Encrypting Virtual Memory

Niels Provos
Center for Information Technology Integration
University of Michigan
provos@citi.umich.edu

Abstract

In modern operating systems, cryptographic file systems can protect confidential data from unauthorized access. However, once an authorized process has accessed data from a cryptographic file system, the data can appear as plaintext in the unprotected virtual memory backing store, even after system shutdown. The solution described in this paper uses swap encryption for processes in possession of confidential data. Volatile encryption keys are chosen randomly, and remain valid only for short time periods. Invalid encryption keys are deleted, effectively erasing all data that was encrypted with them. The swap encryption system has been implemented for the UVM [7] virtual memory system and its performance is acceptable.

1 Introduction

Many computer systems employ cryptographic file systems, *e.g.* CFS [4], TCFS [6] or encryption layers [19], to protect confidential data from prying eyes. A user without the proper cryptographic key is unable to read the contents of the cryptographic file system, nor is he able to glean any useful information from it. However, backing store of the virtual memory system is generally unprotected. Any data read by a process that was originally encrypted can be found as plaintext in swap storage if the process was swapped out. It is possible for passwords and pass phrases to reside in swap long after they have been typed in, even across reboots.

A user expects that all confidential data vanishes with process termination, and is completely unaware that data can remain on backing store. And even if she were aware of it, there is next to nothing she can do to prevent its exposure.

If the integrity of the operating system is compromised and an untrusted party gains root privileges or physical access to the machine itself, she also gains access to the potentially sensitive data retained in backing store.

Our solution to this problem is to encrypt pages that need to be swapped out. These pages are decrypted when they are brought back into physical memory, *e.g.* due to a page fault. After a process terminates, all its pages stored on backing store are invalid, so there is no need to be able to decrypt them; on the contrary, nobody should be able to decrypt them. This suggests the use of volatile random keys that exist only for short time periods.

The remainder of this paper is organized as follows. Section 2 provides further motivation for encrypting the backing store and describes related work. In Section 3 we give a brief overview of virtual memory, note a security problem of secondary storage, and discuss how it can be resolved with encryption. Section 4 explains how we implemented swap encryption. In Section 5 we analyse how the paging times and system throughput are affected. Finally, we conclude in Section 6.

2 Related Work

Computer systems frequently process data that requires protection from unauthorized users. Often it is enough to use access control mechanisms of the operating system to determine who may access specific data. In many cases a system also needs to be secured against physical attacks or protected against security compromises that allow the circumvention of access controls. Blaze addresses data protection with a cryptographic file system called CFS by encrypting all file system data, preventing anyone without the proper cryptographic key from accessing its content [4]. Anderson, Needham and Shamir aim at hiding the existence of data from an attacker by using a "Steganographic File System" [1]. A cryptographic key and the knowledge that a file exists are needed to access a file's contents. However, security depends on the whole system, and an investigation of the interaction with other system components is essential.

Neither paper looks carefully at its operating environment, nor do they take into consideration that confidential data might inadvertently end up in backing store. The storage of confidential data on a swap device may defeat the purpose of encryption in CFS. Swap data can also be used to reconstruct what files are present in a system, thus defeating the purpose of steganography.

Swap encryption is meant to protect confidential data left on the backing store from intruders who have gained physical access to the storage medium. We observe that the same can be achieved by deleting all confidential data once it is no longer referenced. However, Gutmann has shown that it is difficult to delete thoroughly information from magnetic media or random-access memory [16]. He states: "the easiest way to solve the problem of erasing sensitive information from magnetic media is to ensure that it never gets to the media in the first place. Although not practical for general data, it is often worthwhile to take steps to keep particularly important information such as encryption keys from ever being written to disk."

Schneier and Kelsey describe a secure log system that keeps the contents of the log files confidential even if the system has been compromised [24]. While swap encryption is quite different from secure logging, the attack scenario and operating environment is similar.

There are other systems that modify the paging behavior of a virtual memory system. Notably, Fred Douglis' compression cache compresses memory pages to avoid costly disk accesses [10].

3 Virtual Memory System

One purpose of virtual memory is to increase the size of the address space visible to processes by caching frequently-accessed subsets of the address space in physical memory [2]. Data that does not fit in physical memory is saved on secondary storage known as the backing store. Paged out memory is restored to physical memory when a process needs to access it again [7].

