available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

The persistence of memory: Forensic identification and extraction of cryptographic keys

Carsten Maartmann-Moe[a,*,1], *Steffen E. Thorkildsen*[b], *André Årnes*[c,2]

[a]*Department of Telematics, Norwegian University of Science and Technology, O.S. Bragstads Plass 2B, N-7491 Trondheim, Norway*
[b]*National Criminal Investigation Service, Norway*
[c]*Norwegian Information Security Laboratory, Gjøvik University College, PO Box 191, N-2802 Gjøvik, Norway*

Keywords:
Digital forensics
Data hiding and recovery
Memory analysis
Memory dumping
Applied cryptography
Live analysis
Cryptographic evidence
Incident response
Tool testing and development

ABSTRACT

The increasing popularity of cryptography poses a great challenge in the field of digital forensics. Digital evidence protected by strong encryption may be impossible to decrypt without the correct key. We propose novel methods for cryptographic key identification and present a new proof of concept tool named *Interrogate* that searches through volatile memory and recovers cryptographic keys used by the ciphers AES, Serpent and Twofish. By using the tool in a virtual digital crime scene, we simulate and examine the different states of systems where well known and popular cryptosystems are installed. Our experiments show that the chances of uncovering cryptographic keys are high when the digital crime scene are in certain well-defined states. Finally, we argue that the consequence of this and other recent results regarding memory acquisition require that the current practices of digital forensics should be guided towards a more forensically sound way of handling live analysis in a digital crime scene.

1. Introduction

Cryptography has grown to become one of the most important contributors to privacy and data security in an increasingly interconnected world. The use of cryptography also represents a challenge for digital forensics investigators, as it may be used to hide data that may shed light on the chain of events that constitutes an incident or crime. Since the nature of cryptography makes it attractive for hiding incriminating data, encrypted material encountered often contain exactly the evidence sought by investigators.

In this paper, we aim to study new methods for the identification and extraction of cryptographic keys from the volatile memory of computing devices as part of the *digital forensics process*. In this context, the keys and any encrypted contents may be considered to be *digital evidence* (i.e., digital data that contains reliable information that supports or refutes a hypothesis about an incident (Carrier and Spafford, 2004)) that is part of a *digital crime scene*. Note also that the main property of cryptographic keys in the context of digital forensics is that they may be a necessary prerequisite for the successful decryption of encrypted digital evidence.

Digital investigators are often forced to attempt brute-force and dictionary attacks to gain access to encrypted digital evidence, but these methods cannot circumvent strong cryptography and strong passwords. A paradox is that

* *Corresponding author.*
E-mail addresses: carsten@carmaa.com (C. Maartmann-Moe), steffen.thorkildsen@politiet.no (S.E. Thorkildsen), andre.arnes@hig.no (S.E. André Årnes).
[1] Carsten Maartmann-Moe is currently a Consultant at Ernst & Young in Norway.
[2] André Årnes is currently an Adjunct Associate Professor at Gjøvik University College and a Security and Identity Management Architect at Oracle Norway.
1742-2876/$ – see front matter © 2009 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2009.06.002

cryptographic keys may be present in computer memory at the time of the evidence acquisition. However, memory is not always acquired, and there are no standard tools for memory analysis and key extraction based on memory dumps.

The main contributions of this paper is the novel approach to Serpent and Twofish key structure identification and analysis, a method for virtual memory reconstruction, as well as the proposed introduction of cryptographic key searches in memory as part of the digital forensics process. Our results are validated through the implementation of a proof of concept tool and a series of experiments covering three cryptographic algorithms and ten software tools in a virtualized testbed.

The paper is structured as follows. Section 2 contains an overview of related research, Section 3 describes techniques for identifying keys in memory, and Section 4 discusses how to use Windows memory structure to optimize searches. Our experiments and results are presented in Sections 5 and 6, and the implications for the field of digital forensics is discussed in Section 7. Finally, future work and conclusions are provided in Section 8.

2. Related work

The acquisition and analysis of volatile memory for forensics purposes is a relatively immature procedure, even though the concept has been known for a long time (Crescenzo et al., 1999). The memory acquisition process is especially unstandardized, and there exists a large number of different approaches. A good comparison of the available methods for Microsoft Windows operating systems can be found in the paper *Windows Memory Forensics* (Ruff, 2007). The methods for extracting volatile memory ranges from DMA access via FireWire (Dornseif, 2005; Martin, 2007) to simply copying of memory from /dev/mem on Unix-flavor platforms.

