adding up to a strong incentive to include memory dumping in existing digital forensics procedures. The author of this thesis strongly suggests integrating such a procedure, as disregarding volatile memory is disregarding a large part of the digital crime scene.

[1] For **Whole-disk** encryption systems, the success rate in our experiments was 100%.

From a security perspective, the main lesson that can be drawn from this is to never leave a computing device using encryption powered on unless it is in use or physically protected. The memory disclosure attacks described poses as a big threat against handheld devices, and the industry will need to shift its focus towards tamper-resistant hardware devices to mitigate the risk of compromising keys. Using the memory analysis techniques described in this thesis, a skilled attacker can defeat even the strongest software encryption.

While memory analysis and key identification may be possible when a **Live** computer is encountered, the outlook for such an identification is far more dismal when the computing device is turned off. Therefore, significant resources should be directed at the education of forensics teams and other personnel that are likely to encounter digital crime scenes, so that the right decisions are made to minimize the risk of data loss due to encryption.

9.1 Future Work

Further development is needed in the field of cryptographic memory analysis as outlined in Section 8.1, as well in the memory acquisition field. To expand the usage areas of memory forensics, a significant effort is needed to reverse engineer OS and application code. To be able to fully take advantage of the closed-source keys found in this report, applications that are able to make use of the keys are needed; for example encrypted virtual disk mounters. Experiments of cryptographic key searches on more applications are also needed; encryption is in use in far more types of software than the ones included by the three software classes defined in this thesis.

The field of memory forensics is as mentioned relatively young of age, and has yet to move out from a proof of concept stage to a fully fledged science. Further understanding of the memory internals of computers, including handheld devices, is needed to take memory forensics to the next level.

Based on this, several open research questions can be defined. We are still uncertain of the origin and usage of many of the keys found, and a further and deeper treatment of these would prove beneficial. A natural extension to this thesis would be to investigate mobile devices utilizing encryption, as these will be prevalent in the future digital forensics field. In addition, we see a great need for a good framework for incident response teams, complete with decision trees, with respects to volatile memory forensics.

Abbreviations

Abbreviation	Plaintext
AES	Advanced Encryption Standard
AIM	American Online (AOL) Instant Messenger
AKE	Authenticated Key Exchange
AMD	Advanced Micro Devices
ASCII	American Standard Code for Information Interchange
BSoD	Blue Screen of Death, the feared Microsoft error message
CAST	Cipher, from Carlisle Adams and Stafford Tavares
CRHF	Collision-Resistant Hash Function
DES	Digital Encryption Standard
DFRWS	Digital Forensics Research Conference
DH	Diffie-Hellman (key exchange or keys)
DRAM	Dynamic Random Access Memory
DRM	Digital Rights Management
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
FDE	Full Disk Encryption, Whole-disk encryption
GB	Gigabyte, 1024 MB
GCC	GNU Compiler Collection
GF	Galois Field
GNFS	General Number Field Sieve
GNU	GNU's Not Unix
GPG	GNU Privacy Guard
GPL	GNU Public License
GUI	Graphical User Interface
HTTP(S)	Hyper Text Transfer Protocol (SSL/TLS)
IDEA	International Data Encryption Algorithm
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IM	Instant Messaging
IP	Internet Protocol
IT	Information Technology
IV	Initialization Vector
KB	Kilobyte, 1024 bytes
LSB	Least Significant Bit
MB	Megabyte, 1024 KB
MD5	Message Digest algorithm nr. 5
MDS	Maximum Distance Separable

MIPS	Million Instructions Per Second
MitM	Man-in-the-Middle (attack)
MS	Microsoft
MSDN	Microsoft Developer Network
MSB	Most Significant Bit
MSN	Microsoft Network
MY	MIPS Year
NCIS	Norwegian National Criminal Investigation Service
NIC	Network Interface Card
NIST	National Institute of Standards and Technology
OFB	Output FeedBack (mode)
OOV	Order Of Volatility
OS	Operating System
OTR	Off-the-record (messaging)
OWHF	One-Way Hash Function
PAE	Page Address Extension
PDE	Page Directory Entry
PDB	Page Directory Base
PDT	Page Directory Table
PFN	Page Frame Number
PGP	Pretty Good Privacy, a public-key encryption system
PHT	Pseudo-Hadamard Transform
PIN	Personal Identification Number (four digit password)
POST	Power-On Self-Test
POTS	Plain Old Telephone System
PRNG	Pseudo-Random Number Generator
PTE	Page Table Entry
RAM	Random Access Memory
RNG	Random Number Generator
ROM	Read-Only Memory
RSA	Short for "Rivest, Shamir, & Adleman"
RSAP	RSA Problem
SHA-X	Secure Hashing Algorithm (X is the output bit-length)
SIGMA	"SIGn-and-MAc"
SP	Substitution-Permutation (network)
SQL	Structured Query Language
SRAM	Static Random Access Memory
SSH	Secure Shell
SSL	Secure Sockets Layer
TB	Terrabyte, 1024 GB
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TPG	Trusted Platform Group
TPM	Trusted Platform Module
URL	Universal Resource Locator
USB	Universal Serial Bus
VAD	Virtual Address Descriptor
VM	Virtual Machine
WDT	Windows Debugging Tools

Publications

[1] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[2] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Simon & Schuster, 2rev edition, 1997.

[3] E. Mumford. *Dangerous Decisions: Problem Solving in Tomorrow's World.* Springer, 1999.

[4] David Salomon. *Data Privacy and Security.* Springer Professional Computing, 2003.

[5] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST SP800-90 dual EC PRNG. Presentation at the CRYPTO 2007 conference, November 2007.

[6] Mark Stamp and Richard M. Low. *Applied Cryptanalysis.* Wiley, 2007.

[7] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, 2. edition, 1995.

[8] Dorothy E. Denning. *Information Warfare and Security.* Addison Wesley, 1999.

[9] Barak D. Jolish. The encryption debate in plaintext: National security and encryption in United States and Israel. *Lecture Notes in Computer Science*, 1962:202–224, 2001.

[10] Robert Richardson. Csi computer crime and security survey 2007. Technical report, CSI, 2007.

[11] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, 9:5–38, 161–191, 1883.

[12] NIST. Announcing the Advanced Encryption Standard (AES). FIPS 197, NIST, 2001.

[13] Joan Daemen and Vincent Rijmen. *Design of Rijndael: Aes - the Advanced Encryption Standard.* Springer-Verlag Berlin and Heidelberg GmbH Co, 2002.

[14] RSA. PKCS 1 v2.1: RSA cryptography standard. Technical report, RSA Laboratories, 2002.

[15] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. August 2006.

[16] Benjamin Adida, Lauren B. Fletcher, Lydia Sandon, Enouch Chang, Michelle Hong, and Kristina Page. The future of trespass and property in cyberspace. Technical report, Massachusetts Institute of Technology and Harvard Law School, 1998.

[17] Trusted Platform Group. Trusted platform modules strengthen user and platform authenticity. Technical report, Trusted Platform Group, 2005.

[18] Carsten Maartmann-Moe. Digital evidence and cryptography. Minor thesis, Norwegian University of Science and Technology, 2007.

[19] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[20] Timothy Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1(4):315–323, 2006.

[21] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson3. How to forget a secret. *Lecture Notes in Computer Science*, 1563:500–509, 1999.

[22] Nicolas Ruff. Windows memory forensics. *Journal in Computer Virology*, 2007.

[23] Maximillian Dornseif. Firewire - all your memory are belong to us. Presentation at CanSecWest/Core05, 2005.

[24] Antonio Martin. Firewire memory dump of a Windows XP computer: A forensic approach. 2007.

[25] Adi Shamir and Nicko van Someren. Playing hide and seek with stored keys. *Lecture Notes in Computer Science*, 1998.

[26] Torbjörn Pettersson. Cryptographic key recovery from linux memory dumps. In *Chaos Communication Camp*, 2007.

[27] Tobias Klein. All your private keys are belong to us. Tutorial, 2006.

[28] Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *Dependable Systems and Networks*, 2007.

[29] AAron Walters, , and Jr. Nick L. Petroni. Volatools: Integrating volatile memory forensics into the digital investigation process. Technical report, Komoku, Inc., 2007.

[30] J. Alex Halderman, Seth. D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. 2008.

[31] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[32] Andreas Schuster. Searching for processes and threads in Microsoft Windows memory dumps. In *DFRWS*, 2006.

[33] Harlan Carvey. *Windows Forensic Analysis*. Syngress, 2007.

[34] Jason Solomona, Ewa Huebnera, Derek Bema, and Magdalena Szezynska. User data persistence in physical memory. *Digital Investigation*, 4:68–72, 2007.

[35] Bent Kristoffer Onshus. Cryptographic credentials and encrypted data in digital evidence. Minor thesis, Norwegian University of Science and Technology, 2005.

[36] Andreas Grytting Furuseth. Digital forensics : Methods and tools for retrieval and analysis of security credentials and hidden data. Master's thesis, Norwegian University of Science and Technology, 2005.

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[38] Michael Welschenbach. *Cryptography in C and C++*. Apress, 2 edition, 2005.

[39] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology - CRYPTO 2003*, 2729:617–630, 2003.

[40] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer Verlag, 1993.

[41] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Advances in Cryptology*, pages 2–21, 1990.

[42] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). Technical report, NIST, 2000.

[43] Vincent Rijmen and Joan Daemen. AES proposal: Rijndael. 1999.

[44] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the Advanced Encryption Standard. Technical report, Cambridge University, 2000.

[45] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. In *Third AES Candidate Conference*, 2000.

[46] National Institute of Standards and Technology. Data Encryption Standard (DES). FIPS 46-2, NIST, 1975.

[47] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. Wiley, 1999.

[48] Ross Anderson, Eli Biham, and Lars Knudsen. The case for serpent. In *Third AES Candidate Conference*, 2000.

[49] C. Adams and J. Gilchrist. The CAST-256 encryption algorithm. RFC 2612, Network Working Group, 1999.

[50] C. Adams. The CAST-128 encryption algorithm. RFC 2144, Network Working Group, 1997.

[51] Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.

[52] E. Rescorla. Diffie-Hellman key agreement method. RFC 2631, IETF, 1999.

[53] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Technical report, Massachusetts Institute of Technology, 1977.

[54] NIST. Announcing the Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186, NIST, January 2000.

[55] Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[56] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. *Lecture Notes in Computer Science*, 1423:267–288, 1998.

[57] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical report, Certicom Research and University of Waterloo, 1999.

[58] NIST. Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. Special Publication 800-56A, NIST, 2006.

[59] Johannes Blömer. Randomness and secrecy - a brief introduction. *Journal of Universal Computer Science*, 12(6):654–671, 2006.

[60] Olav Kallenberg. *Random Measures*. Akademie-Verlag, 1986.

[61] Donald E. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 3 edition, 1997.

[62] RAND. *A Million Random Digits with 100, 000 Normal Deviates*. RAND, 2001.

[63] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[64] NIST. Security requirements for cryptographic modules. FIPS 140, NIST, 1994.

[65] Electronic Frontier Foundation. *Cracking DES*. O'Reilly Associates, Inc, 1998.

[66] Dag Arne Osvik and Eran Tromer. Cryptanalytic applications of the PlayStation 3: the case of DES. Presentation at the Rump session at SHARCS, 2006.

[67] Whitfield Diffie. Ultimate cryptography. *Communications of the ACM*, 2001.

[68] Matthew E. Briggs. An introduction to the general neumber field sieve. Master's thesis, Virginia Polytechnic Institute and State University, 1998.

[69] On Polynomial Selection For the General Number Field Sieve. Thorsten kleinjung. *Mathematics of Computation*, 75(256), 2006.

[70] Carl Pomerance. A tale of two sieves. *Notices of the AMS*, 1996.

[71] Jens Franke. On the factorization of RSA200. Presentation at SHARCS'06, April 2006.

[72] Arjen K. Lenstra. Matching AES security using public key systems. In *Asiacrypt*, 2001.

[73] National Bureau of Standards. Password usage. FIPS 112, U.S. Department of Commerce, 1985.

[74] Boris Balacheff, Liqun Chen, Siani Pearson, David Plaquin, and Graeme Proudler. *Trusted Computing Platforms*. Prentice Hall, 2003.

[75] ITU-T. Abstract Syntax Notation One (ASN.1) specification of basic notation. Reccomandation X.680, ITU-T, 2002.

[76] RSA. PKCS 8: Private-key information syntax standard. Technical report, RSA, 1993.

[77] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. On the Twofish key schedule. *Lecture Notes in Computer Science*, (1556), 1998.

[78] Fauzan Mirza and Sean Murphy. An observation on the key schedule of Twofish. Technical report, Information Security Group, Royal Holloway, University of London, 1999.

[79] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Further observations on the key schedule of Twofish. Twofish technical report, 1999.

[80] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 2005.

[81] George Mohay, Alison Anderson, Byron Collie, Olivier de Vel, and Rodney McKemmish. *Computer and Intrusion Forensics*. Artech House, 2003.

[82] W. Kruse and J Heiser. *Computer Forensics: Incident Response Essentials*. Addison-Wesley, 2002.

[83] Brian Carrier and Eugene Spafford. An event-based digital forensic investigation framework. *Digital Forensic Research Workshop*, 2004.

[84] Kevin Mandia and Chris Prosise. *Incident Response: Investigating Computer Crime*. McGraw-Hill, 2001.

[85] Marc Rogers, Jim Goldman, Rick Mislan, Tim Wedge, and Steve DeBrota. Computer forensics field triage process model. In *2006 Conference on Digital Forensics, Security and Law*, 2006.

[86] Brian Carrier and Eugene Spafford. Getting physical with the digital investiagtion process. *International Journal of Digital Evidence*, 2, 2003.

[87] D. Brezinski and T. Killalea. Guidelines for evidence collection and archiving. RFC 3227, IETF, 2002.

[88] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, 2005.

[89] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics. In *Asiacrypt*, volume 4284, pages 1–20, 2006.

[90] Brian D. Carrier. Live digital forensic analysis. *Communications of the ACM*, 2006.

[91] Joanna Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition tools. Presentation at Black Hat DC 2007, 2007.

[92] Adam Boileau. Hit by a bus: Physical access attacks with Firewire. Presentation at Ruxcon 2006, 2006.

[93] Bradley Schaltz. Bodysnatcher: Towards reliable volatile memory acquisition by software. *Digital Investigation*, 4:126–134, 2007.

[94] Matthieu Suiche. Sandman project. Technical report, 2008.

[95] Nicolas Ruff and Matthieu Suiche. Enter Sandman. Presentation at PacSec 2007, 2007.

[96] Andreas Schuster. Pool allocations in Windows memory forensics. Presentation at IMF 2006, 2006.

[97] AAron Walters. Fatkit: Detecting malicious library injection and upping the "anti". *Digital Investigation*, 3(4):197–210, 2006.

[98] Brendan Dolan-Gavitt. The VAD tree: A process-eye view of physical memory. *Digital Investigation*, (4):62–64, 2007.

[99] Michael Richmond. ViSe: A virtual security testbed. Master's thesis, Department of Computer Science, University of California, Santa Barbara, 2005.

[100] André Årnes, Paul Haas, Giovanni Vigna, and Richard A. Kemmerer. Digital forensic reconstruction and the virtual security testbed ViSe. *Detection of Intrusions and Malware Vulnerability Assessment*, pages 144–163, 2006.

[101] André Årnes, Paul Haas, Giovanni Vigna, and Richard A. Kemmerer. Using a virtual security testbed for digital forensic reconstruction. *Journal in Computer Virology*, 2(4):275–289, 2007.

[102] Michael Stay. Zip attacks with reduced known plaintext. *Lecture Notes in Computer Science*, 2355, 2002.

[103] Tadayoshi Kohno. Attacking and repairing the WinZip encryption scheme. In *11th ACM Conference on Computer and Communications Security*, 2004.

[104] S.-W. Yeo and C.-W Phan. On the security of the winrar encryption feature. *International Journal of Information Security*, 2006.

[105] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. Presentation at Black Hat Europe 2006, 2006.

[106] Tom Berson. Skype security evaluation. Technical report, Anagram Laboratories, 2005.

[107] Jacob Appelbaum and Ralf-Philipp Weinmann. Unlocking FileVault. Presentation at 23rd Chaos Communication Congress, December 2006.

[108] Eoghan Casey. Practical approaches to recovering encrypted evidence. *International Journal of Digital Evidence*, 1(3), 2002.

[109] Department of Justice. The USA PATRIOT Act: Preserving Life and Liberty, 2001.

[110] Michael W. Burnette. Forensic examination of a RIM (BlackBerry) wireless device. Technical report, Rogers & Hardin LLP, 2002.

[111] Eric Filiol. Strong cryptography armoured computer viruses forbidding code analysis: the bradley virus. In *EICAR Best Paper Proceedings*, 2005.