In many operating systems, the virtual memory pager daemon is responsible for reading and writing pages to and from their designated backing store. When a page has been written, it is marked as "clean" and can be evicted from physical memory. The next time a process accesses the virtual memory that was associated with this page, a page fault occurs.

If the page is still resident in physical memory, it is marked as "recently used," and additionally "dirty" if the page fault is caused by a write access. Otherwise, because the page is no longer resident in physical memory, the pager allocates a page of physical memory and retrieves the data from backing store.

3.1 Secondary Storage

Compared to RAM speeds, secondary storage is usually made up from slow media, *e.g.* raw partitions on disk drives. Unlike primary memory, secondary storage is nonvolatile, and the data stored on it is preserved after a system shutdown. Depending on usage patterns, a swap partition can retain data for many months or even years.

Confidential data in a process' address space might be saved on secondary storage and survive there beyond the expectations of a user. She assumes that all confidential data is deleted with the termination of the process. However, the data found by looking at the content of several swap partitions of machines at the Center of Information Technology Integration included: login passwords[1], PGP pass phrases, email messages, cryptographic keys from ssh-agent, shell command histories, URLs, *etc.*

To avoid this, we developed a system that makes data on the backing store impossible for an attacker to read if it was written a certain time prior to the operating system's compromise.

One approach is to avoid swapping completely by not using secondary storage at all. But this is

[1] The author was amazed to find not only his current password, but also older ones that had not been used for months.

not a general solution, and there are many applications and environments that require a virtual address space bigger than the physical memory present in the system.

An application can prevent memory from being swapped out by using the "mlock()" system call to lock the physical pages associated with a virtual address range into memory [16]. There are several disadvantages with this approach. It requires applications to be rewritten to use "mlock()", which might not be possible for legacy applications or difficult if it requires a complicated analysis of which parts of the memory contain confidential data. In addition, "mlock()" reduces the opportunity of the virtual memory system to evict stale pages from physical memory, which can have a severe impact on system performance.

In general, it is not desirable to prevent the system from swapping memory to the disk. Instead, encryption can be used to protect confidential data when it is written to secondary storage by the pager. A user program could install its own encrypting pager [2]. This would lead to greater complexity, require modification of applications and poses difficult decisions about which cryptosystem to use. If a cryptographic file system like CFS [4] were available, the virtual memory pager could be configured to swap to a file that resided on an encrypted file system.

However, in contrast to common use of encryption [20], we require different characteristics for our cryptographic system:

- When a page on backing store is no longer referenced by its owner, the decryption key for that page should be irretrievably lost after a suitable time period (t_R) has passed.
- Only the virtual memory pager should be able to decrypt data read from the backing store.

Clearly, the best protection is achieved with $t_R = 0$. The decryption key, and indirectly the page's content, is irretrievably removed immediately when the page is no longer referenced. This behavior meets the user's expectation that confidential data in a process' address space is deleted with the termination of the process.

However, this is difficult to achieve, and we have to trade off security against performance. Often, a $t_R > 0$ is still acceptable. In the initial implementation, we only guarantee $t_R \leq$ system uptime, but attempt to minimize the average t_R.

This implies the use of volatile encryption keys, valid maximally for the duration of the system's uptime. Such keys are similar to ephemeral keys used to achieve perfect forward secrecy [9]. A volatile key is completely unrelated to all other keys. Knowledge of it does not allow the decryption of old data on secondary storage. Encryption keys are used only by the virtual memory pager and can be generated on demand when they are required, eliminating the need for complicated key management.

On the other hand, swapping to a cryptographic file system does not fulfill either of the two requirements. Key management is an integral part of an encrypting file system [5]. Consequently, permanent nonvolatile encryption keys are present, making it possible to read the data on the swap storage after the system has been shut down. Furthermore, a user with access rights to the swap file on the encrypted file system - usually the root user - can directly read its contents.

Instead, we employ encryption at the pager level. Pages that are swapped out are (optionally) encrypted, and encrypted pages that are read from secondary storage are decrypted.