Research on the age of freed user process data in physical memory has shown that large segments of pages are unlikely to survive more than 5 min, even on a lightly loaded system (Solomona et al., 2007). However, smaller segments and single pages may be found up to 2 h after initial memory commit. These results may limit the timeframe for successful recovery of cryptographic keys that are left in memory. To counter these issues, Chow et al. have proposed several methods for secure deallocation of sensitive data from memory (Chow et al., 2005). It is nevertheless clear that these results do not mitigate the fact that cryptographic keys need to be present in memory during encryption when using standard computer hardware.

The first approach on cryptographic key search and identification were proposed by Shamir and van Someren in 1998, suggesting the prospect of attacks against mainframes in their article *Playing Hide and Seek with Stored Keys* (Shamir and van Someren, 1998). They propose to use simple statistical and visual methods to locate memory regions that are likely to contain encryption keys. In a more recent article, Pettersson discusses searches for structural properties of the code that is holding the key, by analyzing and ''guesstimating'' the values of surrounding variables (Pettersson, 2007). Ptacek (2008) outlines 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 (Klein, 2006).

The authors of Volatility describe a hypothetical attack against TrueCrypt (Foundation, 2008), by studying its internal structures and behavior (Walters and Nick, 2007). They do, however, not describe how to locate the different structures in memory, and neither do they discuss the fact that some of these may be paged out, thereby breaking the chain of data structures that leads to the master key if only the memory dump is available for analysis.

Halderman et al. presented a recent breakthrough in their paper *Lest We Remember: Cold Boot Attacks on Encryption Keys* (Halderman et al., 2008). They demonstrate that it is possible to leverage remanence effects[3] in DRAM modules to coldboot the target computer, load a custom OS that extracts the memory to an external drive, locate the key material and finally decrypt the hard drives automatically. We owe the idea to utilize key schedules as a means for identification of cryptographic keys to this paper, and lately considerable effort has been directed at creating usable software for decryption of closed-source systems like BitLocker (Kaplan, 2007; Kornblum, 2008).

Most of these methods treat the memory as a large blob of bytes, although in fact memory is quite structured. Some of the methods suggest skipping duplicate regions and reserved address space, but do not consider to reduce the ''haystack'' by only looking at the probable regions of the memory. In other fields of memory analysis, analysts have dumped the memory address space of a specific process by fetching pages from RAM and swap space. The dumps are sometimes sufficient to verify[4] and even completely reconstruct executable files (Kornblum, 2006). According to several articles (for example, see Schuster, 2006 and Carvey, 2007), these techniques are able to identify trojans, rootkits and viruses that are stealthy and/or armored in Windows memory dumps.

Despite all these contemporary studies, there exist little empirical research on whether cryptographic keys are present in memory at the time of acquisition. In this paper we will demonstrate how to utilize several *search strategies* in combination with cryptographic knowledge to extract key material from volatile memory. We perform controlled experiments that will indicate the probability of a successful key extraction.

3. Cryptographic key identification

For the average end user, a cryptographic key is an abstract notation, hidden by obfuscation layers consisting of password churning and key hierarchies. In reality, symmetric cryptographic keys are just short sequences of random-looking bytes, often 16–32 bytes long. Even so, recent studies suggest

[3] Remanence effects is the effect that all Dynamic Random Access Memory (DRAM) modules keep their state for a period of time (typically a few seconds) before it needs to be refreshed by the memory controller, first mentioned as a security risk in a articles by Anderson (2001) and Gutmann (2001, 1996). The process of utilizing this effect to extract cryptographic keys is known as the ''coldboot technique''.

[4] By using tools like SSDeep by J. Kornblum.

that the representation of the keys in memory is far more structured than previously believed. Several properties and search strategies that may be used to locate such keys in a memory image have been suggested:

1. Brute-force with the memory image as dictionary (Kaplan, 2007). This is the ultimate naïve approach, and we did not experiment with this method.
2. Search for high-entropy regions. Using entropy to locate RSA keys were first proposed by Shamir and van Someren (1998).
3. Search for structural properties of the RSA encoding as first proposed by Klein (2006) and Ptacek (2008).
4. Search for the code structures (e.g., C structs) that contain the key. Previously suggested by Pettersson (2007), and later by Walters and Petroni.
5. Search for the key schedule, as suggested by Halderman et al. (2008)

3.1. Proof of concept tool: Interrogate

In our proof of concept tool *Interrogate* we implemented several of the above search methods. The last two methods are discussed cipher by cipher in the following sections together with a description of their representation of cipher keys in memory. In addition, we suggest to combine several of these methods to perform searches for Serpent and Twofish keys. These novel methods are implemented in Interrogate, together with a method for the reconstruction of virtual memory for processes as described in Section 4.