Web References

[And00] Ross Anderson, *Serpent home page*, 2000, http://www.cl.cam.ac.uk/~rja14/serpent.html.

[App08] Apple, *OS X Leopard - security*, 2008, http://www.apple.com/macosx/technology/security.html.

[Bet05] Chris Betz, *Memparser*, 2005, http://sourceforge.net/projects/memparser.

[Car06] Harlan Carvey, *Windows IR*, 2006, http://sourceforge.net/projects/windowsir.

[Cop07] Copacobana, *COPACOBANA project*, 2007, http://www.copacobana.org/, visited November 4, 2007.

[DFR05] DFRWS, *Digital forensics research conference 2005 forensic challenge*, 2005, http://www.dfrws.org/2005/challenge.

[Ef05] E-fense, *Helix - incident response computer forensics live cd*, 2005, http://www.e-fense.com/helix.

[Fel08] Ed Felten, *New research result: Cold boot attacks on disk encryption*, 2008, http://www.freedom-to-tinker.com/?p=1257.

[Fou08a] Truecrypt Foundation, *Truecrypt*, 2008, http://www.truecrypt.org/.

[Fou08b] ———, *Truecrypt source code*, 2008, http://www.truecrypt.org/downloads2.php.

[Gel05] Barton Gellman, *The FBI's secret scrutiny*, Washington Post (2005), http://www.washingtonpost.com/wp-dyn/content/article/2005/11/05/AR2005110501366.html, visited November 14.

[Gla06] Brian Gladman, *AES and combined encryption/authentication modes*, 2006, http://fp.gladmand.plus.com/AES/index.thm.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten, *Lest we remember: Cold boot attacks on encryption keys*, 2008, http://citp.princeton.edu/memory/.

[Inc07] GMG Systems Inc, *Knttools*, 2007, http://www.gmgsystemsinc.com/knttools.

[Jet08] Jetico, *Bestcrypt faq*, 2008, http://www.jetico.com/bestcrypt_faq.htm.

[Kal93] Burton S. Kaliski, *A layman's guide to a subset of ASN.1, BER, and DER*, RSA laboratories technical note, RSA Laboratories, 1993.

[Kle06] Tobias Klein, *Process dumper*, 2006, http://www.trapkit.de/research/forensic/pd.

[Kor07] Jesse Kornblum, *Ssdeep*, 2007, http://ssdeep.sourceforge.net/.

[Lab08] Kapersky Lab, *Kapersky lab reports a new and dangerous blackmailing virus*, 2008, http://www.kapersky.com/news?id=207575650.

[LF05] Mathieu Lafon and Romain Francoise, *Information leak in the linux kernel ext2 implementation*, 2005, http:arkoon.net/advisories/ext2-make-empty-leak.txt.

[McC08] Declan McCullagh, *Disk encryption may not be secure enough, new research finds*, 2008, http://www.news.com/8301-13578_3-9876060-38.html.

[McG08] Wesley McGrew, *msramdmp: Mcgrew security ram dumper*, 2008, http://mcgrewsecurity.com/projects/msramdmp/.

[Mic07] Microsoft, *Windows internals*, 2007, http://www.microsoft.com/technet/sysinternals/information/windowsinternals.mspx.

[Mic08a] _____, *Debugging tools for Windows*, 2008, http://www.windows.com/whdc/devtools/debugging/.

[Mic08b] _____, *Memory limits for Windows releases*, 2008, http://msdn2.microsoft.com/en-us/library/aa366778.

[Mic08c] _____, *Windows Vista: Features explained: BitLocker drive encryption*, 2008, http://www.microsoft.com/windows/products/windowsvista/features/details/bitlocker.mspx.

[MSD08a] MSDN, *Private key blobs*, 2008, http://msdn.microsoft.com/en-us/library/aa387401.aspx.

[MSD08b] _____, *Windows driver kit*, 2008, http://msdn2.microsoft.com/en-us/library/aa972908.

[New06] BBC News, *Security raised over laptop theft*, 2006, http://news.bbc.co.uk/1/hi/uk/6160800.stm.

[Old04] *The old new thing: The oft-misunderstood /3gb switch*, 2004, http://blogs.msdn.com/oldnewthing/archive/2004/08/05/208908.aspx.

[Pta08] Thomas Ptacek, *Recover a private key from process memory*, 2008, http://www.matasano.com/log/178/recover-a-private-key-from-process-memory/.

[RIP00] *Regulation of Investigatory Powers Act (RIPA)*, 2000, http://www.opsi.gov.uk/acts/acts2000/20000023.htm.

[Rus05] Mark E. Russinovich, *Unkillable processes (NotMyFault)*, 2005, http://blog.technet.com/markrussinovich/archive/2005/08/17/unkillable-processes.aspx.

[Rut06] Joanna Rutkowska, *Introducing blue pill*, 2006, http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[Sch98] Bruce Schneier, *Twofish*, 1998, http://www.schneier.com/twofish.html.

[Sch07] Andreas Schuster, *Forensic blog*, 2007, http://computer.forensikblog.de/en/.

[Sch08] Bruce Schneier, *My data, your machine*, 2008, http://www.schneier.com/essay-142.html.

[Sys07] Volatile Systems, *The Volatility framework*, 2007, https://www.volatilesystems.com/VolatileWeb/volatility.asp.

[Vid06] Arne Vidström, *pmdump*, 2006, http://www.ntsecurity.nu/toolbox/pmdump.

[VMw07] VMware, *The VMware site*, 2007, http://www.vmware.com.

[Wal08] John Walker, *ENT: A pseudorandom number sequence test program*, 2008, http://www.fourmilab.ch/random/.

[Wik08] Wikipedia, *Instructions per second*, 2008, http://en.wikipedia.org/wiki/Instructions_per_second.

[Wil08] Svein Y. Willassen, *Moments in time: Evidence is not proof*, 2008, http://willassen.blogspot.com/2008/04/evidence-is-not-proof.html.

[Win06] WinZip, *AES encryption information: Encryption specification AE-1 and AE-2*, 2006, http://www.winzip.com/aes_info.htm.

[Zet08] Kim Zetter, *Researchers: Disk encryption not secure*, 2008, http://blog.wired.com/27bstroke6/2008/02/researchers-dis.html.

Part IV

Appendices

Appendix A

Source Code

A.1 interrogate.h

Listing A.1: interrogate.h

```
/* ========================================================================
 * interrogate.h
 *
 * Main header file for Interrogate: Structural and entropy-based search for
 * crypto keys in binary files or memory dumps.
 *
 * http://interrogate.sourceforge.net
 *
 * Copyright (C) 2008 Carsten Maartmann-Moe <carmaa@gmail.com>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 * ========================================================================
 */

#define NOFSYMBOLS      256 /* Number of symbols in alphabet (ASCII=256) */
#define WINDOWSIZE      256 /* Windowsize in BYTES */
#define KEYSIZE         256 /* Default keysize in BITS */
#define THRESHOLD       7.0 /* Default entropy threshold */
#define BCMOD           20  /* Modifier for byte count threshold */
#define TRUE            1
#define FALSE           0

#define NO_KEYTYPE      -1  /* Keytype definitions below */
#define AES             0
#define RSA             1
#define SERPENT         2
#define TWOFISH         3
#define TWOFISH_TC      4
#define NOF_KEYTYPES    5

#define LEFT 0
#define RIGHT 1

#define rotlFixed(x,n)   (((x) << (n)) | ((x) >> (32 - (n))))
#define rotrFixed(x,n)   (((x) >> (n)) | ((x) << (32 - (n))))
```

```
#define NOF_TF_IMP      4;  /* Number of Twofish implementations */
#define TF_SBOX_SIZE    4096;
#define TF_RUNS         6   /* Runs to measure */

/* ----------------------------
 * Twofish key structures below
 * ----------------------------
 */

/* Twofish key structure, taken from TrueCrypt implementation */
typedef struct {
    unsigned int l_key[40];
    unsigned int s_key[4];
    unsigned int mk_tab[4 * 256];
    unsigned int k_len;
}
twofish_tc;

/* Twofish key sructure from Linux and GPG implementations
 * Isomorphic with SSH impelentation below as far as we are concered. */
typedef struct {
    unsigned int s[4][256], w[8], k[32];
}
twofish_gpg;

/* SSH twofish key schedule */
typedef struct {
    unsigned int s[4][256];               /* Key-dependant S-Boxes */
    unsigned int k[40];                   /* Expanded key words    */
    int for_encryption;                   /* encrypt / decrypt     */
}
twofish_ssh;

/* Twofish key structure taken from Nettle */
typedef struct {
    unsigned int k[40];
    unsigned int s[4][256];
}
twofish_nettle;

/* Twofish optimized implementation */
typedef struct {
    unsigned int K[40];
    unsigned int k_len;
    unsigned int QF[4][256];
}
twofish_opt;

/* Page Table Entry struct (PTE). Note that Windows uses the
 * same structure for Page Directory Entries (PDEs).
 */
typedef struct {
unsigned int valid :
    1;
unsigned int write :
    1;
unsigned int owner :
    1;
unsigned int write_through :
    1;
unsigned int cache_disabled :
    1;
unsigned int accessed :
    1;
unsigned int dirty :
    1;
unsigned int large_page :
    1;
unsigned int global :
    1;
unsigned int copy_on_write :
    1;
unsigned int transition :
```

```
    1;
unsigned int prototype :
    1;
unsigned int pfn :
    20;
}
pte;

/* Virtual address for 32-bit x86 Windows systems */
typedef struct {
unsigned int byte_offset :
    12;
unsigned int pt_index :
    10;
unsigned int pd_index :
    10;
}
virtual_address;

/* Interrogate context */
typedef struct {
    int     keytype,        /* Keytype to be searched for */
    keysize,                /* The key size that are to be searched for */
    wsize,                  /* The search window size */
    nofs,                   /* The number of symbols in our alphabet */
    bitmode,                /* Bitmode boolean */
    verbose,                /* Verbose mode */
    naivemode,              /* Calculate true entropy */
    quickmode,              /* Non-overlapping entropy windows */
    interval,               /* Only search in interval (boolean) */
    from,                   /* Starting point */
    to,                     /* End point */
    cr3,                    /* CR3 offset in case recunstruction of mem */
    filelen,                /* Input file length in bytes */
    bytethreshold;          /* Threshold for bytecount */
    FILE    *output_fp;     /* Pointer to output file for statistics */
    float   threshold;      /* Entropy threshold */
    long    count;          /* Number of keys found */
}
interrogate_context;

/* -------------------
 * Function prototypes
 * -------------------
 */

/* interrogate.c: Main Program */
void init(float *ek);
void initialize();
void keysearch(interrogate_context *ctx, unsigned char *buffer);
void search(interrogate_context *ctx, unsigned char *buffer);
void quicksearch(interrogate_context *ctx, unsigned char *buffer);
void rsa_search(interrogate_context *ctx, unsigned char *buffer);
void aes_search(interrogate_context *ctx, unsigned char *buffer);
void serpent_search(interrogate_context *ctx, unsigned char *buffer);
void twofish_search(interrogate_context *ctx, unsigned char *buffer);
void twofish_search_old(interrogate_context *ctx, unsigned char *buffer);

/* stat.c: Statistics */
double approxlog2(double x);
float ent(interrogate_context *ctx, unsigned char *buffer, int length);
float *ent_opt(unsigned char *buffer);
int countbytes(interrogate_context *ctx, unsigned char *buffer);
void runs(interrogate_context *ctx, unsigned char *buffer, int *runs_count,
          int run_length, int *firstrun, int *lastrun);
void runs_opt(interrogate_context *ctx, unsigned char *buffer,
              int *runs_count, int run_length, int *firstrun, int *lastrun);

/* rsa.c: RSA functions */
int parse_der(unsigned char *buffer, int offset);
void output_der(unsigned char *buffer, int offset, size_t size, long *count);

/* aes.c: AES functions */
void rotate(unsigned char *in);
```

```
unsigned char rcon(unsigned char in);
unsigned char gmul(unsigned char a, unsigned char b);
unsigned char gmul_inverse(unsigned char in);
unsigned char sbox(unsigned char in);
void schedule_core(unsigned char *in, unsigned char i);
void expand_key(unsigned char *in);
void expand_key_192(unsigned char *in);
void expand_key_256(unsigned char *in);

/* serpent.c: Serpent functions */
void serpent_set_key(const unsigned char userKey[], int keylen,
                     unsigned char *ks);

/* twofish.c TwoFish functions */
void twofish_set_key(twofish_tc *instance, const unsigned int in_key[],
                     const unsigned int key_len);
unsigned int mds_rem(unsigned int p0, unsigned int p1);
void gen_mk_tab(twofish_tc *instance, unsigned int key[]);

/* nppool.c Nonpaged Pool functions */
void reconstruct(interrogate_context *ctx, unsigned char *buffer);
void print_pte(virtual_address *addr, pte *pd, pte *pde, pte *pt, pte *pte,
               unsigned char *page);

/* util.c: Utility functions */
unsigned char *read_file(interrogate_context *ctx, FILE *fp);
FILE *open_file(interrogate_context *ctx, char *filename, char *mode);
int checkbyte(unsigned char index, int *array);
void printblobinfo(int start, int end, int bytes, float wins, float entropy);
void print_hex_array(unsigned char *buffer, int length, int columns);
void print_hex_words(unsigned int *buffer, int length, int columns);
int validkeytype(char *keytype, int length);
int min(int a, int b);
void print_to_file(FILE *fp, float value);
unsigned getbits(unsigned x, int p, int n);
unsigned int byteshift(unsigned int x, int direction, int n);
int is_mk_tab(int *run);
void validate_tf_ks(interrogate_context *ctx, unsigned char *buffer,
                    int offset);
double format(double Value, int nPrecision);
```