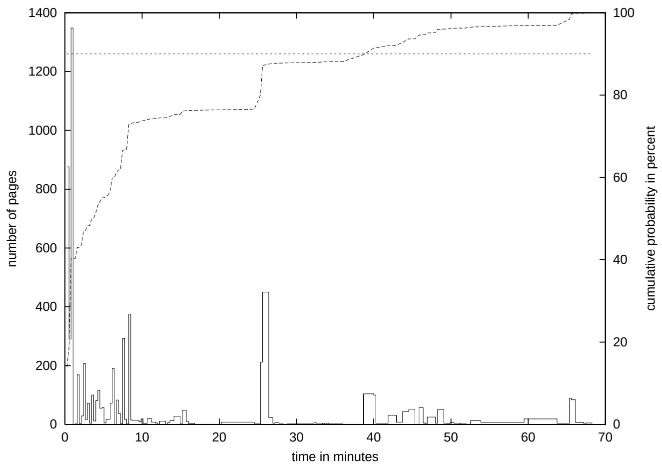

Figure 1: Histogram of page residency in secondary storage for a desktop session and corresponding cumulative probability.

We compared page encryption to zeroing a page on the backing store after it is dereferenced. To get a better understanding of the overheard incurred by such a measure, we recorded how long pages reside on backing store. Figure 1 shows the result for a desktop session.

Most pages remain in the backing store for only a few minutes. The strong temporal correlation between swapping and zeroing can result in unnecessary cleaning of pages that will be overwritten immediately, and will impact on system performance due to expensive write operations. Zeroing pages also fails to protect against physical attacks that prevent writes to secondary storage, *e.g.* an attacker stealing disks or turning off the system's power supply.

In summary, encryption has the following advantages over physically zeroing pages on the backing store.

- Deleting data by erasing it on disk incurs extra seek time and additional I/O for writing. On the other hand, with encryption the content of a page disappears when its respective encryption key is deleted. Furthermore, encrypting a page is fast compared to writing it, and the encryption cost is spread evenly over the whole swapping process.
- Encryption provides better protection against physical attacks. Mere possession of the disk drive is not sufficient to read its content. The correct encryption key is required, but many physical attacks disrupt the operation of the machine; the content of physical memory is lost, and thus also the encryption key. Additionally, encryption prevents "compromising emanations" caused by data transfers to secondary storage, *i.e.* electromagnetic radiation that carries sensitive information and can be received remotely [11].
- Reliably deleting data from magnetic media is difficult, a problem that does not apply when using encryption [16].

In the next section, we describe our implementation of swap encryption.

4 Swap Encryption

Swap encryption divides naturally into two separate functions: encryption and decryption. The former requires a policy decision about when to encrypt pages. The latter requires knowing which pages read from swap need to be decrypted. The encryption policy can be very simple, *e.g.* all pages that go to swap will be encrypted. A more sophisticated policy might encrypt only pages of processes that have read data from a cryptographic file system. The enumeration of such policies is the subject of future work.

In all cases, though, the decryption is completely independent from the decision to encrypt. For that reason, we keep a bitmap in the swap device that indicates for each page whether it needs to be decrypted after it has been read. Thus, it is possible to change the encryption policy during the runtime of the system without affecting the decryption of pages that have been encrypted while a different policy was in effect.

To achieve lower upper bounds on the window of vulnerability (t_R), we divide the backing store into sections of 512 KByte[2], and give each section its own key. A key consists of a 128-bit encryption key, a reference counter and an expiration time. For a backing store of 256 MByte, keys occupy 14 KByte of memory.

A section's 128-bit cryptographic key is created randomly the first time it is needed, and its reference counter is set to 0. Each time a new page is encrypted with it, the counter is incremented.

When a page is freed on the backing store, the reference counter of the respective key is decremented. A key is immediately deleted when the reference counter reaches 0. Thus, all data encrypted with that key can no longer be decrypted and is effectively erased.

At the moment the first page in a section becomes unreferenced, its encryption key is set to expire after a time period t_R. After t_R has been reached, all pages that reference it have to be re-encrypted with a new key. The number of pages that need to be processed is bounded by the section size, so that the additional encryption overhead is configurable.

The framework for expiration exists, but we have yet to implement re-encryption. However, once this has been done, we can make stricter guarantees for the time that pages remain readable on the backing

[2]The section size is configurable, and depends on how much memory is available for cryptographic keys.

store.

Figure 2 describes the paging process in several steps, and shows where encryption and decryption take place:

1. A user process references memory.
2. If the referenced address has a valid mapping, the data is accessed from the mapped physical page.
3. If the referenced address has an invalid mapping, a page fault occurs.
4. The pager reads the corresponding page from secondary storage.
5. The page is decrypted if its entry in the bitmap indicates that it is encrypted.
6. Finally, the page is mapped into physical memory, and the page fault is resolved.
7. Conversely, if the page daemon decides to evict a page from physical memory,
8. the pager encrypts the page with the encryption key of the section that the page belongs to.
   (a) If the section does not have an encryption key, *e.g.* it is the first encryption, a volatile encryption key is initialized from the kernel's entropy pool.
9. Afterwards, the page is written to secondary storage.