Interrogate is provided under the GNU Public License on http://sourceforge.net/projects/interrogate/ and features search strategies for RSA, AES, Serpent and Twofish keys. RSA and Serpent keys are found independent of their length, while Twofish keys are required to be 256 bits. For AES, a specification of 128, 192 or 256 bits is required. The tool is not limited to memory dumps; swap space and decoded hibernation-files may also be interesting targets.

3.2. AES key representation in memory

The Rijndael cipher was selected as the Advanced Encryption Standard (AES) in 2001 (NIST, 2001), formed from a proposal by Joan Daemen and Vincent Rijmen. It is a Substitution-Permutation (SP)-network based cipher that works on 128-bit 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, BestCrypt, PGP, Protect-Drive and Pointsec.

In Halderman et al. (2008), the researchers use 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, and kept in the memory while encryption/decryption is performed.

The AES key schedule computation generally uses the same approach for 128, 192 and 256 bit keys, albeit with slight variations. The generation procedure for all key sizes can be found in (NIST, 2001).

3.2.1. AES keys

The 128-bit *empty key* (all zeroes) generates the following AES 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

Notably, the key schedule is represented as a flat array of bytes in memory, where the first 16 bytes (or 128 bits) constitutes the original key. The remaining 112 bytes are the round keys derived from this key. As Halderman et al. noted, this makes it possible to generate key schedules for all offsets in memory and check whether the next 112 bytes matches the generated schedule. If it does, it is probably an AES key. This method holds for 192 and 256-bit keys as well. Furthermore, the key schedule acts as an error-correcting code, so that we may output all key schedules that have small Hamming distances from the generated schedule, and thereby compensate for an eventual bit decay due to the memory acquisition method (e.g., coldboot).

Since the code from Halderman et al. (2008) was not available at the time of research, we designed and implemented the algorithm in *Interrogate* to test the presence of keys in volatile memory based on the description in the paper. Additionally, we added support for 128 and 192-bit keys.

3.3. Serpent key representation in memory

Serpent came second in the AES selection process, based on a submission from Ross Anderson, Eli Biham and Lars Knudsen (Anderson et al., 2000). 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. Because of this design, we can treat all Serpent keys as 256-bit keys as far as this paper is concerned. Although not as widely adopted as AES, Serpent is featured in several cryptographic applications, among others TrueCrypt and BestCrypt.

Like AES, the Serpent cipher also generates its key schedule ahead of time. In addition, the key schedule is similar to that of AES (Anderson et al., 2000); it uses the user-supplied key as the first round key, with the following round keys are derived from this master key.

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, and utilizes a pre-key transformation of the user-supplied master key and its S-boxes to compute its key schedule. The result is a 560-byte array of the master key together with the 33 derived round keys; the two first 16-byte vectors are the 256-bit master key, and the 33 remaining rows the 128-bit sub (or round) keys.

3.3.1. *Identifying serpent keys*

We discovered that the error-correcting code properties of the AES key schedule also holds for Serpent, and we can consequently utilize a similar search strategy to identify Serpent keys. We designed and implemented this novel algorithm in the proof of concept tool Interrogate used to gather statistics for this paper.

3.4. *Twofish key representation in memory*

Twofish came in third at the last AES conference, submitted by Bruce Schneier et al. (2000). It is a 128-bit cipher that accepts variable-length keys with size $N = \{128, 192, 256\}$ bits. 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. These transformations include 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. Over 50 applications feature Twofish encryption, among others TrueCrypt, BestCrypt and PGP.

Twofish uses a slightly different approach than AES and Serpent, by utilizing key-dependent S-boxes together with the round keys in the encryption process (Schneier et al., 2000). 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 (we will use the term *MK table* throughout the paper) in memory. This is mainly done for performance, and the resulting encryption operation reduces itself to only four table lookups and three XORs.

The complex key schedule generation procedure generates a large amount of keying material. The full key schedule consists of 40 32-bit words of expanded key $K_0, \ldots, K_{39}$ where the first eight words $K_0, \ldots K_3$ and $K_4, \ldots K_7$ are the input and output whitening keys, respectively. Furthermore, it consists of the keys for the S-boxes, $S_{k-1}, \ldots, S_0$ where $k = N/64$, and the optional MK table of 4 KB.

The MK table makes an excellent search signature, but because the size of the whole key schedule data structure exceeds 4096 bytes (which is the usual size of a page in memory), the key schedule may be scattered over several pages at different locations in the physical memory. Worse, for the sake of our research, Twofish does not use its master key as part of its keying material. Thus we cannot use a similar procedure as for AES and Serpent to search for Twofish keys.