A.2 interrogate.c

Listing A.2: interrogate.c

```
/* ========================================================================
 * interrogate.c
 *
 * Structural and entropy-based search for crypto keys in binary files or
 * memory dumps.
 *
 * http://interrogate.sourceforge.net
 *
 * Copyright (C) 2008 Carsten Maartmann-Moe <carmaa@gmail.com>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 * ========================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <string.h>
#include <time.h>

#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#else
#include <unistd.h>
#endif

#include "interrogate.h"

/*
 * Main search method.
 *
 * Reads entire file (memory dump) into memory and searches file for
 * cryptographic keys. Dispatches appropriate searching method based on user
 * input (e.g., the switches set at the command line. Also prints some
 * headers for entropy searches.
 */
void keysearch(interrogate_context *ctx, unsigned char *buffer) {

    printf("Success, starting search.\n\n");

    if ((ctx->keytype == NO_KEYTYPE)) {
        printf(" Interval            | Size     | Windows | %s\n",
                (ctx->naivemode) ? "Entropy" : "Byte Count");
    }
    printf("----------------------------------------"
        "----------------------------------------\n");

    /* Set filelen to be the interval ending point if interval mode is
     * set */
    if (ctx->interval)
        ctx->filelen = ctx->to;

    /* Search */
    switch (ctx->keytype) {
    case RSA:
        rsa_search(ctx, buffer);
```

```
        break;
    case AES:
        aes_search(ctx, buffer);
        break;
    case SERPENT:
        serpent_search(ctx, buffer);
        break;
    case TWOFISH:
        twofish_search(ctx, buffer);
        break;
    case TWOFISH_TC:
        twofish_search_old(ctx, buffer);
        break;
    default:
        if (ctx->quickmode) {
            quicksearch(ctx, buffer);
        } else {
            search(ctx, buffer);
        }
        break;
    }

    free(buffer);
}

/* ============================================================
 * Searchfunctions for RSA, AES, SERPENT and TWOFISH key types.
 * ============================================================
 */

void rsa_search(interrogate_context *ctx, unsigned char *buffer) {
    int i;

    /* Calculate der-encoding parameters like lenght of data blob etc.
     * according to PKCS #8 */
    int FLAG1 = 0x30;
    int FLAG2 = 0x82;

    /* Set interval parameter */

    if (ctx->interval)
        ctx->filelen = ctx->to;

    for (i = ctx->from; i < ctx->filelen - 1; i += 2) {
        unsigned char c1, c2, c3;
        int foundAt = -1;
        c1 = (unsigned char) buffer[i];
        c2 = (unsigned char) buffer[i + 1];
        if (c1 == FLAG1) {
            if (c2 == FLAG2) {
                foundAt = i;
            }
        } else if (c2 == FLAG1) {
            c3 = (unsigned char) buffer[i + 2];
            if (c3 == FLAG2) {
                foundAt = i + 1;
            }
        }
        if (foundAt != -1) {
            if (ctx->verbose)
                printf("Signature hit...");
            int derLength;
            if ((derLength = parse_der(buffer, foundAt))) {
                ctx->count++;
                output_der(buffer, foundAt, derLength, &(ctx->count));
                //Skip the bytes containing the key
                i += derLength;
            } else {
                if (ctx->verbose)
                    printf("not a key.\n");
            }
        }
    }
}
```

```c
void aes_search(interrogate_context *ctx, unsigned char *buffer) {
    int i;

    /* Set key schedule sizes */
    int kssize = 176;
    if (ctx->keysize == 192) {
        kssize = 208;
    } else if (ctx->keysize == 256) {
        kssize = 240;
    }

    unsigned char *ks = malloc(kssize * sizeof(unsigned char));

    for (i = ctx->from; i < ctx->filelen - kssize; i++) {
        /* Copy a chunk of data from buffer, expand it using AES key
         * schedule routines */
        ks = memcpy(ks, &buffer[i], kssize);
        if ((ctx->keysize == 128))
            expand_key(ks);
        else if ((ctx->keysize == 192))
            expand_key_192(ks);
        else
            expand_key_256(ks);
        /* Compare expanded key schedule to the data proceeding the chunk */
        if (memcmp(ks, &buffer[i], kssize) == 0) {
            ctx->count++;
            printf("Found (probable) AES key at offset %.8x:\n", i);
            print_hex_array(ks, ctx->keysize / 8, 16);
            printf("Expanded key:\n");
            print_hex_array(ks, kssize, 16);
        }
    }

}

void serpent_search(interrogate_context *ctx, unsigned char *buffer) {
    int i;
    /* Key schedule size for SERPENT is always 560 bytes*/
    int kssize = 560;

    unsigned char *ks = calloc(kssize, sizeof(unsigned char));

    /* Iterate byte by byte through memory */
    for (i = ctx->from; i < ctx->filelen - kssize; i++) {
        /* Copy chunk of data from buffer, and expand with SERPENT key
         * schedule expansion */
        ks = memcpy(ks, &buffer[i], kssize);
        serpent_set_key(ks, ctx->keysize, ks);
        /* Compare result to the original buffer data */
        if (memcmp(ks, &buffer[i], kssize) == 0) {
            ctx->count++;
            printf("Found (probable) SERPENT key at offset %.8x:\n", i);
            print_hex_array(ks, ctx->keysize / 8, ctx->keysize / 8);
            printf("Expanded key:\n");
            print_hex_array(ks, kssize, 16);
        }
    }
}

void twofish_search(interrogate_context *ctx, unsigned char *buffer) {
    int i, firstrun, lastrun;
    /* Override user selected window size */
    ctx->wsize = 4096;

    /* Check that the input file can actually hold a full key schedule */
    size_t tfi_size = sizeof(twofish_tc); // Largest key schedule
    if (ctx->filelen < tfi_size) {
        fprintf(stderr, "Filesize too small to hold a TwoFish key.\n");
        return;
    }

    int run[TF_RUNS];
    firstrun = lastrun = 0;
```

```
    /* Check first window and initialize */
    i = ctx->from;
    runs(ctx, &buffer[i], run, TF_RUNS, &firstrun, &lastrun);
    if (is_mk_tab(run)) {
        validate_tf_ks(ctx, buffer, i);
    }

    /* Check each sequential window */
    for (; i < ctx->filelen; i++) {
        runs_opt(ctx, &buffer[i], run, TF_RUNS, &firstrun, &lastrun);
        if (is_mk_tab(run)) {
            validate_tf_ks(ctx, buffer, i);
        }
    }

}

/*
 * Deprecated. Old twofish key search method. Use twofish_search() instead.
 * This method will only work for truecrypt-like implementations.
 */
void twofish_search_old(interrogate_context *ctx, unsigned char *buffer) {
    twofish_tc *instance = malloc(sizeof(twofish_tc));
    int i;
    float entropy;

    /* Check that the input file can actually hold a full key schedule */
    size_t tfi_size = sizeof(twofish_tc);
    if (ctx->filelen < tfi_size) {
        fprintf(stderr, "Filesize too small to hold a TwoFish key.\n");
        return;
    }

    /* For each byte in memory, interpret it as the start of a
     * twofish_nstance struct, and check whether it has 2, 3 or 4 as the
     * twofish  key_len. If so, perform structural and statistical tests to
     * verify that it is a valid TWOFISH key schedule */
    for (i = ctx->from; i < ctx->filelen - tfi_size; i++) {
        instance = (twofish_tc *)&buffer[i];
        switch (instance->k_len) {
        case 2:
            /* Potential 128-bit key.
             * If key_len is 2, only the two leftmost s_keys are non-zero */
            if ((instance->s_key[2] == 0) && (instance->s_key[3] == 0)
                    && (instance->l_key[0] != 0)) {
                entropy = ent(ctx, (unsigned char *)instance->mk_tab,
                        sizeof(instance->mk_tab));
                /* The entropy of mk_tab is awlways maximum (8) */
                if (entropy == 8) {
                    /* Calculate entropy of the l_keys */
                    entropy = ent(ctx, (unsigned char *)instance->l_key,
                            sizeof(instance->l_key));
                    if ((entropy > 6) && (entropy < 7.2)) {
                        ctx->count++;
                        printf("Found (probable) TwoFish key at "
                            "offset %.8x:\n", i);
                        printf("Expanded key:\n");
                        print_hex_words((unsigned int *)instance,
                                tfi_size / 4, 4);
                    }
                }
            }
            break;
        case 3:
            /* Potential 198-bit key.
             * If key_len is 3, only the leftmost s_key is non-zero */
            if ((instance->s_key[3] == 0) && (instance->l_key[0] != 0)) {
                entropy = ent(ctx, (unsigned char *)instance->mk_tab,
                        sizeof(instance->mk_tab));
                /* The entropy of mk_tab is awlways maximum (8) */
                if (entropy == 8) {
                    /* Calculate entropy of the l_keys */
                    entropy = ent(ctx, (unsigned char *)instance->l_key,
                            sizeof(instance->l_key));
```

```
                    if ((entropy > 4)) {
                        ctx->count++;
                        printf("Found (probable) TwoFish key at "
                            "offset %.8x:\n", i);
                        printf("Expanded key:\n");
                        print_hex_words((unsigned int *)instance,
                                tfi_size / 4, 4);
                    }
                }
            }
            break;
        case 4:
            /* Potential 256-bit key */
            entropy = ent(ctx, (unsigned char *)instance->mk_tab,
                    sizeof(instance->mk_tab));
            if ((entropy == 8)) {
                /* Calculate entropy of the l_keys */
                entropy = ent(ctx, (unsigned char *)instance->l_key,
                        sizeof(instance->l_key));
                if ((entropy > 6) && (entropy < 7.2)) {
                    /* Calculate entropy of the l_keys */
                    entropy = ent(ctx, (unsigned char *)instance->s_key,
                                        sizeof(instance->s_key));
                    ctx->count++;
                    printf("Found (probable) TwoFish key at "
                        "offset %.8x:\n", i);
                    printf("Expanded key:\n");
                    print_hex_words((unsigned int *)instance,
                            tfi_size / 4, 4);
                }
            }
            break;
        }
    }
}

/* ----------------------------------------
 * Search functions for entropy-based search.
 * ----------------------------------------
 */

void search(interrogate_context *ctx, unsigned char *buffer) {
    int i, found, start, end;
    float entropy, cent;
    found = FALSE;
    entropy = cent = 0.0;
    start = ctx->from;

    //TODO: Change from continous sections to only windows of entropy
    for (i = ctx->from; i < ctx->filelen - ctx->wsize; i++) {
        /* Calculate entropy (if naivemode) or simply count unique bytes */
        entropy = (ctx->naivemode) ? ent(ctx, &buffer[i], ctx->wsize)
                : countbytes(ctx, &buffer[i]);
        /* Print value to file if the -p switch is set */
        if (ctx->output_fp != NULL)
            print_to_file(ctx->output_fp, entropy);

        if (entropy >= ctx->threshold) {
            if (!found) {
                start = i;
                ctx->count++;
                found = TRUE;
            }
            cent += entropy;
        } else {
            if (found) {
                end = i + ctx->wsize - 1; /* Ended at previous round */
                int bytes = end - start;
                float numblocks = (float) bytes / ctx->wsize;
                printblobinfo(start, end, bytes, numblocks, cent / (bytes
                        - ctx->wsize + 1));
                cent = 0;
                found = FALSE;
            }
```

```
        }
    }

    /* If found is true here, we found something in the last round, print
     * it */
    if (found) {
        end = i + ctx->wsize;
        int bytes = end - start;
        float numblocks = (float) bytes / ctx->wsize;
        printblobinfo(start, end, bytes, numblocks,
                cent / (bytes - ctx->wsize));
    }

}

void quicksearch(interrogate_context *ctx, unsigned char *buffer) {
    /* Move window over file and calculate entropy for each window
     * position */
    int i;
    float entropy = 0.0;
    int eof= FALSE;
    int found= FALSE;
    float cent = 0; /* Cumulative entropy */
    int start, end;
    start = i = ctx->from;
    int oldwsize = ctx->wsize;

    while (!eof) { /* Last round, make sure the window fits */
        if ((i >= ctx->filelen - ctx->wsize)) {
            eof = TRUE;
            ctx->wsize = ctx->filelen - i;
        }
        /* The end of the current search window */
        end = i + ctx->wsize;

        /* Calculate entropy (if naivemode) or simply count unique bytes */
        entropy = (ctx->naivemode) ? ent(ctx, &buffer[i], ctx->wsize)
                : countbytes(ctx, &buffer[i]);
        /* Print value to file if the -p switch is set */
        if (ctx->output_fp != NULL)
            print_to_file(ctx->output_fp, entropy);

        if (entropy >= ctx->threshold) {
            /* If found is false, the last block did not contain high
             * entropy. In that case, mark the start of a new block,
             * increment block counter and set fount to true */
            if (!found) {
                start = i;
                ctx->count++;
                found = TRUE;
            }

            /* Accumulate total entropy */
            cent += entropy;

            if (eof) { // If this is the last round, print it right away
                int bytes = end - start;
                float numblocks = (float) bytes / oldwsize;
                printblobinfo(start, end, bytes, numblocks,
                        cent / numblocks);
            }
        } else {
            /* If found is true, the last block examined contained high
             * entropy, but the current block did not. In that case
             * the entropy blob has reached its end after the previous
             * block, and we'll print its data. */
            if (found) {
                int prevend = end - ctx->wsize;
                int bytes = prevend - start;
                float numblocks = (float) bytes / oldwsize;
                printblobinfo(start, prevend, bytes, numblocks, cent
                        / numblocks);
                cent = 0;
                found = FALSE;
```

```
            }
        }
        i += ctx->wsize; // Increment counter, move wsize bytes each round
    }
    ctx->wsize = oldwsize; // Restore window size
}

/* -----------------------
 * Main program functions.
 * -----------------------
 */

/*
 * Prints usage and help info
 */
void help() {
    printf("Usage: interrogate [OPTION]... [FILE]...\n"
    "Search for cryptographic keys in the FILEs (memory dumps).\n"
    "\n"
    "  -a algorithm    search for keys of a certain type (algorithm).\n"
    "                    Valid parameters are aes, rsa, serpent or\n"
    "                    [tc-]twofish. Use the -k switch to specify AES "
    "                    key lengths (128, 198, or 256 bits). RSA keys are\n"
    "                    found independent of their length, while SERPENT\n"
    "                    and TWOFISH keys are required to be 256 bits.\n"
    "  -h                prints usage and help information (this message).\n"
    "  -i interval     only search within interval. Format of interval is\n"
    "                    from_offset:to_offset where the offset values\n"
    "                    are interpreted as hexadecimal values. Omitting\n"
    "                    one of the offsets will indicate the start or\n"
    "                    the end of the FILEs, respectively. Used with\n"
    "                    the -r switch, the interval will be interpreted\n"
    "                    as the virtual address space that are to be\n"
    "                    reconstructed.\n"
    "  -k keylength    length of key to be searched for (NB: in BITS)\n"
    "  -n                naive mode, calculates true entropy instead of\n"
    "                    counting unique bytes (which is the normal\n"
    "                    mode). This may be useful if you get bad quality\n"
    "                    results, but may yield some performance\n"
    "                    degradation.\n"
    "  -p filename     print entropy values for each window separated\n"
    "                    by newlines to file specified by filename. This\n"
    "                    may be used as input to plotting tools (gnuplot)\n"
    "                    WARNING: Slow and generates large files, one\n"
    "                    input byte maps to potentially six output bytes.\n"
    "  -q              quick mode, does not use overlapping windows. The\n"
    "                    larger the window size, the quicker. Use -w to\n"
    "                    specify window size.\n"
    "  -r CR3          reconstructs the virtual address space for the\n"
    "                    process at offset PDB. The PDB is the location of\n"
    "                    the page directory base, and can be found by\n"
    "                    scanning for EPROCESSes using PTfinder,\n"
    "                    Volatility or other similar tools. The\n"
    "                    regonstructed memory is written to file\n"
    "                    'pages', and are searched subsequently for\n"
    "                    keys. The -i option may be used to specify a\n"
    "                    virtual address space interval.\n"
    "  -t threshold    sets the entropy threshold (default = 7.0).\n"
    "  -w windowsize   sets the window size. Not compatible with the -a\n"
    "                    option.\n");
}

/*
 * Initializes the context of Interrogate.
 */
void initialize(interrogate_context *ctx) {
    ctx->keytype = NO_KEYTYPE;          /* No keytype by default */
    ctx->keysize = 0;                   /* Size of key to (in bits) */
    ctx->wsize = WINDOWSIZE;            /* Size of search window */
    ctx->nofs = NOFSYMBOLS;             /* Size of our alphabet */
    ctx->threshold = THRESHOLD;         /* Default entropy threshold */
    ctx->bitmode = FALSE;               /* Bit-mode is false by default */
    ctx->naivemode = FALSE;             /* Naive mode is false by default */
    ctx->quickmode = FALSE;             /* Quickmode turned off by default */
```

```
    ctx->interval = FALSE;              /* Interval turned off by default */
    ctx->verbose = FALSE;               /* Verbose mode is per def false */
    ctx->from = ctx->to = 0;            /* Interval is zero by default */
    ctx->cr3 = 0;                       /* Don't reconstruct (default) */
    ctx->filelen = 0;                   /* Zero file length */
    ctx->count = 0;                     /* Set key counter to zero */
}

/*
 * Main program, parse parameters and set context
 */
int main(int argc, char **argv) {
    int c;                                          /* Stores argument options */
    int i;                                          /* Counter */
    FILE *fp;                                       /* Pointer to input file */
    interrogate_context *ctx =
        malloc(sizeof(interrogate_context));        /* Program context */

    printf(
    "Interrogate  Copyright (C) 2008  Carsten Maartmann-Moe "
    "<carmaa@gmail.com>\n"
    "This program comes with ABSOLUTELY NO WARRANTY; for details use `-h'.\n"
    "This is free software, and you are welcome to redistribute it\n"
    "under certain conditions; see bundled file licence.txt for details.\n\n"
            );

    initialize(ctx);

    /* Parse arguments and set options, see help() method for explaination */
    while ((c = getopt(argc, argv, "a:hi:k:np:qr:t:vw:")) != -1) {
        switch (c) {
        case 'a':
            if (strncmp(optarg, "aes", 3) == 0) {
                ctx->keytype = AES;
            } else if (strncmp(optarg, "rsa", 3) == 0) {
                ctx->keytype = RSA;
            } else if (strncmp(optarg, "serpent", 7) == 0) {
                ctx->keytype = SERPENT;
                /* We only have support for 256-bit SERPENT keys */
                ctx->keysize = 256;
            } else if (strncmp(optarg, "twofish", 7) == 0) {
                ctx->keytype = TWOFISH;
                /* We only have support for 256-bit TWOFISH keys */
                ctx->keysize = 256;
            } else if (strncmp(optarg, "tc-twofish", 10) == 0) {
                ctx->keytype = TWOFISH_TC;
                /* We only have support for 256-bit Truecrypt TWOFISH keys */
                ctx->keysize = 256;
            } else {
                fprintf(stderr, "Invalid keytype.\n");
                help();
                exit(-1);
            }
            break;
        case 'h':
            help();
            exit(0);
        case 'i':
            ctx->interval = TRUE;
            /* Do ugly parsing of argument :-/ */
            char *to_ptr = strstr(optarg, ":"); // Find ':'
            *to_ptr = '\0'; // Replace with string terminator
            to_ptr++;
            /* Convert from hexadecimal ASCII */
            ctx->from = (int)strtol(optarg, (char**)NULL, 16);
            ctx->to = (int)strtol(to_ptr, (char **)NULL, 16);
            if (ctx->to < ctx->from && ctx->to != 0) {
                fprintf(stderr, "Error in interval, the start offset "
                        "is bigger than the end offset.\n");
                exit(-1);
            }
            break;
        case 'k':
            ctx->keysize = atoi(optarg);
```

```
            printf("Using key size: %i bits.\n", ctx->keysize);
            break;
        case 'n':
            ctx->naivemode = TRUE;
            printf("Using naive mode, searching for true entropy.\n");
            break;
        case 'p':
            ctx->output_fp = open_file(ctx, optarg, "w");
            break;
        case 'q':
            ctx->quickmode = TRUE;
            printf("Using quickmode.\n");
            break;
        case 'r':
            ctx->cr3 = (int)strtol(optarg, (char**)NULL, 16);
            break;
        case 't':
            ctx->threshold = atof(optarg);
            printf("Using entropy threshold: %f bits per symbol.\n",
                    ctx->threshold);
            break;
        case 'v':
            ctx->verbose = TRUE;
            printf("Verbose mode.\n");
            break;
        case 'w':
            ctx->wsize = atoi(optarg);
            printf("Using window size: %i bytes.\n", ctx->wsize);
            break;
        case '?':
            if (optopt == 'c' || optopt == 'w') {
                fprintf(stderr, "Option -%c requires an "
                "argument.\n", optopt);
            } else if (isprint(optopt)) {
                fprintf(stderr, "Unknown option `-%c'.\n",
                optopt);
            } else {
                fprintf(stderr, "Unknown option character "
                "`\\x%x'.\n", optopt);
            }
            return 1;
        default:
            exit(-1);
        }
    }
    /* Check that the windowsize is reasonable */
    if (ctx->naivemode && (ctx->wsize < (ctx->nofs / 2))) {
        printf("WARNING: You're using a windowsize smaller than half of the "
            "number of symbols together with naive mode, this might not "
            "yield a good result. Try dropping -n.\n");
    }
    /* Check that keytypes match supported key lengths */
    switch (ctx->keytype) {
    case AES:
        if (!(ctx->keysize == 128 ||
                ctx->keysize == 192 ||
                ctx->keysize == 256)) {
            fprintf(stderr, "A key size of 128, 192 or 256 bits are "
            "required for AES search.\n");
            exit(-1);
        }
        break;
    case SERPENT:
        if (!(ctx->keysize == 256)) {
            fprintf(stderr, "A key size of 256 bits are required for "
            "SERPENT search.\n");
            exit(-1);
        }
        break;
    case TWOFISH:
        if (!(ctx->keysize == 256)) {
            fprintf(stderr, "A key size of 256 bits are required for "
            "TWOFISH search.\n");
            exit(-1);
```

```
        }
        break;
    }

    if ((!ctx->naivemode)
            && (ctx->keytype == NO_KEYTYPE && ctx->threshold == 7)) {
        /* Set relaxed byte count threshold since the user didn't
         * specify one*/
        ctx->threshold = floor((ctx->wsize / NOFSYMBOLS) * ctx->threshold
                * BCMOD);
        printf("WARNING: No -t option specified, bytecount threshold was "
            "set to  %f. This may yield inaccurate results.\n",
            ctx->threshold);
    }

    /* The rest of the args are treated as files */
    if (optind < argc) {
        for (i = optind; i < argc; i++) {
            /* Check and open file for reading */
            fp = open_file(ctx, argv[i], "rb");
            printf("Using input file: %s.\n", argv[i]);
            if (ctx->interval) {
                /* Check if intervals are out of bounds */
                if (ctx->from < 0) {
                    ctx->from = 0;
                    printf("WARNING: Interval out of bounds, changed it "
                           "for you:\n");
                }
                /* If the upper bound is too big, set it to filelenght */
                if (ctx->to > ctx->filelen) {
                    ctx->to = ctx->filelen;
                    /* If the lower bound is too low, set it to zero */
                    if (ctx->to < ctx->from)
                        ctx->from = 0;
                    printf("WARNING: Interval out of bounds, changed it "
                           "for you:\n");
                }
                /* If no upper bound is given, set it to filelength */
                if (ctx->to == 0) {
                    ctx->to = ctx->filelen;
                }
                printf("Searching in interval 0x%08X - 0x%08X.\n",
                       ctx->from, ctx->to);
            }

            unsigned char *buffer =
                malloc(ctx->filelen * sizeof(unsigned char));
            buffer = read_file(ctx, fp);

            /* Reconstruct memory if the -r switch is on */
            if(ctx->cr3 != 0) {
                printf("Reconstructing virtual memory for process with PDB "
                        "at %08x, please stand by...\n", ctx->cr3);
                reconstruct(ctx, buffer);
                printf("Using recontructed virtual memory file "
                        "'pages' for search.\n");
                fp = open_file(ctx, "pages", "rb");
                buffer =
                    realloc(buffer, ctx->filelen * sizeof(unsigned char));
                buffer = read_file(ctx, fp);
            }

            /* Perform search */
            keysearch(ctx, buffer);

            /* Clean up */
            if (ctx->output_fp != NULL) {
                fclose(ctx->output_fp);
            }
            fclose(fp);
        }
        printf("\nA total of %li %s found.\n", ctx->count, (ctx->keytype
                == NO_KEYTYPE) ? "entropy blobs" : "keys");
        printf("Spent %li seconds of your day looking for the key.\n",
```

```
                clock() / CLOCKS_PER_SEC);
    } else {
        fprintf(stderr, "Missing input file.\n");
        help();
    }
    free(ctx);
    return 0;
}
```