There is one central difference between page encryption and decryption. Pages can be decrypted in place because immediately after they have been read into memory, no process is allowed to access these pages until they have been decrypted. On the other hand, even after a page has been swapped out, a process may access it at any time. This precludes in-place encryption. Instead, we have to allocate pages into which to store temporarily the encryption result, placing additional pressure on the already memory limited VM system.

The volatile keys are stored in an unmanaged part of the kernel memory. As a result, they are never paged out.

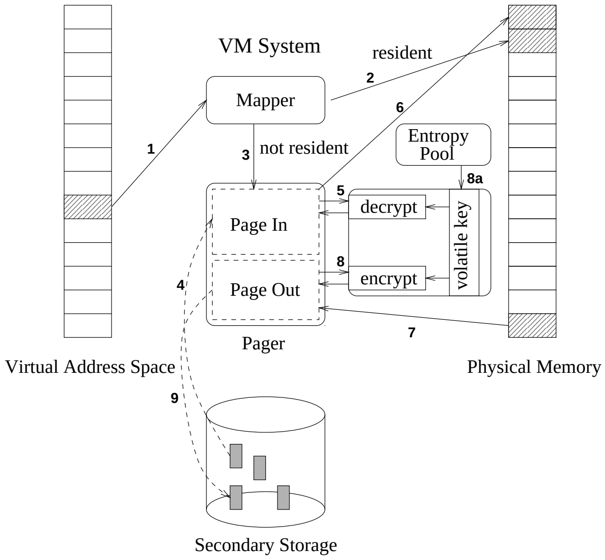

Figure 2: An overview of the swap encryption process.

4.1 Cipher Selection

To be suitable for swap encryption, a cipher needs to fulfill at least three important criteria:

- Encryption and decryption need to be fast compared to disk I/O, so that the encryption does not become the limiting factor in the swapping process.
- The generation of a cipher's key schedule should be inexpensive compared to encrypting a page, so that changing the key schedule does not affect performance. The key schedule of a cipher is usually larger than its encryption key. To conserve system memory we should recompute it every time we switch encryption keys, *e.g.* the encryption key changes when pages are written to different sections.
- The cipher has to support encryption and decryption on a page by page basis, since page in and page out are not sequential. This precludes the use of a stream cipher.

Initially, we planned to employ Schneier's Blowfish encryption algorithm [23]. Its software implementation is very fast, and it has been in use for several years without any apparent security flaws. Nonetheless, Blowfish has one critical drawback. The computation of its key schedule is very expensive, and requires more than 4 KByte of memory.

For that reason, computing the key schedule when it is needed is too expensive, and precomputation is not possible due to large memory requirements.

Based on our environmental constraints, the cipher that matches our needs the best is Rijndael [8]. We describe it in the next section.

4.2 Rijndael

Rijndael is one of the finalists in the advanced encryption standard (AES) competition. It is a variable block and key length cipher. In contrast to many other block ciphers, its round transformation does not have the Feistel structure. Instead, the round transformation is composed of distinct layers: a linear mixing layer, a non-linear layer, and a key addition layer. Rijndael's design tries to achieve resistance against all known attacks while maintaining simplicity [8].

Compared to Blowfish, Rijndael is faster in all aspects, but less studied [12]. We decided to use Rijndael with 128-bit blocks and 128-bit keys. With the optimized C implementation by Gladman [13], the encryption key schedule can be computed in 305 cycles on a Pentium Pro; the decryption key schedule costs 1398 cycles. A block can be encrypted in 374 cycles, and block decryption takes 352 cycles.

However, because all encryption and decryption is done on 4 KByte units, the cost of the key schedule computation is amortized. Therefore, even if we change the key schedule every time, the encryption cost is only 375 cycles on average, and for decryption it is 357 cycles.

Normally, the overall performance of an encryption algorithm is influenced by word conversion to accommodate little and big endian architectures. However, because encryption and decryption happen on the same machine, the word order of the algorithm's output is not relevant, and we do not need to take endianness into consideration.