3.4.1. *Notes on the Twofish key schedule*

Early in the AES selection process, certain notes were made on the Twofish key schedule both by the authors of the algorithm (Schneier, 1998) and others (Mirza and Murphy, 1999). The Twofish team quickly researched the matter, and later proved that the properties did not affect the security of the cipher (Schneier et al., 1999). However, it is possible to leverage these properties to locate Twofish keys in volatile memory.

We have performed a quantitative analysis of some related statistical properties of the key schedule structure in order to generate a search signature for Twofish keys. For a large number of random Twofish key schedules, we see that the entropy value of the S-box keys (Fig. 1a) does not take on a uniformly distributed high-entropy value. We generated 10^{12} key schedules based on random master keys, and found that the entropy of the S key vector rather falls within distinct values.

Furthermore, we measured the entropy of the sub keys K_j, discovering that the they have entropy values in the relaxed range [6.1, 7.4], as seen in Fig. 1b. If we look at the MK table, we see that it can only take on one distinct entropy value, namely the maximum possible 8 bits per byte.

3.4.2. *Identifying TrueCrypt Twofish keys*

The TrueCrypt source code uses the following C structure to store the key schedule:

```
typedef struct {
   unsigned int l_key[40];
   unsigned int s_key[4];
   unsigned int mk_tab[4 * 256];
   unsigned int k_len;
} twofish_tc;
```

By searching for this structure and verifying the above entropic measurements, we are able to locate TrueCrypt Twofish keys.

3.4.3. *A less implementation-dependent search*

To counter the drawback of only being able to search for keys specific to TrueCrypt, we propose another method of locating Twofish key schedules[5] by means of counting runs. In addition to being highly entropic, the MK table also has a quite constant number of byte runs.[6] 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 1. By counting runs in a sliding 4 KB window, we can locate probable MK tables. To verify these tables, we perform the same checkups as with the TrueCrypt Twofish key schedule on the surrounding data, using

[5] It should also be noted that this method does not output the master *key*, but rather the *key schedule*. Nevertheless, that is enough to decrypt content encrypted under the related master key.

[6] A *byte run* is a sequence of bytes with the same value, e.g., a run of three with the byte `0x0f` is `0x0f0f0f`.

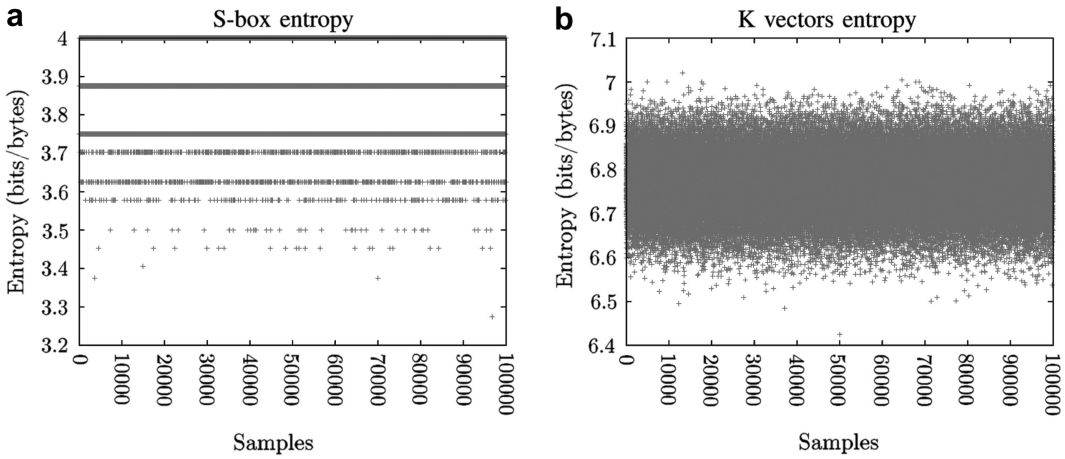

Fig. 1 – Plots of entropy from the Twofish S-box and *K* key vectors of 256-bit keys.

data structures taken from five publicly available implementations of Twofish. This novel search facilitates finding more than one type of key schedule data structures.[7]

By keeping track of the runs that fall out and enter the searching window, we can optimize our algorithm and thus reduce the runtime significantly. This implementation can be found in the source code of Interrogate.

4. Leveraging memory structure

As previously discussed, a digital investigator may face keys that are distributed over several non-contiguous pages in memory. In order to counter this situation and illustrate how to use Windows memory structures to optimize searches, 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 facilitates a significant reduction of search data given a normal amount of memory.