A.3 stat.c

Listing A.3: stat.c

```
/* ==========================================================================
 * stat.c
 *
 * Statistcal functions used in Interrogate
 *
 * Author: Carsten Maartmann-Moe <carmaa@gmail.com>
 * ==========================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "interrogate.h"

#define LOG2OF10 3.32192809488736234787

int r[6] = {0, 0, 0, 0, 0, 0};

/*
 * Calculate log2
 */
double approxlog2(double x) {
    return LOG2OF10 * log10(x);
}

/*
 * Calculates entropy of char array, with length window size and 'nofs'
 * symbols.
 */
float ent(interrogate_context *ctx, unsigned char *buffer, int length) {
    int i, count = 0;   /* Counters */
    float entropy = 0.0;    /* The entropy */
    unsigned char c;    /* Char read from file buffer */
    int *ccount;        /* Bins for counting chars */
    float *p;       /* Bins for char probabilities */

    /* Reserve space. ccount is zeroed out, p is not (we're iterating through
     * p later anyways).*/
    ccount = (int *) calloc(ctx->nofs, sizeof(int));
    p = (float *) malloc(ctx->nofs * sizeof(float));

    /* Count occurrences of each char and the total count within window */
    while (count < length) {
        c = (unsigned char) *buffer++;
        ccount[c]++;
        count++;
    }

    /* Calculate probabilitiy of each char, and update entropy */
    for (i = 0; i < ctx->nofs; i++) {
        p[i] = ((float) ccount[i]) / length;
        if (p[i] > 0.0)
            entropy -= (float) p[i] * approxlog2(p[i]);
    }
    free(ccount);
    free(p);
    return entropy;
}

/*
 * Returns the minimum value of two ints
 */
int min(int a, int b) {
    return (a < b)? a : b;
}

/*
```

```
 * Checks if a byte in an array is set. The unsigned char is simply
 * the index in the array that has to be checked.
 */
int checkbyte(unsigned char index, int *array) {
    return array[index];
}

/*
 * Counts number of unique bytes within a non-overlapping window.
 */
int countbytes(interrogate_context *ctx, unsigned char *buffer) {
    int count = 0;      /* Window counter */
    int bytecount = 0;  /* The unique byte counter */
    int *ccount;        /* Bins for already discovered bytes */
    unsigned char c;    /* Char read from file buffer */

    ccount = (int *) calloc(ctx->nofs, sizeof(int));

    while ( count < ctx->wsize ) {
        c = (unsigned char) *buffer++;
        if (ccount[c] == 0) {
            ccount[c]++;
            bytecount++;
        }
        count++;
    }
    free(ccount);
    return bytecount;
}

/*
 * Count byte runs. A one-byte run is defined as two sequential bytes of
 * equal value. Thus, a six-byte run of 0x41 is actually seven sequential
 * 0x41s. All runs longer than 'run_length' are counted in the last bin, e.g.
 * as a 'run_length'-byte run. A call to this method is required to
 * initialize the optimized runs method 'runs_opt'.
 */
void runs(interrogate_context *ctx, unsigned char *buffer, int *runs_count,
          int run_length, int *firstrun, int *lastrun) {
    int i;
    int overflow = 0;
    unsigned char last = 0;
    int current_run = 0;
    memset(runs_count, 0, run_length * sizeof(int));
    for (i = 0; i < ctx->wsize; i++) {
        unsigned char c = buffer[i];
        /* Don't count the first char as a run */
        if (i != 0) {
            if (c == last) {
                if (current_run < run_length) {
                    /* Only decrement counter if such a bin exists */
                    if (current_run != 0)
                        runs_count[current_run - 1]--;
                    runs_count[current_run]++;
                    current_run++;
                } else {
                    overflow++;
                }
            } else {
                /* Check if the run went on from the start; if so save */
                if (i == current_run + overflow + 1) {
                    *firstrun = current_run;
                }
                /* Reset runs counters */
                current_run = overflow = 0;
            }
        }
        last = c;
    }
    /* Save if the last char was a part of a run */
    *lastrun = current_run;
}

/*
```

```c
 * Optimized 'runs' method. See runs(). Needs to be initialized by a call
 * to runs() before excecution; to count runs in the initial window, and
 * set lastrun and firstrun counters. The algorithm basically keeps track of
 * the runs in the ends of the buffer, and increments and decrements run
 * counts as needed. It is intended to work on a unsigned char buffer, and be
 * fed sub-buffers of this buffer in a sequential fashion. For example, a
 * call procedure like this will work:
 *
 * int *runs_count = {0, 0, 0, 0, 0, 0}; // Initalize array for storage
 * lastrun = firstrun = 0; // Initialize counters
 * runs(...); // Initialize by calling 'runs()' function
 * for (i = 0; i < buffersize; i++) {
 *     runs_opt(context, &buffer[i], runs_count, ...);
 * }
 *
 * This method has a significant performance gain compared to calling runs
 * sequentially, typically linear vs. exponential time complexity. For some
 * reason, this method is known to not work with gcc optimization e.g., no
 * -Ox options.
 */
void runs_opt(interrogate_context *ctx, unsigned char *buffer,
              int *runs_count, int run_length, int *firstrun, int *lastrun) {
    unsigned char *buf_ptr = buffer;
    int new_firstrun = 0;
    /* Count the new first run */
    while((*buf_ptr == *++buf_ptr) && new_firstrun < run_length) {
        new_firstrun++;
    }
    if (ctx->wsize < 2 * run_length) {
        fprintf(stderr, "A window size of at least two times the run "
                "length is required for this function to work.\n");
        exit(-1);
    }
    /* Since C indexes runs from 0 we need to subtract one from every
     * count to form indices in the runs_count table. If the new firs run
           * is its maximum, it implies that the counts should not be
           * decremented
     */
    if (*firstrun > 0 && !(new_firstrun == 6)) {
        /* Decrement bin count for the byte that "fell out" */
        runs_count[*firstrun - 1]--;
        /* Subract the byte that "fell out" of the buffer */
        (*firstrun)--;
        /* If there exists a bin for a smaller run, increment it */
        if (*firstrun != 0)
            runs_count[*firstrun - 1]++;
    } else {
        /* Count an eventual new run */
        *firstrun = new_firstrun;
    }
    /* Check if the last two chars in the buffer match */
    if (buffer[ctx->wsize - 2] == buffer[ctx->wsize - 1]) {
        /* Decrement the count for the previous run */
        if (*lastrun > 0)
            runs_count[*lastrun - 1]--;
        /* Increment lastrun if its less than max run length */
        if (*lastrun < run_length)
            (*lastrun)++;
        /* Increment bin for current count */
        runs_count[*lastrun - 1]++;
    } else {
        /* Reset lastrun if the two last chars doesn't match */
        *lastrun = 0;
    }
}
```

A.4 util.c

Listing A.4: util.c

```
/* ========================================================================
 * util.c
 *
 * Utility toolbox for Interrogate
 *
 * Author: Carsten Maartmann-Moe <carmaa@gmail.com>
 * ========================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "interrogate.h"

/*
 * Open file, return pointer
 */
FILE *open_file(interrogate_context *ctx, char *filename, char *mode) {
    struct stat st;         /* Stat struct for input file */
    FILE *fp;               /* Pointer to input file */
    if (stat(filename, &st) == -1) {
        perror("stat()");
        fprintf(stderr, "Failed to stat %s.\n", filename);
        exit(-1);
    } else {
        ctx->filelen = st.st_size;
    }

    fp = fopen(filename, mode);
    if (fp == NULL) {
        perror("fopen()");
        fprintf(stderr, "Failed to open %s.\n", filename);
        exit(-1);
    }
    return fp;
}

/*
 * Reads entire file into memory and returns buffer
 */
unsigned char *read_file(interrogate_context *ctx, FILE *fp) {
    unsigned char *buffer; /* Buffer (entire file) */

    /* Get the length of the file and rewind */
    fseek(fp, 0L, SEEK_END);
    ctx->filelen = ftell(fp);
    rewind(fp);

    /* Try to allocate enough memory for entire file. Should work for
     * large files if the system uses virtual memory. calloc()
     * initializes all bytes to 0, so we don't have to worry about
     * setting the NULL-terminator. */
    buffer = calloc(ctx->filelen + 1, sizeof(unsigned char));
    if (buffer == NULL) {
        fprintf(stderr, "Not enough memory to read entire file.\n");
        exit(1);
    }

    /* Read file into buffer */
    printf("Attempting to load entire file into memory, please stand "
           "by...\n");
    size_t res = fread(buffer, 1, ctx->filelen, fp);
    if (res != ctx->filelen) {
        fprintf(stderr, "Reading error.\n");
        exit(3);
    }
```

```
    return buffer;
}

/*
 * Prints info about entropy blobs
 */
void printblobinfo(int start, int end, int bytes, float wins, float ent) {
    printf(" %.8x - %.8x | %8i | %7.2f | %f \n",
           start, end, bytes, wins, ent);
}

/*
 * Prints raw data in hexadecimal form to stdout. Bytes are separated wiht
 * spaces, and linefeeds are inserted after 'column' bytes
 */
void print_hex_array(unsigned char *buffer, int length, int columns) {
    int i;
    for (i = 0; i < length; i++) {
        if ((i % columns) == 0)
            printf("\n");
        printf("%02x ", buffer[i]);
    }
    printf("\n\n");
}

/*
 * Prints raw data in hexadecimal, 32-bit word, little-endian form to stdout.
 * Words are separated with spaces, and linefeeds are inserted after
 * 'columns' words
 */
void print_hex_words(unsigned int *buffer, int length, int columns) {
    int i;
    for (i = 0; i < length; i++) {
        if ((i % columns) == 0)
            printf("\n");
        printf("%08x ", buffer[i]);
    }
    printf("\n\n");
}

/*
 * Windows getopt() :-/
 */
#ifdef _WIN32
static int optind = 1;

static int getopt(int argc, char *argv[], char *opts) {
    static char *opp = NULL;
    int o;

    while (opp == NULL) {
        if ((optind >= argc) || (*argv[optind] != '-')) {
            return -1;
        }
        opp = argv[optind] + 1;
        optind++;
        if (*opp == 0) {
            opp = NULL;
        }
    }
    o = *opp++;
    if (*opp == 0) {
        opp = NULL;
    }
    return strchr(opts, o) == NULL ? '?' : o;
}
#endif

void print_to_file(FILE *fp, float value) {
    char str[30];
    snprintf(str, 30, "%.4g", value);
    strncat(str, "\n", 1);
    fputs(str, fp);
}
```

```
unsigned getbits(unsigned x, int p, int n) {
    return (x >> (p + 1 - n)) & ~(~0 << n);
}

/*
 * Truncates of the nPrecision last digits of a float
 */
double format(double Value, int nPrecision) {
    char *buffer = malloc(128*sizeof(char));
    snprintf(buffer,127,"%0.*f",nPrecision,Value);
    double d = atof(buffer);
    free(buffer);
    return d;
}

/*
 * Checks if the runs lies within a relaxed set of heuristic values.
 */
int is_mk_tab(int *run) {
    return (run[0] < 520 &&
            run[0] > 485 &&
            run[1] == 0 &&
            run[2] <= 12 &&
            run[2] >= 1 &&
            run[3] == 0 &&
            run[4] == 0 &&
            run[5] <= 1 &&
            run[5] >= 0);
}

/*
 * Heuristic check for Twofish sub- and whitening keys
 */
int is_l_key(interrogate_context *ctx, unsigned int *l_key) {
    float entropy = ent(ctx, (unsigned char *)l_key, 160);
    return (entropy < 7.2 && entropy > 6.3);
}

/*
 * Heuristic check for Twofish S-box keys
 */
int is_s_key(interrogate_context *ctx, unsigned int *s_key) {
    float entropy = format(ent(ctx, (unsigned char *)s_key, 16), 4);
    return (entropy == 4.0000 ||
            entropy == 3.8750 ||
            entropy == 3.7500 ||
            entropy == 3.7028 ||
            entropy == 3.6250 ||
            entropy == 3.5778 ||
            entropy == 3.5000 ||
            entropy == 3.4528 ||
            entropy == 3.4056 ||
            entropy == 3.3750 ||
            entropy == 3.3278 ||
            entropy == 3.2806 ||
            entropy == 3.2744 ||
            entropy == 3.2500 ||
            entropy == 3.2028 ||
            entropy == 3.1556 ||
            entropy == 3.1494 ||
            entropy == 3.1250 ||
            entropy == 3.0778 ||
            entropy == 3.0306 ||
            entropy == 3.0244 ||
            ((entropy <= 3.0000) &&
             (entropy >= 2.0000)));
}

/*
 * Validates a Twofish key schedule by structural checkups. Prints info.
 */
void validate_tf_ks(interrogate_context *ctx, unsigned char *buffer,
                    int offset) {
```

```
float entropy;
/* Try each of the different structs, and return the first match */

/* Truecrypt */
int tc_offs = offset - (44 * sizeof(unsigned int));
if (tc_offs >= 0) {
    twofish_tc *tc = (twofish_tc *) (buffer + tc_offs);
    entropy = ent(ctx, (unsigned char *)tc->mk_tab,
                  sizeof(tc->mk_tab));
    if (entropy == 8 && tc->k_len == 4) {
        if (is_l_key(ctx, tc->l_key)) {
            if(is_s_key(ctx, tc->s_key)) {
                printf("Truecrypt Twofish key found at %08x. "
                       "Expanded key:\n", tc_offs);
                printf("Key words:");
                print_hex_words((unsigned int *)tc->l_key,
                                (sizeof(tc->l_key)) / 4, 4);
                printf("S-box keys:");
                print_hex_words((unsigned int *)tc->s_key,
                                sizeof(tc->s_key) / 4, 4);
                printf("S-box array:");
                print_hex_words((unsigned int *)tc->mk_tab,
                                sizeof(tc->mk_tab) / 4, 4);
                printf("Key length:");
                print_hex_words(&tc->k_len,
                                sizeof(tc->k_len) / 4, 4);
                ctx->count++;
            }
        }
    }
}

/* Optimized */
int opt_offs = offset - (41 * sizeof(unsigned int));
if (opt_offs >= 0) {
    twofish_opt *tc4 = (twofish_opt *) (buffer + opt_offs);
    entropy = ent(ctx, (unsigned char *)tc4->QF,
                  sizeof(tc4->QF));
    if (entropy == 8 && (tc4->k_len == 0 || tc4->k_len == 1) ) {
        if (is_l_key(ctx, tc4->K)) {
            printf("Twofish key found at %08x. Expanded key:\n\n",
                   opt_offs);
            printf("Key words:");
            print_hex_words((unsigned int *)tc4->K,
                            (sizeof(tc4->K)) / 4, 4);
            printf("S-box array:");
            print_hex_words((unsigned int *)tc4->QF,
                            (sizeof(tc4->QF)) / 4, 4);
            ctx->count++;
        }
    }
}

/* GPG/Linux and SSH */
twofish_gpg *tc2 = (twofish_gpg *) (buffer + offset);
entropy = ent(ctx, (unsigned char *)tc2->s,
              sizeof(tc2->s));
if (entropy == 8) {
    if (is_l_key(ctx, tc2->w)) {
        printf("GPG or SSH Twofish key found at %08x. Expanded key:\n",
               offset);
        printf("Key words:");
        print_hex_words((unsigned int *)tc2->w,
                        (sizeof(tc2->w) + sizeof(tc2->k)) / 4, 4);
        printf("S-box array:");
        print_hex_words((unsigned int *)tc2->s,
                        (sizeof(tc2->s)) / 4, 4);
        ctx->count++;
    }
}

/* Nettle */
int nettle_offs = offset - (40 * sizeof(unsigned int));
if (nettle_offs >= 0) {
```

```
        twofish_nettle *tc3 = (twofish_nettle *) (buffer + nettle_offs);
        entropy = ent(ctx, (unsigned char *)tc3->s,
                      sizeof(tc3->s));
        if (entropy == 8) {
            if (is_l_key(ctx, tc3->k)) {
                printf("Nettle Twofish key found at %08x. Expanded key:\n\n",
                       nettle_offs);
                printf("Key words:");
                print_hex_words((unsigned int *)tc3->k,
                                (sizeof(tc3->k)) / 4, 4);
                printf("S-box array:");
                print_hex_words((unsigned int *)tc3->s,
                                (sizeof(tc3->s)) / 4, 4);
                ctx->count++;
            }
        }
    }

}
```