To reconstruct the virtual address space of a process, we only need to know the location of its Page Directory Base (PDB) (Russinovich and Solomon, 2005). Using this, the reconstruction procedure 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 knowledge about the cryptographic application (i.e., which process handles the cryptographic keys). For whole- and virtual disk cryptosystems on Windows, we have found that these threads usually operates in the System process, which has its PDB at `0x00039000` (assuming /NOPAE and no /3 GB boot switches set).

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

The reconstruction method can be used as a preprocessing step to reduce the search space for all the above search strategies, and hence significantly improve the performance of the search. It also enables the use of reconstructed memory in a dictionary attack to identify keys. Using preprocessing makes the average AES and Serpent key search on a 1 GB memory dump around 100 times faster, decreasing the running time from circa 2 h to 75 s on a 2 GHz processor.

5. Experiments

In our experiments, we search for the cryptographic keys of applications running in simulated digital crime scenes with a set of predefined system states. We perform a set of tests were all cryptographic applications run in all the relevant system states. The proof of concept tool *Interrogate* is then used to search for the cryptographic keys.

In this section, we describe our testbed setup, the type of cryptographic applications to be targeted and the procedure used for case generation. A set of generic system states is defined, and finally we describe the actual implementation of the cryptographic key search strategies.

Table 1 – Intervals of measured runs of different lengths in the Twofish key schedule.

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

[7] This method does not work with BestCrypt Twofish keys, most likely because BestCrypt uses a slightly different data structure to hold the key or does not utilize the optional MK table. Also please note that neither the AES nor Serpent searches produce any false positives/negatives, but this Twofish method does; we experienced on average 10 false positives/duplicates per search.

5.1. Simulation of digital crime scenes

We choose to utilize the virtualization software VMware Server version 1.0.5 to simulate the digital crime scenes. Through virtualization, a virtual machine runs on top of the host hardware and operating system, and it is possible to run several instances of various guest operating systems. An essential property for the purpose of our research is that the virtual hard drive and physical memory of a VMware machine can be accessed atomically through files on the host hard drive. For a VMware virtual machine, the raw binary contents of the RAM are written to a file named VMname-SnapshotXX.vmem on the host computer in an atomic operation when the snapshot function is triggered.[8] During our experiments, all networking and shared folder support were turned off to isolate the experiments.

5.2. Classes of cryptographic software

Leading practice suggests that cryptographic applications should shred keys and plaintexts from memory as soon as they are no longer needed (Code, 1995). Keys that must reside in memory while the application is running should be purged the moment it terminates. Cryptographic software also need to make sure that keys never are written to disk as a result of virtual memory management. To a forensic investigator, these potential weaknesses of software encryption provides an opportunity to break cryptographically secure ciphers by uncovering their keys.

For the sake of clarity and simplicity, we define three main *classes* of cryptographic software that each of the cryptographic applications tested fall into. The classification of the applications are done according to the expected presence and lifetime of their keys in memory. The three main classes are:

5.2.1. Whole-disk encryption

Applications that provide full disk encryption and other cryptosystems that need to keep their keys in memory while a system is powered on falls within this class. Such applications should feature pre-boot authentication, and should never load cryptographic keys into memory until after the authentication is successfully completed. *TrueCrypt 5.1a* (AES-256, Serpent, Twofish), *BitLocker* (AES-128), *FileVault* (AES-128), *PGP 9.6* (AES-256, Twofish) and *Protectdrive 8.2* (AES-256) were tested as a part of this class.

5.2.2. Virtual disk encryption

Applications that provide file disk encryption as standalone file containers. These applications need to keep keys in memory while mounted, but should immediately upon dismounting or closing wipe its keys. *TrueCrypt 5.1a* (AES-256, Serpent, Twofish), *FileVault* (AES-128),[9] *BestCrypt 8.04.4* (AES-256, Serpent, Twofish), *PGP 9.6* (AES-256, Twofish) and *Protectdrive 8.2* (AES-256) were tested as a part of this class.

5.2.3. Session-based encryption

Applications that generate session or short-lived keys to encrypt session-based information. Some applications may indeed generate a new key for each cryptogram. These applications should wipe keys from memory as soon as a session is closed or the one-time key is used. Typical cryptosystems that falls within this category includes e-mail and instant messaging encryption. *WinZip 11.2* (AES-256 and AES-128), *WinRAR 3.71* (AES-256), *Skype 3.8.0.115* (AES-256) and *Simp Lite MSN 2.2.11* (AES-128) were tested as a part of this class.

Good cryptographic practice suggests that applications in the Whole-Disk and Virtual Disk encryption classes should detect shutdown, screensaver activation and hibernation in time to wipe the keys from memory.

5.3. Definition of system states

A digital forensics investigator may face several different *system states* in a digital crime scene. By categorizing and merging the infinite number of possible states of the modern computer, we define eight states that are decipherable and clarifying to any person encountering a system where cryptography is or has been in use. This is not an exhaustive list of states, but we consider these states to be sufficiently common and generic to be meaningful in the context of estimating the likelihood of successfully finding keys in volatile memory:

The Live State has a logged in user and running cryptosystems. Virtual disks are mounted and session-based cryptography are in progress.

The Screensaver State is a live state with the default Windows screensaver activated due to a 1 min timeout. The screensaver is password protected. The virtual system is immediately suspended using VMware after screensaver activation.

The Dismounted State is a live state with dismounted virtual disks. The virtual system is suspended using VMware immediately after dismounting. Only applicable to Virtual Disk cryptosystems.

The Hibernation State is a state where the system has been put into hibernation mode. The hibernation file is extracted from the system to the host. Not applicable for Whole-disk cryptosystems.

The Terminated State is a terminated state for cryptographic applications. After termination, the virtualized system is immediately suspended using VMware. Not applicable for Whole-disk cryptosystems.

The Logged out State is a live state where the user has logged off after recent activity on the system using the target cryptographic application. Note that this is not identical to a freshly booted system.

The Reboot State is a state where the system is rebooted, but no user actions have yet been performed. This may leave the system in several different sub states as boot prompt, cryptographic pre-boot authentication mechanism or Windows logon screen.

The Boot State is state of freshly booted systems which has been powered off for an extended period of time to ensure that any DRAM remanence effects are ineffectual. VMware automatically wipes the virtual RAM at shutdown, so in our case the virtual machine was restarted immediately.

[8] Note that VMware automatically clears the virtual RAM at virtual machine shutdown.

[9] FileVault falls within both classes as it only encrypts the home folder of the user.

5.4. *Virtualized case generation procedure*

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

1) Windows XP SP2 was installed and updated with all security patches, a snapshot of the clean OS was taken and stored at an external drive.
2) Cryptographic applications were installed and keys were generated.
3) Another snapshot was taken. This snapshot is the basis of our analysis of each cryptographic application.
4) One of the cryptographic applications were utilized together with other software to create a simulated normal operating state.
5) A snapshot was taken, the resulting .vmem memory image was seized, hashed using SHA-256 and analyzed using Interrogate.
6) 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.
7) The system was reverted to the snapshot taken in step 5, and the procedure continued with another snapshot being taken to conserve the new state. We iterated this loop until all system states had been tested.
8) Finally we restored the snapshot from the external hard drive, and repeated from step 3 for each cryptographic application.

5.5. *Real-world testing*

We also setup a testbed for testing on real hardware. The target was a laptop running Windows XP SP3 with TrueCrypt utilizing Serpent in XTS-mode, encrypting the entire disk. To simulate real usage and paging, we instantiated a small application filling up the RAM with random strings, thereby putting the machine under enough stress to commence paging. The memory was then acquired at different points in time using the coldboot method with booting over PXE and analyzed for cryptographic keys. Similarly, we tested OS X FileVault using an Apple MacBook and EFI coldboot.

6. Results

We performed 10 tests per cryptographic application in each available state for the software class. In total, we tested 10 different cryptosystems as listed and categorized in Section 5.2, where several of them were tested with different types of ciphers and modes (whole-disk, virtual disk). The results are summarized[10] in Table 2, where the percentage of keys found are arranged per software class and system state.

To further verify our results, we also performed the tests described in Section 5.5. Memory was then seized after 1, 5 and 10 h (under heavy load) using the coldboot technique, and analyzed for cryptographic keys. We iterated the whole process five times, and in all the 15 cases we were able to recover the keys from the resulting memory image.

Generally we observe that the **Whole-disk** cryptosystems are vulnerable in all states after authentication except **Reboot**. In the states **Live**, **Screensaver** and **Logged out** we were always able to locate both header and master keys. Combined, this creates a large window of opportunity for an adversary to dump and analyze memory. Two of the cryptosystems, PGP and Protectdrive does not purge their keys at reboot, resulting in the possibility of recovering keys from the previous successfully authenticated user even after rebooting (and a 29% hit rate in the **Reboot** state).

For the **Virtual Disk** software class, we found keys in the expected **Live** and **Screensaver** states when the containers are mounted.

A smaller window of opportunity seems to be 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.