A.5 virtmem.c

Listing A.5: virtmem.c

```
/* ========================================================================
 * virtmem.c
 *
 * Utility to reconstruct virtual memory from the Nonpaged Pool of a
 * process. Part of Interrogate
 *
 * Author: Carsten Maartmann-Moe <carmaa@gmail.com>
 * ========================================================================
 */
#include <stdio.h>
#include <stdlib.h>

#include "interrogate.h"

/* Iterate through the virtual addresses in the Nonpaged Pool virtual
 * address space and fetch pages from the physical memory, using the
 * CR3 address as Page Directory base.
 */
void reconstruct(interrogate_context *ctx, unsigned char *buffer) {
    pte *pd, *pt;                      /* Page directory and table pointers */
    pte pd_entry, pt_entry;            /* Page directory and table entries */
    virtual_address *addr;             /* Virtual address */
    FILE *fp = fopen("pages", "wb");   /* Output file */
    unsigned int *frames;              /* Fetched page frame numbers */
    unsigned long this_pagesize;       /* The current pagesize (large page) */
    unsigned char *page;               /* Current page */
    unsigned int i, last_i, l_pc, pc;  /* (Page) counters */
    unsigned int lim_low;              /* Lower virtual address space bound */
    unsigned int lim_high;             /* Upper virtual address space bound */

    pd = malloc(sizeof(pte) * 1024);
    pt = malloc(sizeof(pte) * 1024);
    addr = malloc(sizeof(virtual_address));
    long memorysize = ctx->filelen;
    l_pc = pc = last_i = 0;

    /* Assume standard pagesize */
    int pagesize = 4096;

    /* Allocate and zero out memory for already fetched pages db */
    frames = calloc(memorysize / pagesize, sizeof(unsigned int));

    /* The page directory is located at the offset pointed to by CR3 */
    pd = (pte *)&buffer[ctx->cr3];

    if (ctx->interval) {
        lim_low = ctx->from;
        lim_high = ctx->to;
        ctx->interval = FALSE; // To prevent interval-search in main
    } else {
        /* A bit more dirty; use the whole virtual address space :-/ */
        lim_low = 0x00000000;
        lim_high = 0xffffffff;
    }
    printf("Reconstructing virtual memory from %08x to %08x. To change "
           "this, use the -i switch.\n", lim_low, lim_high);

    /* Large pages are only available with physical memory size > 255 MB */
    int large_pages = (memorysize > (255 * 1024));
    if (large_pages) {
        page = malloc(pagesize * 1024 * sizeof(unsigned char)); // 4 MB pages
    } else {
        page = malloc(pagesize * sizeof(unsigned char)); // 4 KB pages
    }

    for (i = lim_low; i < lim_high; i += pagesize) {
        /* Break if 'i' wraps around e.g. integer overflow */
        if (i < last_i)
            break;
```

```
        addr = (virtual_address *)&i;
        pd_entry = pd[addr->pd_index];
        /* Skip NULL entries */
        if (!*(unsigned int *)&pd_entry)
            continue;

        /* The target page table is found via the pfn of the pde */
        unsigned long pde_offset = pd_entry.pfn * pagesize;
        /* Check that the page is in memory, and that it is within bounds */
        if ((pde_offset < memorysize) &&
                pd_entry.valid) {
            pt = (pte *)&buffer[pde_offset];
            if (!pt)
                continue; // Null pointer
            pt_entry = pt[addr->pt_index];
            /* Skip NULL entries */
            if (!*(unsigned int *)&pt_entry)
                continue;
        }

        unsigned long pte_offset = pt_entry.pfn * pagesize;
        /* Check that the page is in memory, and that it is within bounds */
        if ((pte_offset < memorysize) &&
                pt_entry.valid) {
            if (!frames[pt_entry.pfn]) { // If the page (frame) is new
                /* Mark page as found, and fetch from buffer */
                frames[pt_entry.pfn] = 1;
                page = &buffer[pte_offset];

                if (ctx->verbose) {
                    print_pte(addr, pd, &pd_entry, pt, &pt_entry, page);
                }

                /* Set proper pagesize for current page */
                if (pt_entry.large_page && large_pages) {
                    l_pc++;
                    this_pagesize = pagesize * 1024;
                } else {
                    pc++;
                    this_pagesize = pagesize;
                }

                /* Place each page fetched sequentially in a new file */
                fwrite(page, sizeof(unsigned char), this_pagesize, fp);
            }
        }
        last_i = i; // Update the last value of 'i'
    }
    printf("Wrote %i pages to disk, %i normal and %i large, a total of "
           "%.2f MB.\n", l_pc + pc, pc, l_pc,
           ((double)ftell(fp) / (1024*1024)));
    fclose(fp);
}

void print_pte(virtual_address *addr, pte *pd, pte *pde, pte *pt, pte *pte,
               unsigned char *page) {
    printf( "Vitual address: %08x\n"
            "PD index:       %08x -> Byte offset:       %08x\n"
            "PDE value:      %08x -> Page frame number: %08x\n"
            "PT index:       %08x -> Byte offset:       %08x\n"
            "PTE value:      %08x -> Page frame number: %08x\n"
            "Flags:          %c%c%c%c%c%c%c%c%c\n"
            "First 16 bytes of page: ",
            *(unsigned int *)addr, addr->pd_index, addr->pd_index * 4,
            *(unsigned int *)&pde, pde->pfn,
            addr->pt_index, addr->pt_index * 4,
            *(unsigned int*)&pte, pte->pfn,
            (pte->copy_on_write)?'C':'-', (pte->global)?'G':'-',
            (pte->large_page)?'L':'-', (pte->dirty)?'D':'-',
            (pte->accessed)?'A':'-', (pte->cache_disabled)?'N':'-',
            (pte->write_through)?'T':'-', (pte->owner)?'U':'K',
            (pte->write)?'W':'R', (pte->valid)?'V':'-'
          );
```

```
    /* Print first 16 bytes of page */
    print_hex_array(page, 16, 16);
}
```

A.6 rsa.c

Listing A.6: rsa.c

```
/* ==========================================================================
 * rsa.c
 *
 * RSA-specific methods for Interrogate. Parses DER-encoded blobs and ouputs
 * to file in the format privkey-00x.der
 *
 * Author: Carsten Maartmann-Moe <carmaa@gmail.com>
 * ==========================================================================
 */

#include <stdio.h>
#include <stdlib.h>

#include "interrogate.h"

/* Perform basic structural check on possible DER-encoded private key.
 * Returns 0 if invalid, and the length of the DER blob if it is valid. Also
 * prints some info about the key.
 */
int parse_der(unsigned char *buffer, int offset) {
    if (buffer[offset + 4] == 0x02 &&
            buffer[offset + 5] == 0x01 &&
            buffer[offset + 6] == 0x00 &&
            buffer[offset + 7] == 0x02) {
        int length = (buffer[offset+2] << 8) |
                     (unsigned char) buffer[offset+3];
        int end = 4 + length;
        int pub_exp_field_length = 0;
        int modlength, asn1length = (unsigned char) buffer[offset + 8];
        if ((asn1length & 0x80) == 0) {
            modlength = asn1length;
            pub_exp_field_length = 1;
        } else {
            int numbytes = asn1length & 0x7F;
            if (numbytes <= 8) {
                int i;
                pub_exp_field_length = 1 + numbytes;
                modlength = (unsigned char) buffer[offset + 9];
                for (i = 1; i < numbytes; i++) {
                    modlength = (modlength << 8) |
                                (unsigned char) buffer[offset + 9 + i];
                }
            } else {
                printf("Found modulus length > 64 bits, this is not "
                       "supported.");
                return 0;
            }
        }
        int pub_exp_offset = offset + 8 + pub_exp_field_length + modlength;
        int pub_exp = 0;
        if (buffer[pub_exp_offset] == 0x02) {
            if (buffer[pub_exp_offset + 1] == 0x01 &&
                    buffer[pub_exp_offset + 2] == 0x01) {
                pub_exp = 1;
            } else if (buffer[pub_exp_offset + 1] == 0x03 &&
                       buffer[pub_exp_offset + 2] == 0x01 &&
                       buffer[pub_exp_offset + 3] == 0x00 &&
                       buffer[pub_exp_offset + 4] == 0x01) {
                pub_exp = 65537;
            } else {
                printf("Could not find public exponent, not a valid "
                       "key.\n");
                return 0;
            }
        }
        if (pub_exp != 0) {
            printf("%08x: Key: %i bits, public exponent %i.\n", offset,
                   (modlength - 1) * 8, pub_exp);
            return end;
```

```
        } else {
            return 0;
        }
    } else {
#if DEGUG
        printf("Invalid key found.");
#endif

        return 0;
    }
}
/*
 * Output DER information at offset 'offs'.
 */
void output_der(unsigned char *buffer, int offs, size_t size, long *count) {
    char filename[15];
    sprintf(filename, "privkey-%02li.der", *count);
    FILE *fp = fopen(filename, "wb");
    if (fp == NULL) {
        perror("fopen()");
        fprintf(stderr, "Failed to open %s.\n", filename);
        exit(-1);
    } else {
        fwrite(&buffer[offs], 1, size, fp);
        printf("Wrote key to file %s.\n", filename);
    }

    fclose(fp);
}
```