We found the key management procedures around hibernation dismounting of virtual disks inadequate, where we found 44% of the expected keys. This may lead to the possibility of uncovering keys from the hibernation file. Protectdrive also fails to purge its keys when the user dismounts, terminates the encryption application or logs out with the disk mounted, resulting in vulnerable **Dismounted**, **Terminated** and **Logged out** states.

Our results indicate that most cryptographic applications feature strong key management. With some exceptions, namely PGP and ProtectDrive, keys were rarely encountered in unexpected states. Especially ProtectDrive seems to practice sloppy key management where up to 14 duplicate keys were found even after uninstallation of the software. Both PGP[11] and SafeNet Inc. has been notified of these findings.

7. Towards forensically sound cryptographic memory forensics

The results clearly indicate that the state of a system at the point of acquisition plays a vital role for an investigator. It is therefore increasingly important to know what to do if a digital crime scene contains a live system using cryptography.

First, upon arriving at a digital crime scene, it is desirable to be able to identify whether cryptography is in use. On-the-fly applications or any of the other **Whole-disk** and **Virtual Disk** encryption systems can be transparent to users and hard to detect. If an investigator fails to detect mounted encrypted volumes on the target computer, potentially crucial digital evidence become unaccessible without the appropriate passwords or cryptographic keys.

[10] Because of the combined number of states, cryptosystems, ciphers, key lengths and software classes, the full results are not printed here. Instead, please be referred to Maartmann-Moe (2008), chapter 7 for details.

[11] PGP reported this finding during our research, and has since the released a new version of PGP (9.9.0, shipped on 25 August 2008) that claims to have fixed this issue.

Table 2 – Experiment results.

State/class	Whole-disk	Virtual disk	Session-based
Live	100%	83%	0%
Screensaver	100%	83%	0%
Dismounted	N/A	11%	N/A
Hibernation	N/A	44%	0%
Terminated	N/A	11%	0%
Logged out	100%	11%	0%
Reboot	29%	11%	0%
Boot	0%	0%	0%

When approaching live systems with cryptographic software, an investigator may choose to copy all data on encrypted drives instead of dumping memory. This may, just as software memory dumping, effect the state of both volatile memory and the hard drives, and thereby compromise data integrity. The risk of evidence tampering should be assessed and compared against the risk of loosing data because of encryption, and this assessment can often be difficult.

We experienced several other challenges that need to be handled by the digital investigator at a case-by-case basis. For example, to dump memory with the coldboot method, the investigator needs to be able to control the boot sequence to load the custom OS. Other acquisition methods have their own challenges as described in (Ruff, 2007). Even though we never experienced bit decay or other difficulties when utilizing the coldboot method, we cannot exclude the possibility of this being owed to the characteristics of the hardware used in the tests.

For an adversary, there are several ways to minimize the risk of memory dumping. The low-hanging fruit is to minimize the time window where an attack is possible by powering off the machine when not used. Other measures that could thwart an investigator could be to restrict boot options and enable BIOS password protection. Physically disabling hardware like FireWire-ports will also restrict the options for memory dumping.

It is important to note that memory analysis and cryptographic key searches are not alternatives to classical digital forensics, but rather *additions* to the existing methods that enable us to acquire as much data as possible. Even though memory acquisition and analysis methods are immature, there are many tools and methods available.

We believe that at the present time, the investigator is faced with a core choice: *To dump memory or not*. Furthermore, we believe that memory dumping should be performed as routinely as disk imaging in any digital forensics investigation, assuming proper tools and training. Failing to dump memory effectively means disregarding a large portion of the digital crime scene, and hereby potential evidence. If cryptography is in use, large amounts of digital evidence may be rendered useless.

8. Conclusions and future work

This paper has attempted 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 at live digital crime scenes. We find the chances of locating encryption keys surprisingly high. Based on the results of this paper, we believe that there's a substantial upside to memory dumping combined with classical digital forensics. The advantages of acquiring memory dumps in an investigation will also likely rise, as the maturity and availability of analysis software increases.

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, although our findings indicate that other states may be vulnerable as well. Cryptographic systems that pre-compute key schedule have all been found to be vulnerable to key schedule searches, adding up to a strong incentive to include memory dumping as part of digital forensics procedures. The outlook for successfully retrieving cryptographic key material is far more dismal when a computing device is turned off, so significant resources should be directed towards the education of digital forensics in the areas of live memory acquisitions.

From a security perspective, the main conclusion is to never leave a computing device powered on unless it is in use or physically protected. The memory disclosure attacks described represent a big threat against both laptops and 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 paper, a skilled attacker can mount attacks against even the strongest software encryption systems.