A.7 aes.c

Listing A.7: aes.c

```
/* ========================================================================
 * aes.c
 *
 * AES key schedule implementation for Interrogate
 *
 * Code by Sam Trenholme (http://www.samiam.org/rijndael.html)
 *
 * Errors corrected and code modified for use in Interrogate by
 * Carsten Maartmann-Moe <carmaa@gmail.com>
 * ========================================================================
 */

#include <stdio.h>

#include "interrogate.h"

/* Log table using 0xe5 (229) as the generator */
unsigned char ltable[256] = {
    0x00, 0xff, 0xc8, 0x08, 0x91, 0x10, 0xd0, 0x36,
    0x5a, 0x3e, 0xd8, 0x43, 0x99, 0x77, 0xfe, 0x18,
    0x23, 0x20, 0x07, 0x70, 0xa1, 0x6c, 0x0c, 0x7f,
    0x62, 0x8b, 0x40, 0x46, 0xc7, 0x4b, 0xe0, 0x0e,
    0xeb, 0x16, 0xe8, 0xad, 0xcf, 0xcd, 0x39, 0x53,
    0x6a, 0x27, 0x35, 0x93, 0xd4, 0x4e, 0x48, 0xc3,
    0x2b, 0x79, 0x54, 0x28, 0x09, 0x78, 0x0f, 0x21,
    0x90, 0x87, 0x14, 0x2a, 0xa9, 0x9c, 0xd6, 0x74,
    0xb4, 0x7c, 0xde, 0xed, 0xb1, 0x86, 0x76, 0xa4,
    0x98, 0xe2, 0x96, 0x8f, 0x02, 0x32, 0x1c, 0xc1,
    0x33, 0xee, 0xef, 0x81, 0xfd, 0x30, 0x5c, 0x13,
    0x9d, 0x29, 0x17, 0xc4, 0x11, 0x44, 0x8c, 0x80,
    0xf3, 0x73, 0x42, 0x1e, 0x1d, 0xb5, 0xf0, 0x12,
    0xd1, 0x5b, 0x41, 0xa2, 0xd7, 0x2c, 0xe9, 0xd5,
    0x59, 0xcb, 0x50, 0xa8, 0xdc, 0xfc, 0xf2, 0x56,
    0x72, 0xa6, 0x65, 0x2f, 0x9f, 0x9b, 0x3d, 0xba,
    0x7d, 0xc2, 0x45, 0x82, 0xa7, 0x57, 0xb6, 0xa3,
    0x7a, 0x75, 0x4f, 0xae, 0x3f, 0x37, 0x6d, 0x47,
    0x61, 0xbe, 0xab, 0xd3, 0x5f, 0xb0, 0x58, 0xaf,
    0xca, 0x5e, 0xfa, 0x85, 0xe4, 0x4d, 0x8a, 0x05,
    0xfb, 0x60, 0xb7, 0x7b, 0xb8, 0x26, 0x4a, 0x67,
    0xc6, 0x1a, 0xf8, 0x69, 0x25, 0xb3, 0xdb, 0xbd,
    0x66, 0xdd, 0xf1, 0xd2, 0xdf, 0x03, 0x8d, 0x34,
    0xd9, 0x92, 0x0d, 0x63, 0x55, 0xaa, 0x49, 0xec,
    0xbc, 0x95, 0x3c, 0x84, 0x0b, 0xf5, 0xe6, 0xe7,
    0xe5, 0xac, 0x7e, 0x6e, 0xb9, 0xf9, 0xda, 0x8e,
    0x9a, 0xc9, 0x24, 0xe1, 0x0a, 0x15, 0x6b, 0x3a,
    0xa0, 0x51, 0xf4, 0xea, 0xb2, 0x97, 0x9e, 0x5d,
    0x22, 0x88, 0x94, 0xce, 0x19, 0x01, 0x71, 0x4c,
    0xa5, 0xe3, 0xc5, 0x31, 0xbb, 0xcc, 0x1f, 0x2d,
    0x3b, 0x52, 0x6f, 0xf6, 0x2e, 0x89, 0xf7, 0xc0,
    0x68, 0x1b, 0x64, 0x04, 0x06, 0xbf, 0x83, 0x38 };

/* Anti-log table: */
unsigned char atable[256] = {
    0x01, 0xe5, 0x4c, 0xb5, 0xfb, 0x9f, 0xfc, 0x12,
    0x03, 0x34, 0xd4, 0xc4, 0x16, 0xba, 0x1f, 0x36,
    0x05, 0x5c, 0x67, 0x57, 0x3a, 0xd5, 0x21, 0x5a,
    0x0f, 0xe4, 0xa9, 0xf9, 0x4e, 0x64, 0x63, 0xee,
    0x11, 0x37, 0xe0, 0x10, 0xd2, 0xac, 0xa5, 0x29,
    0x33, 0x59, 0x3b, 0x30, 0x6d, 0xef, 0xf4, 0x7b,
    0x55, 0xeb, 0x4d, 0x50, 0xb7, 0x2a, 0x07, 0x8d,
    0xff, 0x26, 0xd7, 0xf0, 0xc2, 0x7e, 0x09, 0x8c,
    0x1a, 0x6a, 0x62, 0x0b, 0x5d, 0x82, 0x1b, 0x8f,
    0x2e, 0xbe, 0xa6, 0x1d, 0xe7, 0x9d, 0x2d, 0x8a,
    0x72, 0xd9, 0xf1, 0x27, 0x32, 0xbc, 0x77, 0x85,
    0x96, 0x70, 0x08, 0x69, 0x56, 0xdf, 0x99, 0x94,
    0xa1, 0x90, 0x18, 0xbb, 0xfa, 0x7a, 0xb0, 0xa7,
    0xf8, 0xab, 0x28, 0xd6, 0x15, 0x8e, 0xcb, 0xf2,
    0x13, 0xe6, 0x78, 0x61, 0x3f, 0x89, 0x46, 0x0d,
    0x35, 0x31, 0x88, 0xa3, 0x41, 0x80, 0xca, 0x17,
```

```
    0x5f, 0x53, 0x83, 0xfe, 0xc3, 0x9b, 0x45, 0x39,
    0xe1, 0xf5, 0x9e, 0x19, 0x5e, 0xb6, 0xcf, 0x4b,
    0x38, 0x04, 0xb9, 0x2b, 0xe2, 0xc1, 0x4a, 0xdd,
    0x48, 0x0c, 0xd0, 0x7d, 0x3d, 0x58, 0xde, 0x7c,
    0xd8, 0x14, 0x6b, 0x87, 0x47, 0xe8, 0x79, 0x84,
    0x73, 0x3c, 0xbd, 0x92, 0xc9, 0x23, 0x8b, 0x97,
    0x95, 0x44, 0xdc, 0xad, 0x40, 0x65, 0x86, 0xa2,
    0xa4, 0xcc, 0x7f, 0xec, 0xc0, 0xaf, 0x91, 0xfd,
    0xf7, 0x4f, 0x81, 0x2f, 0x5b, 0xea, 0xa8, 0x1c,
    0x02, 0xd1, 0x98, 0x71, 0xed, 0x25, 0xe3, 0x24,
    0x06, 0x68, 0xb3, 0x93, 0x2c, 0x6f, 0x3e, 0x6c,
    0x0a, 0xb8, 0xce, 0xae, 0x74, 0xb1, 0x42, 0xb4,
    0x1e, 0xd3, 0x49, 0xe9, 0x9c, 0xc8, 0xc6, 0xc7,
    0x22, 0x6e, 0xdb, 0x20, 0xbf, 0x43, 0x51, 0x52,
    0x66, 0xb2, 0x76, 0x60, 0xda, 0xc5, 0xf3, 0xf6,
    0xaa, 0xcd, 0x9a, 0xa0, 0x75, 0x54, 0x0e, 0x01 };

/* Circular rotate */
void rotate(unsigned char *in) {
    unsigned char a,c;
    a = in[0];
    for(c=0;c<3;c++)
        in[c] = in[c + 1];
    in[3] = a;
    return;
}

/* Calculate the rcon used in key expansion */
unsigned char rcon(unsigned char in) {
    unsigned char c=1;
    if(in == 0)
        return 0;
    while(in != 1) {
        c = gmul(c,2);
        in--;
    }
    return c;
}

/* Galois field multiplication */
unsigned char gmul(unsigned char a, unsigned char b) {
    int s;
    int q;
    int z = 0;
    s = ltable[a] + ltable[b];
    s %= 255;
    /* Get the antilog */
    s = atable[s];
    /* Now, we have some fancy code that returns 0 if either
     a or b are zero; we write the code this way so that the
     code will (hopefully) run at a constant speed in order to
     minimize the risk of timing attacks */
    q = s;
    if(a == 0) {
        s = z;
    } else {
        s = q;
    }
    if(b == 0) {
        s = z;
    } else {
        q = z;
    }
    return s;
}

/* Inverse Galois field multiplication */
unsigned char gmul_inverse(unsigned char in) {
    /* 0 is self inverting */
    if(in == 0)
        return 0;
    else
        return atable[(255 - ltable[in])];
}
```

```
/* Calculate the s-box for a given number */
unsigned char sbox(unsigned char in) {
    unsigned char c, s, x;
    s = x = gmul_inverse(in);
    for(c = 0; c < 4; c++) {
        /* One bit circular rotate to the left */
        s = (s << 1) | (s >> 7);
        /* xor with x */
        x ^= s;
    }
    x ^= 99; /* 0x63 */
    return x;
}

/* This is the core key expansion, which, given a 4-byte value,
 * does some scrambling */
void schedule_core(unsigned char *in, unsigned char i) {
    unsigned char a;
    /* Rotate the input 8 bits to the left */
    rotate(in);
    /* Apply Rijndael's s-box on all 4 bytes */
    for(a = 0; a < 4; a++)
        in[a] = sbox(in[a]);
    /* On just the first byte, add 2^i to the byte */
    in[0] ^= rcon(i);
}

/* Key expansion function for 128-bit keys */
void expand_key(unsigned char *in) {
    unsigned char t[4];
    /* c is 16 because the first sub-key is the user-supplied key */
    unsigned char c = 16;
    unsigned char i = 1;
    unsigned char a;

    /* We need 11 sets of sixteen bytes each for 128-bit mode */
    while(c < 176) {
        /* Copy the temporary variable over from the last 4-byte
         * block */
        for(a = 0; a < 4; a++)
            t[a] = in[a + c - 4];
        /* Every four blocks (of four bytes),
         * do a complex calculation */
        if(c % 16 == 0) {
            schedule_core(t,i);
            i++;
        }
        for(a = 0; a < 4; a++) {
            in[c] = in[c - 16] ^ t[a];
            c++;
        }
    }
}

/* Key expansion function for 192-bit keys */
void expand_key_192(unsigned char *in) {
    unsigned char t[4];
    unsigned char c = 24;
    unsigned char i = 1;
    unsigned char a;
    while(c < 208) {
        /* Copy the temporary variable over */
        for(a = 0; a < 4; a++)
            t[a] = in[a + c - 4];
        /* Every six sets, do a complex calculation */
        if(c % 24 == 0) {
            schedule_core(t,i);
            i++;
        }
        for(a = 0; a < 4; a++) {
            in[c] = in[c - 24] ^ t[a];
            c++;
        }
```

```
    }
}

/* Key expansion function for 256-bit keys */
void expand_key_256(unsigned char *in) {
    unsigned char t[4];
    unsigned char c = 32;
    unsigned char i = 1;
    unsigned char a;
    while(c < 240) {
        /* Copy the temporary variable over */
        for(a = 0; a < 4; a++)
            t[a] = in[a + c - 4];
        /* Every eight sets, do a complex calculation */
        if(c % 32 == 0) {
            schedule_core(t,i);
            i++;
        }
        /* For 256-bit keys, we add an extra sbox to the
         * calculation */
        if(c % 32 == 16) {
            for(a = 0; a < 4; a++)
                t[a] = sbox(t[a]);
        }
        for(a = 0; a < 4; a++) {
            in[c] = in[c - 32] ^ t[a];
            c++;
        }
    }
}
```

A.8 serpent.c

Listing A.8: serpent.c

```
/* ========================================================================
 * serpent.c
 *
 * Serpent key schedule implementation for Interrogate
 *
 * Adapted from serpent.cpp -- written and placed in the public domain by Wei
 * Dai. Interrogate version by Carsten Maartmann-Moe <carmaa@gmail.com>
 * ========================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include "interrogate.h"

/* -------
 * S-boxes
 * -------
 */
static void S0f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r3 ^= *r0;
    *r4 = *r1;
    *r1 &= *r3;
    *r4 ^= *r2;
    *r1 ^= *r0;
    *r0 |= *r3;
    *r0 ^= *r4;
    *r4 ^= *r3;
    *r3 ^= *r2;
    *r2 |= *r1;
    *r2 ^= *r4;
    *r4 = ~*r4;
    *r4 |= *r1;
    *r1 ^= *r3;
    *r1 ^= *r4;
    *r3 |= *r0;
    *r1 ^= *r3;
    *r4 ^= *r3;
}

static void S1f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r0 = ~*r0;
    *r2 = ~*r2;
    *r4 = *r0;
    *r0 &= *r1;
    *r2 ^= *r0;
    *r0 |= *r3;
    *r3 ^= *r2;
    *r1 ^= *r0;
    *r0 ^= *r4;
    *r4 |= *r1;
    *r1 ^= *r3;
    *r2 |= *r0;
    *r2 &= *r4;
    *r0 ^= *r1;
    *r1 &= *r2;
    *r1 ^= *r0;
    *r0 &= *r2;
    *r0 ^= *r4;
}

static void S2f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r4 = *r0;
    *r0 &= *r2;
    *r0 ^= *r3;
    *r2 ^= *r1;
    *r2 ^= *r0;
```

```
    *r3 |= *r4;
    *r3 ^= *r1;
    *r4 ^= *r2;
    *r1 = *r3;
    *r3 |= *r4;
    *r3 ^= *r0;
    *r0 &= *r1;
    *r4 ^= *r0;
    *r1 ^= *r3;
    *r1 ^= *r4;
    *r4 = ~*r4;
}

static void S3f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r4 = *r0;
    *r0 |= *r3;
    *r3 ^= *r1;
    *r1 &= *r4;
    *r4 ^= *r2;
    *r2 ^= *r3;
    *r3 &= *r0;
    *r4 |= *r1;
    *r3 ^= *r4;
    *r0 ^= *r1;
    *r4 &= *r0;
    *r1 ^= *r3;
    *r4 ^= *r2;
    *r1 |= *r0;
    *r1 ^= *r2;
    *r0 ^= *r3;
    *r2 = *r1;
    *r1 |= *r3;
    *r1 ^= *r0;
}

static void S4f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r1 ^= *r3;
    *r3 = ~*r3;
    *r2 ^= *r3;
    *r3 ^= *r0;
    *r4 = *r1;
    *r1 &= *r3;
    *r1 ^= *r2;
    *r4 ^= *r3;
    *r0 ^= *r4;
    *r2 &= *r4;
    *r2 ^= *r0;
    *r0 &= *r1;
    *r3 ^= *r0;
    *r4 |= *r1;
    *r4 ^= *r0;
    *r0 |= *r3;
    *r0 ^= *r2;
    *r2 &= *r3;
    *r0 = ~*r0;
    *r4 ^= *r2;
}

static void S5f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r0 ^= *r1;
    *r1 ^= *r3;
    *r3 = ~*r3;
    *r4 = *r1;
    *r1 &= *r0;
    *r2 ^= *r3;
    *r1 ^= *r2;
    *r2 |= *r4;
    *r4 ^= *r3;
    *r3 &= *r1;
    *r3 ^= *r0;
    *r4 ^= *r1;
```

```
    *r4 ^= *r2;
    *r2 ^= *r0;
    *r0 &= *r3;
    *r2 = ~*r2;
    *r0 ^= *r4;
    *r4 |= *r3;
    *r2 ^= *r4;
}

static void S6f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r2 = ~*r2;
    *r4 = *r3;
    *r3 &= *r0;
    *r0 ^= *r4;
    *r3 ^= *r2;
    *r2 |= *r4;
    *r1 ^= *r3;
    *r2 ^= *r0;
    *r0 |= *r1;
    *r2 ^= *r1;
    *r4 ^= *r0;
    *r0 |= *r3;
    *r0 ^= *r2;
    *r4 ^= *r3;
    *r4 ^= *r0;
    *r3 = ~*r3;
    *r2 &= *r4;
    *r2 ^= *r3;
}

static void S7f (unsigned int *r0, unsigned int *r1, unsigned int *r2,
                 unsigned int *r3, unsigned int *r4) {
    *r4 = *r2;
    *r2 &= *r1;
    *r2 ^= *r3;
    *r3 &= *r1;
    *r4 ^= *r2;
    *r2 ^= *r1;
    *r1 ^= *r0;
    *r0 |= *r4;
    *r0 ^= *r2;
    *r3 ^= *r1;
    *r2 ^= *r3;
    *r3 &= *r0;
    *r3 ^= *r4;
    *r4 ^= *r2;
    *r2 &= *r0;
    *r4 = ~*r4;
    *r2 ^= *r4;
    *r4 &= *r0;
    *r1 ^= *r3;
    *r4 ^= *r1;
}

static void LKf (unsigned int *k, unsigned int r, unsigned int *a,
                 unsigned int *b, unsigned int *c, unsigned int *d) {
    *a = k[r];
    *b = k[r + 1];
    *c = k[r + 2];
    *d = k[r + 3];
}

static void SKf (unsigned int *k, unsigned int r, unsigned int *a,
                 unsigned int *b, unsigned int *c, unsigned int *d) {
    k[r + 4] = *a;
    k[r + 5] = *b;
    k[r + 6] = *c;
    k[r + 7] = *d;
}

unsigned int LE32 (unsigned int x) {
    unsigned int n = (unsigned char) x;
    n <<= 8;
```

```
    n |= (unsigned char) (x >> 8);
    n <<= 8;
    n |= (unsigned char) (x >> 16);
    return (n << 8) | (unsigned char) (x >> 24);
}

/*
 * Sets the Serpent key schedule. Input: User supplied key, keysize in bytes,
 *  pointer to the key schedule storage.
 */
void serpent_set_key(const unsigned char userKey[], int keylen,
                     unsigned char *ks) {
    unsigned int a,b,c,d,e;
    unsigned int *k = (unsigned int *)ks;
    unsigned int t;
    int i;

    for (i = 0; i < keylen / (int)sizeof(int); i++)
        k[i] = ((unsigned int*)userKey)[i];

    if (keylen < 32)
        k[keylen/4] |= (unsigned int)1 << ((keylen%4)*8);

    k += 8;
    t = k[-1];
    for (i = 0; i < 132; ++i)
        k[i] = t = rotlFixed(k[i-8] ^ k[i-5] ^ k[i-3] ^ t ^ 0x9e3779b9 ^ i,
                             11);
    k -= 20;

    for (i=0; i<4; i++) {
        LKf (k, 20, &a, &e, &b, &d);
        S3f (&a, &e, &b, &d, &c);
        SKf (k, 16, &e, &b, &d, &c);

        LKf (k, 24, &c, &b, &a, &e);
        S2f (&c, &b, &a, &e, &d);
        SKf (k, 20, &a, &e, &b, &d);

        LKf (k, 28, &b, &e, &c, &a);
        S1f (&b, &e, &c, &a, &d);
        SKf (k, 24, &c, &b, &a, &e);

        LKf (k, 32, &a, &b, &c, &d);
        S0f (&a, &b, &c, &d, &e);
        SKf (k, 28, &b, &e, &c, &a);

        k += 8*4;

        LKf (k,  4, &a, &c, &d, &b);
        S7f (&a, &c, &d, &b, &e);
        SKf (k,  0, &d, &e, &b, &a);

        LKf (k,  8, &a, &c, &b, &e);
        S6f (&a, &c, &b, &e, &d);
        SKf (k,  4, &a, &c, &d, &b);

        LKf (k, 12, &b, &a, &e, &c);
        S5f (&b, &a, &e, &c, &d);
        SKf (k,  8, &a, &c, &b, &e);

        LKf (k, 16, &e, &b, &d, &c);
        S4f (&e, &b, &d, &c, &a);
        SKf (k, 12, &b, &a, &e, &c);
    }
    LKf (k, 20, &a, &e, &b, &d);
    S3f (&a, &e, &b, &d, &c);
    SKf (k, 16, &e, &b, &d, &c);
}
```