All acquisitions were performed during or immediately following the execution of cryptographic software. Thus, we have not addressed how long data survives in volatile memory. Furthermore, research is needed on software and hardware based memory dumping and analysis of their impact on system state, including files like pagefile.sys. Finally, where legislation and EULAs allow, efforts on reverse engineering closed-source cryptographic applications are needed to put uncovered keys to good use.

Acknowledgements

We appreciate the steady hand and valuable advice of Professor Svein J. Knapskog at the Norwegian University of Science Technology during the research that lead to this paper. Furthermore, we thank the Norwegian Criminal Investigation Service for providing facilities and resources for the project, and Associate Professor Katrin Franke for helpful advice during the writing of the paper. Finally, we wish to thank the anonymous reviewers for their detailed feedback.

REFERENCES

Anderson R, Biham E, Knudsen L. Serpent: a proposal for the advanced encryption standard. Cambridge University; 2000. Tech. Rep.

Anderson R. Security engineering: a guide to building dependable distributed systems. 1st ed. Wiley; January 2001.
Carrier B, Spafford E. An event-based digital forensic investigation framework. Digital Forensic Research Workshop, 2004.
Carvey H. Windows forensic analysis. Syngress 2007.
Chow J, Pfaff B, Garfinkel T, Rosenblum M. Shredding your garbage: reducing data lifetime. In Proc. 14th USENIX Security Symposium, August 2005.
Crescenzo GD, Ferguson N, Impagliazzo R, Jakobsson M. How to forget a secret. Lecture Notes in Computer Science 1999;1563: 500–9.
Dornseif M. Firewire – all your memory are belong to us. Presentation at CanSecWest/Core05, 2005.
Gutmann P. Secure deletion of data from magnetic and solid-state memory. Proc. 6th USENIX Security Symposium, pp. 77–90, July 1996.
Gutmann P. Data remanence in semiconductor devices. Proc. 10th USENIX Security Symposium, pp. 39–54, August 2001.
Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold boot attacks on encryption keys, 2008.
Kaplan B. Ram is key: extracting disk encryption keys from volatile memory. Master's thesis, Carnegie Mellon University, 2007.
Klein T. All your private keys are belong to us. Tech. Rep., 2006.
Kornblum J. ''Identifying almost identical files using context triggered piecewise hashing''. Digital Investigation 2006;3: 91–7.
Kornblum J. Practical cryptographic key recovery. In Open Memory Forensics Workshop (OMFW), 2008.
Maartmann-Moe C. Forensic key discovery and identification. Master's thesis, Norwegian University of Science and Technology, 2008.
Martin A. Firewire memory dump of a Windows XP computer: a forensic approach. Tech. Rep., 2007.
Mirza F, Murphy S. An observation on the key schedule of Twofish. Information Security Group, Royal Holloway, University of London; 1999. Tech. Rep.
NIST. Announcing the advanced encryption standard (AES). NIST, FIPS 197; 2001.
Pettersson T. Cryptographic key recovery from linux memory dumps. In Chaos Communication Camp, 2007.
Ptacek T. Recover a private key from process memory [online]. Available from: http://www.matasano.com/log/178/recover-a-private-key-from-process-memory/; 2008.
Ruff N. Windows memory forensics. Journal in Computer Virology 2007.
Russinovich ME, Solomon DA. Microsoft windows internals. Microsoft Press; 2005.
Schneier B, Kelsey J, Whiting D, Wagner D, Hall C, Ferguson N. Further observations on the key schedule of Twofish. Twofish Technical Report, 1999.
Schneier B, Kelsey J, Whiting D, Wagner D, Hall C, Ferguson N. Twofish: a 128-bit block cipher. In Third AES Candidate Conference, 2000.
Schneier B. Applied cryptography: protocols, algorithms, and source code in C. 2nd ed. John Wiley & Sons; 1995.
Schneier B. On the Twofish key schedule. Lecture Notes in Computer Science 1998;1556.
Schuster A. Searching for processes and threads in microsoft windows memory dumps. In: DFRWS; 2006.
Shamir A, van Someren N. Playing hide and seek with stored keys. In: Financial cryptography (LNCS 1648). Springer-Verlag; 1998. p. 118–24.
Solomona J, Huebnera E, Bema D, Szezynska M. User data persistence in physical memory. Digital Investigation 2007;4: 68–72.
Truecrypt Foundation. Truecrypt source code [online]. Available from: http://www.truecrypt.org/; 2008.
Walters A, Petroni Jr Nick L. Volatools: integrating volatile memory forensics into the digital investigation process. Komoku, Inc.; 2007. Tech. Rep.