A.9 twofish.c

Listing A.9: twofish.c

```
/* ==========================================================================
 * twofish.c
 *
 * Twofish key schedule implementation for Interrogate
 *
 * Adapted for Interrogate use by Carsten Maartmann-Moe
 * <maartman@stud.ntnu.no>, see full licencing details for original code
 * below.
 * ==========================================================================
 */

/*
 ---------------------------------------------------------------------------
 Copyright (c) 1999, Dr Brian Gladman, Worcester, UK.   All rights reserved.

 LICENSE TERMS

 The free distribution and use of this software is allowed (with or without
 changes) provided that:

  1. source code distributions include the above copyright notice, this
     list of conditions and the following disclaimer;

  2. binary distributions include the above copyright notice, this list
     of conditions and the following disclaimer in their documentation;

  3. the name of the copyright holder is not used to endorse products
     built using this software without specific written permission.

 DISCLAIMER

 This software is provided 'as is' with no explicit or implied warranties
 in respect of its properties, including, but not limited to, correctness
 and/or fitness for purpose.
 ---------------------------------------------------------------------------

 My thanks to Doug Whiting and Niels Ferguson for comments that led
 to improvements in this implementation.

 Issue Date: 14th January 1999
*/

#include <stdio.h>
#include <stdlib.h>
#include "interrogate.h"

#define extract_byte(x,n)   ((unsigned char)((x) >> (8 * n)))

#define G_M 0x0169

unsigned char RS[4][8] =
{
    { 0x01, 0xA4, 0x55, 0x87, 0x5A, 0x58, 0xDB, 0x9E, },
    { 0xA4, 0x56, 0x82, 0xF3, 0x1E, 0xC6, 0x68, 0xE5, },
    { 0x02, 0xA1, 0xFC, 0xC1, 0x47, 0xAE, 0x3D, 0x19, },
    { 0xA4, 0x55, 0x87, 0x5A, 0x58, 0xDB, 0x9E, 0x03, },
};

static unsigned char  tab_5b[4] =
    { 0, G_M >> 2, G_M >> 1, (G_M >> 1) ^ (G_M >> 2) };
static unsigned char  tab_ef[4] =
    { 0, (G_M >> 1) ^ (G_M >> 2), G_M >> 1, G_M >> 2 };

#define ffm_01(x)    (x)
#define ffm_5b(x)   ((x) ^ ((x) >> 2) ^ tab_5b[(x) & 3])
#define ffm_ef(x)   ((x) ^ ((x) >> 1) ^ ((x) >> 2) ^ tab_ef[(x) & 3])

static unsigned char ror4[16] = { 0, 8, 1, 9, 2, 10, 3, 11,
                                  4, 12, 5, 13, 6, 14, 7, 15 };
```

```
static unsigned char ashx[16] = { 0, 9, 2, 11, 4, 13, 6, 15,
                                  8, 1, 10, 3, 12, 5, 14, 7 };

static unsigned char qt0[2][16] =
    {   { 8, 1, 7, 13, 6, 15, 3, 2, 0, 11, 5, 9, 14, 12, 10, 4 },
        { 2, 8, 11, 13, 15, 7, 6, 14, 3, 1, 9, 4, 0, 10, 12, 5 }
    };

static unsigned char qt1[2][16] =
    {   { 14, 12, 11, 8, 1, 2, 3, 5, 15, 4, 10, 6, 7, 0, 9, 13 },
        { 1, 14, 2, 11, 4, 12, 3, 7, 6, 13, 10, 5, 15, 9, 0, 8 }
    };

static unsigned char qt2[2][16] =
    {   { 11, 10, 5, 14, 6, 13, 9, 0, 12, 8, 15, 3, 2, 4, 7, 1 },
        { 4, 12, 7, 5, 1, 6, 9, 10, 0, 14, 13, 8, 2, 11, 3, 15 }
    };

static unsigned char qt3[2][16] =
    {   { 13, 7, 15, 4, 1, 2, 6, 14, 9, 11, 3, 0, 8, 5, 12, 10 },
        { 11, 9, 5, 1, 12, 3, 13, 14, 6, 4, 7, 15, 2, 0, 8, 10 }
    };


static unsigned char qp(const unsigned int n, const unsigned char x) {
    unsigned char  a0, a1, a2, a3, a4, b0, b1, b2, b3, b4;

    a0 = x >> 4;
    b0 = x & 15;
    a1 = a0 ^ b0;
    b1 = ror4[b0] ^ ashx[a0];
    a2 = qt0[n][a1];
    b2 = qt1[n][b1];
    a3 = a2 ^ b2;
    b3 = ror4[b2] ^ ashx[a2];
    a4 = qt2[n][a3];
    b4 = qt3[n][b3];
    return (b4 << 4) | a4;
};

/* Q tables */

static unsigned int  qt_gen = 0;
static unsigned char  q_tab[2][256];

#define q(n,x)  q_tab[n][x]

static void gen_qtab(void) {
    unsigned int  i;

    for(i = 0; i < 256; ++i) {
        q(0,i) = qp(0, (unsigned char)i);
        q(1,i) = qp(1, (unsigned char)i);
    }
};

/* M tables */
static unsigned int  mt_gen = 0;
static unsigned int  m_tab[4][256];

static void gen_mtab(void) {
    unsigned int  i, f01, f5b, fef;

    for(i = 0; i < 256; ++i) {
        f01 = q(1,i);
        f5b = ffm_5b(f01);
        fef = ffm_ef(f01);
        m_tab[0][i] = f01 + (f5b << 8) + (fef << 16) + (fef << 24);
        m_tab[2][i] = f5b + (fef << 8) + (f01 << 16) + (fef << 24);

        f01 = q(0,i);
        f5b = ffm_5b(f01);
        fef = ffm_ef(f01);
        m_tab[1][i] = fef + (fef << 8) + (f5b << 16) + (f01 << 24);
```

```
        m_tab[3][i] = f5b + (f01 << 8) + (fef << 16) + (f5b << 24);
    }
};

#define mds(n,x)    m_tab[n][x]

static unsigned int h_fun(twofish_tc *instance, const unsigned int x,
                          const unsigned int key[]) {
    unsigned int  b0, b1, b2, b3;

    b0 = extract_byte(x, 0);
    b1 = extract_byte(x, 1);
    b2 = extract_byte(x, 2);
    b3 = extract_byte(x, 3);

    switch(instance->k_len) {
    case 4:
        b0 = q(1, (unsigned char) b0) ^ extract_byte(key[3],0);
        b1 = q(0, (unsigned char) b1) ^ extract_byte(key[3],1);
        b2 = q(0, (unsigned char) b2) ^ extract_byte(key[3],2);
        b3 = q(1, (unsigned char) b3) ^ extract_byte(key[3],3);
    case 3:
        b0 = q(1, (unsigned char) b0) ^ extract_byte(key[2],0);
        b1 = q(1, (unsigned char) b1) ^ extract_byte(key[2],1);
        b2 = q(0, (unsigned char) b2) ^ extract_byte(key[2],2);
        b3 = q(0, (unsigned char) b3) ^ extract_byte(key[2],3);
    case 2:
        b0 = q(0, (unsigned char) (q(0, (unsigned char) b0) ^
                                   extract_byte(key[1],0))) ^
                                   extract_byte(key[0],0);
        b1 = q(0, (unsigned char) (q(1, (unsigned char) b1) ^
                                   extract_byte(key[1],1))) ^
                                   extract_byte(key[0],1);
        b2 = q(1, (unsigned char) (q(0, (unsigned char) b2) ^
                                   extract_byte(key[1],2))) ^
                                   extract_byte(key[0],2);
        b3 = q(1, (unsigned char) (q(1, (unsigned char) b3) ^
                                   extract_byte(key[1],3))) ^
                                   extract_byte(key[0],3);
    }

    return  mds(0, b0) ^ mds(1, b1) ^ mds(2, b2) ^ mds(3, b3);
};

#define q20(x)  q(0,q(0,x) ^ extract_byte(key[1],0)) ^ extract_byte(key[0],0)
#define q21(x)  q(0,q(1,x) ^ extract_byte(key[1],1)) ^ extract_byte(key[0],1)
#define q22(x)  q(1,q(0,x) ^ extract_byte(key[1],2)) ^ extract_byte(key[0],2)
#define q23(x)  q(1,q(1,x) ^ extract_byte(key[1],3)) ^ extract_byte(key[0],3)

#define q30(x)  q(0,q(0,q(1, x) ^ extract_byte(key[2],0)) ^ extract_byte(key
     [1],0)) ^ extract_byte(key[0],0)
#define q31(x)  q(0,q(1,q(1, x) ^ extract_byte(key[2],1)) ^ extract_byte(key
     [1],1)) ^ extract_byte(key[0],1)
#define q32(x)  q(1,q(0,q(0, x) ^ extract_byte(key[2],2)) ^ extract_byte(key
     [1],2)) ^ extract_byte(key[0],2)
#define q33(x)  q(1,q(1,q(0, x) ^ extract_byte(key[2],3)) ^ extract_byte(key
     [1],3)) ^ extract_byte(key[0],3)

#define q40(x)  q(0,q(0,q(1, q(1, x) ^ extract_byte(key[3],0)) ^ extract_byte
     (key[2],0)) ^ extract_byte(key[1],0)) ^ extract_byte(key[0],0)
#define q41(x)  q(0,q(1,q(1, q(0, x) ^ extract_byte(key[3],1)) ^ extract_byte
     (key[2],1)) ^ extract_byte(key[1],1)) ^ extract_byte(key[0],1)
#define q42(x)  q(1,q(0,q(0, q(0, x) ^ extract_byte(key[3],2)) ^ extract_byte
     (key[2],2)) ^ extract_byte(key[1],2)) ^ extract_byte(key[0],2)
#define q43(x)  q(1,q(1,q(0, q(1, x) ^ extract_byte(key[3],3)) ^ extract_byte
     (key[2],3)) ^ extract_byte(key[1],3)) ^ extract_byte(key[0],3)

void gen_mk_tab(twofish_tc *instance, unsigned int key[]) {
    unsigned int  i;
    unsigned char  by;

    unsigned int *mk_tab = instance->mk_tab;

    switch(instance->k_len) {
```

```
    case 2:
        for(i = 0; i < 256; ++i) {
            by = (unsigned char)i;

            mk_tab[0 + 4*i] = mds(0, q20(by));
            mk_tab[1 + 4*i] = mds(1, q21(by));

            mk_tab[2 + 4*i] = mds(2, q22(by));
            mk_tab[3 + 4*i] = mds(3, q23(by));

        }
        break;

    case 3:
        for(i = 0; i < 256; ++i) {
            by = (unsigned char)i;

            mk_tab[0 + 4*i] = mds(0, q30(by));
            mk_tab[1 + 4*i] = mds(1, q31(by));

        }
        break;

    case 4:
        for(i = 0; i < 256; ++i) {
            by = (unsigned char)i;

            mk_tab[0 + 4*i] = mds(0, q40(by));
            mk_tab[1 + 4*i] = mds(1, q41(by));

            mk_tab[2 + 4*i] = mds(2, q42(by));
            mk_tab[3 + 4*i] = mds(3, q43(by));

        }
    }
};

#    define g0_fun(x) ( mk_tab[0 + 4*extract_byte(x,0)] ^ mk_tab[1 + 4*
     extract_byte(x,1)] \
                        ^ mk_tab[2 + 4*extract_byte(x,2)] ^ mk_tab[3 + 4*
                            extract_byte(x,3)] )
#    define g1_fun(x) ( mk_tab[0 + 4*extract_byte(x,3)] ^ mk_tab[1 + 4*
     extract_byte(x,0)] \
                        ^ mk_tab[2 + 4*extract_byte(x,1)] ^ mk_tab[3 + 4*
                            extract_byte(x,2)] )

#define G_MOD   0x0000014d

unsigned int mds_rem(unsigned int p0, unsigned int p1) {
    unsigned int  i, t, u;

    for(i = 0; i < 8; ++i) {
        t = p1 >> 24;   // get most significant coefficient
        p1 = (p1 << 8) | (p0 >> 24);
        p0 <<= 8;  // shift others up
        // multiply t by a (the primitive element - i.e. left shift)
        u = (t << 1);
        if(t & 0x80)            // subtract modular polynomial on overflow
            u ^= G_MOD;
        p1 ^= t ^ (u << 16);    // remove t * (a * x^2 + 1)
        u ^= (t >> 1);          // form u = a * t + t / a = t * (a + 1 / a);
        if(t & 0x01)            // add the modular polynomial on underflow
            u ^= G_MOD >> 1;
        p1 ^= (u << 24) | (u << 8); // remove t * (a + 1/a) * (x^3 + x)
    }

    return p1;
};

/* Initialise the key schedule from the user supplied key   */
void twofish_set_key(twofish_tc *instance, const unsigned int in_key[], const
      unsigned int key_len) {
    unsigned int  i, a, b, me_key[4], mo_key[4];
    unsigned int *l_key, *s_key;
```

```
    l_key = instance->l_key;
    s_key = instance->s_key;

    if(!qt_gen) {
        gen_qtab();
        qt_gen = 1;
    }

    if(!mt_gen) {
        gen_mtab();
        mt_gen = 1;
    }

    instance->k_len = key_len / 64;   /* 2, 3 or 4 */

    for(i = 0; i < instance->k_len; ++i) {
        a = in_key[i + i];
        me_key[i] = a;
        b = in_key[i + i + 1];
        mo_key[i] = b;
        s_key[instance->k_len - i - 1] = mds_rem(a, b);
    }

    for(i = 0; i < 40; i += 2) {
        a = 0x01010101 * i;
        b = a + 0x01010101;
        a = h_fun(instance, a, me_key);
        b = rotlFixed(h_fun(instance, b, mo_key), 8);
        l_key[i] = a + b;
        l_key[i + 1] = rotlFixed(a + 2 * b, 9);
    }
    gen_mk_tab(instance, s_key);

    return;
};
```

A.10 Makefile

Listing A.10: Makefile

```
# ==========================================================================
# Makefile
#
# Makefile for Interrogate
#
# Author: Carsten Maartmann-Moe <carmaa@gmail.com>
# ==========================================================================

.SUFFIXES:
.SUFFIXES: .c .o .do
CC=gcc
CFLAGS=-Wall
LDFLAGS=
DEBUGFLAGS=-Wall -DDEBUG -g
LIBS=-lm
OBJS=interrogate.o stat.o rsa.o aes.o serpent.o twofish.o util.o virtmem.o
DBOBJS=interrogate.do stat.do rsa.do aes.do serpent.do twofish.do util.do
     virtmem.do
EXECNAME=interrogate

.c.do:; $(CC) -c -o $@ $(DEBUGFLAGS) $<

all: interrogate

interrogate: $(OBJS)
    $(CC) $(CFLAGS) -o $(EXECNAME) $(OBJS) $(LIBS)

debug: $(DBOBJS)
    $(CC) $(DEBUGFLAGS) -o $(EXECNAME) $(DBOBJS) $(LIBS)

clean:
    rm -f *.o *.do *.bak *.der interrogate
```

Appendix B

Data Structures Related to Windows Memory Analysis

In this appendix, we present some of the memory-related structures in Windows XP, as outputted from the Windows Debugging Tools. These are provided as a convenience for developers that wish to extend the authors work in this thesis, or in other ways contribute towards forensics procedures in the field of memory analysis. No further explanation of these structures are given; for a good treatment of windows memory internals, the *Windows Internals* [80] series of books are a good references.

Listing B.1: EPROCESS data structure

```
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
   +0x11c VadRoot          : Ptr32 Void
   +0x120 VadHint          : Ptr32 Void
   +0x124 CloneRoot        : Ptr32 Void
   +0x128 NumberOfPrivatePages : Uint4B
   +0x12c NumberOfLockedPages : Uint4B
   +0x130 Win32Process     : Ptr32 Void
   +0x134 Job              : Ptr32 _EJOB
   +0x138 SectionObject    : Ptr32 Void
   +0x13c SectionBaseAddress : Ptr32 Void
   +0x140 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
   +0x144 WorkingSetWatch  : Ptr32 _PAGEFAULT_HISTORY
```

```
+0x148 Win32WindowStation : Ptr32 Void
+0x14c InheritedFromUniqueProcessId : Ptr32 Void
+0x150 LdtInformation   : Ptr32 Void
+0x154 VadFreeHint      : Ptr32 Void
+0x158 VdmObjects       : Ptr32 Void
+0x15c DeviceMap        : Ptr32 Void
+0x160 PhysicalVadList  : _LIST_ENTRY
+0x168 PageDirectoryPte : _HARDWARE_PTE_X86
+0x168 Filler           : Uint8B
+0x170 Session          : Ptr32 Void
+0x174 ImageFileName    : [16] UChar
+0x184 JobLinks         : _LIST_ENTRY
+0x18c LockedPagesList  : Ptr32 Void
+0x190 ThreadListHead   : _LIST_ENTRY
+0x198 SecurityPort     : Ptr32 Void
+0x19c PaeTop           : Ptr32 Void
+0x1a0 ActiveThreads    : Uint4B
+0x1a4 GrantedAccess    : Uint4B
+0x1a8 DefaultHardErrorProcessing : Uint4B
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 Peb              : Ptr32 _PEB
+0x1b4 PrefetchTrace    : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER
+0x1c0 WriteOperationCount : _LARGE_INTEGER
+0x1c8 OtherOperationCount : _LARGE_INTEGER
+0x1d0 ReadTransferCount : _LARGE_INTEGER
+0x1d8 WriteTransferCount : _LARGE_INTEGER
+0x1e0 OtherTransferCount : _LARGE_INTEGER
+0x1e8 CommitChargeLimit : Uint4B
+0x1ec CommitChargePeak : Uint4B
+0x1f0 AweInfo          : Ptr32 Void
+0x1f4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm               : _MMSUPPORT
+0x238 LastFaultCount   : Uint4B
+0x23c ModifiedPageCount : Uint4B
+0x240 NumberOfVads     : Uint4B
+0x244 JobStatus        : Uint4B
+0x248 Flags            : Uint4B
+0x248 CreateReported   : Pos 0, 1 Bit
+0x248 NoDebugInherit   : Pos 1, 1 Bit
+0x248 ProcessExiting   : Pos 2, 1 Bit
+0x248 ProcessDelete    : Pos 3, 1 Bit
+0x248 Wow64SplitPages  : Pos 4, 1 Bit
+0x248 VmDeleted        : Pos 5, 1 Bit
+0x248 OutswapEnabled   : Pos 6, 1 Bit
+0x248 Outswapped       : Pos 7, 1 Bit
+0x248 ForkFailed       : Pos 8, 1 Bit
+0x248 HasPhysicalVad   : Pos 9, 1 Bit
+0x248 AddressSpaceInitialized : Pos 10, 2 Bits
+0x248 SetTimerResolution : Pos 12, 1 Bit
+0x248 BreakOnTermination : Pos 13, 1 Bit
+0x248 SessionCreationUnderway : Pos 14, 1 Bit
+0x248 WriteWatch       : Pos 15, 1 Bit
+0x248 ProcessInSession : Pos 16, 1 Bit
+0x248 OverrideAddressSpace : Pos 17, 1 Bit
+0x248 HasAddressSpace  : Pos 18, 1 Bit
+0x248 LaunchPrefetched : Pos 19, 1 Bit
+0x248 InjectInpageErrors : Pos 20, 1 Bit
+0x248 VmTopDown        : Pos 21, 1 Bit
+0x248 Unused3          : Pos 22, 1 Bit
+0x248 Unused4          : Pos 23, 1 Bit
+0x248 VdmAllowed       : Pos 24, 1 Bit
+0x248 Unused           : Pos 25, 5 Bits
+0x248 Unused1          : Pos 30, 1 Bit
+0x248 Unused2          : Pos 31, 1 Bit
+0x24c ExitStatus       : Int4B
+0x250 NextPageColor    : Uint2B
+0x252 SubSystemMinorVersion : UChar
+0x253 SubSystemMajorVersion : UChar
+0x252 SubSystemVersion : Uint2B
+0x254 PriorityClass    : UChar
+0x255 WorkingSetAcquiredUnsafe : UChar
+0x258 Cookie           : Uint4B
```

Listing B.2: KPROCESS data structure

```
ntdll!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
   +0x067 DisableBoost     : UChar
   +0x068 PowerState       : UChar
   +0x069 DisableQuantum   : UChar
   +0x06a IdealNode        : UChar
   +0x06b Flags            : _KEXECUTE_OPTIONS
   +0x06b ExecuteOptions   : UChar
```

Listing B.3: PEB data structure

```
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [1] Uint4B
   +0x034 AtlThunkSListPtr32 : Uint4B
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 Ptr32 Void
   +0x094 GdiSharedHandleTable : Ptr32 Void
```

```
   +0x098 ProcessStarterHelper : Ptr32 Void
   +0x09c GdiDCAttributeList : Uint4B
   +0x0a0 LoaderLock       : Ptr32 Void
   +0x0a4 OSMajorVersion   : Uint4B
   +0x0a8 OSMinorVersion   : Uint4B
   +0x0ac OSBuildNumber    : Uint2B
   +0x0ae OSCSDVersion     : Uint2B
   +0x0b0 OSPlatformId     : Uint4B
   +0x0b4 ImageSubsystem   : Uint4B
   +0x0b8 ImageSubsystemMajorVersion : Uint4B
   +0x0bc ImageSubsystemMinorVersion : Uint4B
   +0x0c0 ImageProcessAffinityMask : Uint4B
   +0x0c4 GdiHandleBuffer  : [34] Uint4B
   +0x14c PostProcessInitRoutine : Ptr32     void
   +0x150 TlsExpansionBitmap : Ptr32 Void
   +0x154 TlsExpansionBitmapBits : [32] Uint4B
   +0x1d4 SessionId        : Uint4B
   +0x1d8 AppCompatFlags   : _ULARGE_INTEGER
   +0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
   +0x1e8 pShimData        : Ptr32 Void
   +0x1ec AppCompatInfo    : Ptr32 Void
   +0x1f0 CSDVersion       : _UNICODE_STRING
   +0x1f8 ActivationContextData : Ptr32 Void
   +0x1fc ProcessAssemblyStorageMap : Ptr32 Void
   +0x200 SystemDefaultActivationContextData : Ptr32 Void
   +0x204 SystemAssemblyStorageMap : Ptr32 Void
   +0x208 MinimumStackCommit : Uint4B
```

Listing B.4: ETHREAD data structure

```
ntdll!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
   +0x224 StartAddress     : Ptr32 Void
   +0x228 Win32StartAddress : Ptr32 Void
   +0x228 LpcReceivedMessageId : Uint4B
   +0x22c ThreadListEntry  : _LIST_ENTRY
   +0x234 RundownProtect   : _EX_RUNDOWN_REF
   +0x238 ThreadLock       : _EX_PUSH_LOCK
   +0x23c LpcReplyMessageId : Uint4B
   +0x240 ReadClusterSize  : Uint4B
   +0x244 GrantedAccess    : Uint4B
   +0x248 CrossThreadFlags : Uint4B
   +0x248 Terminated       : Pos 0, 1 Bit
   +0x248 DeadThread       : Pos 1, 1 Bit
   +0x248 HideFromDebugger : Pos 2, 1 Bit
   +0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
   +0x248 SystemThread     : Pos 4, 1 Bit
   +0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
   +0x248 BreakOnTermination : Pos 6, 1 Bit
```

```
   +0x248 SkipCreationMsg  : Pos 7, 1 Bit
   +0x248 SkipTerminationMsg : Pos 8, 1 Bit
   +0x24c SameThreadPassiveFlags : Uint4B
   +0x24c ActiveExWorker   : Pos 0, 1 Bit
   +0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
   +0x24c MemoryMaker      : Pos 2, 1 Bit
   +0x250 SameThreadApcFlags : Uint4B
   +0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
   +0x250 LpcExitThreadCalled : Pos 1, 1 Bit
   +0x250 AddressSpaceOwner : Pos 2, 1 Bit
   +0x254 ForwardClusterOnly : UChar
   +0x255 DisablePageFaultClustering : UChar
```

Listing B.5: KTHREAD data structure

```
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
   +0x050 IdleSwapBlock    : UChar
   +0x051 Spare0           : [3] UChar
   +0x054 WaitStatus       : Int4B
   +0x058 WaitIrql         : UChar
   +0x059 WaitMode         : Char
   +0x05a WaitNext         : UChar
   +0x05b WaitReason       : UChar
   +0x05c WaitBlockList    : Ptr32 _KWAIT_BLOCK
   +0x060 WaitListEntry    : _LIST_ENTRY
   +0x060 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x068 WaitTime         : Uint4B
   +0x06c BasePriority     : Char
   +0x06d DecrementCount   : UChar
   +0x06e PriorityDecrement : Char
   +0x06f Quantum          : Char
   +0x070 WaitBlock        : [4] _KWAIT_BLOCK
   +0x0d0 LegoData         : Ptr32 Void
   +0x0d4 KernelApcDisable : Uint4B
   +0x0d8 UserAffinity     : Uint4B
   +0x0dc SystemAffinityActive : UChar
   +0x0dd PowerState       : UChar
   +0x0de NpxIrql          : UChar
   +0x0df InitialNode      : UChar
   +0x0e0 ServiceTable     : Ptr32 Void
   +0x0e4 Queue            : Ptr32 _KQUEUE
   +0x0e8 ApcQueueLock     : Uint4B
   +0x0f0 Timer            : _KTIMER
   +0x118 QueueListEntry   : _LIST_ENTRY
   +0x120 SoftAffinity     : Uint4B
   +0x124 Affinity         : Uint4B
   +0x128 Preempted        : UChar
   +0x129 ProcessReadyQueue : UChar
   +0x12a KernelStackResident : UChar
   +0x12b NextProcessor    : UChar
   +0x12c CallbackStack    : Ptr32 Void
   +0x130 Win32Thread      : Ptr32 Void
   +0x134 TrapFrame        : Ptr32 _KTRAP_FRAME
   +0x138 ApcStatePointer  : [2] Ptr32 _KAPC_STATE
   +0x140 PreviousMode     : Char
   +0x141 EnableStackSwap  : UChar
   +0x142 LargeStack       : UChar
```

```
   +0x143 ResourceIndex    : UChar
   +0x144 KernelTime       : Uint4B
   +0x148 UserTime         : Uint4B
   +0x14c SavedApcState    : _KAPC_STATE
   +0x164 Alertable        : UChar
   +0x165 ApcStateIndex    : UChar
   +0x166 ApcQueueable     : UChar
   +0x167 AutoAlignment    : UChar
   +0x168 StackBase        : Ptr32 Void
   +0x16c SuspendApc       : _KAPC
   +0x19c SuspendSemaphore : _KSEMAPHORE
   +0x1b0 ThreadListEntry  : _LIST_ENTRY
   +0x1b8 FreezeCount      : Char
   +0x1b9 SuspendCount     : Char
   +0x1ba IdealProcessor   : UChar
   +0x1bb DisableBoost     : UChar
```

Listing B.6: TEB data structure

```
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   +0x040 Win32ThreadInfo  : Ptr32 Void
   +0x044 User32Reserved   : [26] Uint4B
   +0x0ac UserReserved     : [5] Uint4B
   +0x0c0 WOW32Reserved    : Ptr32 Void
   +0x0c4 CurrentLocale    : Uint4B
   +0x0c8 FpSoftwareStatusRegister : Uint4B
   +0x0cc SystemReserved1  : [54] Ptr32 Void
   +0x1a4 ExceptionCode    : Int4B
   +0x1a8 ActivationContextStack : _ACTIVATION_CONTEXT_STACK
   +0x1bc SpareBytes1      : [24] UChar
   +0x1d4 GdiTebBatch      : _GDI_TEB_BATCH
   +0x6b4 RealClientId     : _CLIENT_ID
   +0x6bc GdiCachedProcessHandle : Ptr32 Void
   +0x6c0 GdiClientPID     : Uint4B
   +0x6c4 GdiClientTID     : Uint4B
   +0x6c8 GdiThreadLocalInfo : Ptr32 Void
   +0x6cc Win32ClientInfo  : [62] Uint4B
   +0x7c4 glDispatchTable  : [233] Ptr32 Void
   +0xb68 glReserved1      : [29] Uint4B
   +0xbdc glReserved2      : Ptr32 Void
   +0xbe0 glSectionInfo    : Ptr32 Void
   +0xbe4 glSection        : Ptr32 Void
   +0xbe8 glTable          : Ptr32 Void
   +0xbec glCurrentRC      : Ptr32 Void
   +0xbf0 glContext        : Ptr32 Void
   +0xbf4 LastStatusValue  : Uint4B
   +0xbf8 StaticUnicodeString : _UNICODE_STRING
   +0xc00 StaticUnicodeBuffer : [261] Uint2B
   +0xe0c DeallocationStack : Ptr32 Void
   +0xe10 TlsSlots         : [64] Ptr32 Void
   +0xf10 TlsLinks         : _LIST_ENTRY
   +0xf18 Vdm              : Ptr32 Void
   +0xf1c ReservedForNtRpc : Ptr32 Void
   +0xf20 DbgSsReserved    : [2] Ptr32 Void
   +0xf28 HardErrorsAreDisabled : Uint4B
   +0xf2c Instrumentation  : [16] Ptr32 Void
   +0xf6c WinSockData      : Ptr32 Void
   +0xf70 GdiBatchCount    : Uint4B
   +0xf74 InDbgPrint       : UChar
   +0xf75 FreeStackOnTermination : UChar
   +0xf76 HasFiberData     : UChar
   +0xf77 IdealProcessor   : UChar
   +0xf78 Spare3           : Uint4B
   +0xf7c ReservedForPerf  : Ptr32 Void
```

```
      +0xf80 ReservedForOle   : Ptr32 Void
      +0xf84 WaitingOnLoaderLock : Uint4B
      +0xf88 Wx86Thread       : _Wx86ThreadState
      +0xf94 TlsExpansionSlots : Ptr32 Ptr32 Void
      +0xf98 ImpersonationLocale : Uint4B
      +0xf9c IsImpersonating  : Uint4B
      +0xfa0 NlsCache         : Ptr32 Void
      +0xfa4 pShimData        : Ptr32 Void
      +0xfa8 HeapVirtualAffinity : Uint4B
      +0xfac CurrentTransactionHandle : Ptr32 Void
      +0xfb0 ActiveFrame      : Ptr32 _TEB_ACTIVE_FRAME
      +0xfb4 SafeThunkCall    : UChar
      +0xfb5 BooleanSpare     : [3] UChar
```

Listing B.7: POOL_HEADER data structure

```
nt!_POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 9 Bits
   +0x000 PoolIndex        : Pos 9, 7 Bits
   +0x002 BlockSize        : Pos 0, 9 Bits
   +0x002 PoolType         : Pos 9, 7 Bits
   +0x000 Ulong1           : Uint4B
   +0x004 ProcessBilled    : Ptr32 _EPROCESS
   +0x004 PoolTag          : Uint4B
   +0x004 AllocatorBackTraceIndex : Uint2B
   +0x006 PoolTagHash      : Uint2B
```

Appendix C

Copyright Information

All copyrights not owned by the author is listed in the following section.

C.1 Interrogate Source Code Licence (GPL)

For the Interrrogate GPL licence, please see `http://www.gnu.org/licenses/gpl.html`.

C.2 Wikimedia Content

The following figures are taken from WikiMedia Commons (`http://commons.wikimedia.org/`).

Figure 3.1 Copyright © User:Dysprosia, all rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, and this list of conditions;
- Redistributions in binary form must reproduce the above copyright notice, and this list of conditions in the documentation and/or other materials provided with the distribution;
- Neither the name of en:User:Dysprosia nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Figure 3.4 This file is licensed under the Creative Commons Attribution ShareAlike license versions 3.0 (`http://creativecommons.org/licenses/by-sa/3.0/`), and created by WikiMedia user RokerHRO (`http://commons.wikimedia.org/wiki/User:RokerHRO`).

C.3 Copyrighted Content

The images on the front page and in Figure 2.3(a), *The Disintegration of the Persistence of Memory* and *The Persistence of Memory* are both by Salvador Dalí, ©2008 Salvador Dalí, Gala-Salvador Dalí Foundation/Artist Rights Society (ARS), New York. They are (partly) reproduced here under the Fair Use doctrine of the U.S. Copyright Act of 1976, 17 U.S.C. § 107 and the Norwegian Copyright Law "Åndsverkloven" § 23 under the following reasoning:

1. The images are only used for informational, illustrative and educational purposes only
2. The images are readily available on the Internet and in the public domain
3. Images are of low resolution and would be unlikely to impact sales of prints or be usable as a desktop background
4. The images are used for non-profit research
5. There is no alternative, public domain or free-copyrighted replacement available

The Persistence of Memory was taken from MoMA.org (`http://www.moma.org/collection/browse_results.php?object_id=79018`), and *The Disintegration of the Persistence of Memory* from `http://www.dali-gallery.com/html/galleries/painting19.htm`.