We use Rijndael in cipher-block chaining (CBC) mode. The CBC mode of operation involves the use of a 128-bit initialization vector. Identical plaintext blocks encrypted under the same key but different IVs, produce different cipher blocks. With $c_0 = IV$, the result of the encryption is defined as

$$c_i = E_K(c_{i-1} \oplus x_i),$$

where the x_i are the plaintext and c_i the ciphertext blocks. The decryption is similar

$$x_i = c_{i-1} \oplus E_K^{-1}(c_i).$$

For swap encryption, the initial 128-bit IV is the 64-bit block number to which the page is written, concatenated with its bitwise complement. This ensures that each page is encrypted uniquely.

Caution is indicated because changing the IV in sequential increments for adjacent pages may result in only small input differences to the encryption function. The attacks described in "From Differential Cryptanalysis to Ciphertext-Only Attacks" [3] might apply in such a situation. For that reason, we encrypt the block number and use that for the IV. Biryukov and Kushilevitz also state, "Another method of IV choice is the encryption of the datagram sequence numbers [...], and sending [the] IV in [the] clear (explicit IV method) [...]. This method is also very vulnerable to our analysis, [...]." Nevertheless, in our case the IV is not explicit, and no IV differences can be observed directly.

4.3 Pseudo-random Generator

To initialize a volatile encryption key we require a source of random bits. The generation of randomness with deterministic computers is very hard. In particular, we do not strive to create perfect randomness characterized by the uniform distribution. Instead, we use pseudo-random generators.

A pseudo-random generator has the goal that its output is computationally indistinguishable from the uniform distribution, while its execution must be feasible [14]. A pseudo-random generator is realized by a stretching function g that maps strings of length n to strings of length $l(n) > n$. If X is a random variable uniformly distributed on strings of length n then $g(X)$ appears to be uniformly distributed on strings of length $l(n)$ [18].

For our purpose, we use the pseudo-random number generator (PRNG) provided by the OpenBSD kernel [21]. The PRNG is a cryptographic stream cipher that uses a source of strong randomness[3] for

[3] The term "source of strong randomness" represents a generator whose output is not really random, but depends on so many entropy providing physical processes that an attacker can not practically predict its output.

initialization and reseeding. This source is referred to as the "entropy pool."

Nonetheless, the problem on how to accumulate strong randomness for the entropy pool remains. Fortunately, a multi-user operating system has many external events from which it can derive some randomness. Gutmann describes a generic framework for a randomness pool [17].

In OpenBSD, the entropy pool

$$P := \{p_1, p_2, \ldots, p_{128}\}$$

consists of 128 32-bit words. To increase the pool's randomness the kernel collects measurements from various physical events: the inter-keypress timing from terminals, the mouse interrupt timing and the reported position of the mouse cursor, the arrival time of network packets, and the finishing time of disk requests.

The measured values from these sources are added to the entropy pool by a mixing function. For each value, the function replaces one word in the pool as follows:

$$p_i \leftarrow u \oplus p_{i+99} \oplus p_{i+59} \oplus p_{i+31} \oplus p_{i+9} \oplus p_{i+7} \oplus p_i,$$

where i is the current position in the pool, and u the 32-bit word that is added. Index addition is modulo 128. After a value has been added i is decremented. To estimate the randomness in the pool, the entropy is measured by a heuristic based on the derivatives of differences in the input values.

A random seed is extracted from the entropy pool as follows: First, the concatenation of $p_1 p_2 \ldots p_{128}$ is given as input to an MD5 hash [22]. Second, the internal state of the MD5 hash for the previous computation is added into the entropy pool. Third, the resulting pool is fed once more into the MD5 hash. Finally, the message digest is calculated. The output is "folded" in half by XOR-ing its upper and lower word. The resulting 64 bits are returned as the seed.

The stretching function is implemented by ARC4, a cipher equivalent to RSADSI's RC4 [25]. The cipher has an internal memory size of $M = n2^n + 2n$, with in our case $n = 8$. We use the random seeds extracted from the entropy pool to initialize the M bits. The output of RC4 is expected to cycle after 2^{M-1} iterations. However, Golić showed that a correlation between the second binary derivative of the least significant bit output sequence and 1 can be detected in significantly fewer iterations [15], which allows the differentiation of $RC4$ from a uniform distribution. We can avoid this problem by reseeding RC4's internal state before the number of critical iterations has been reached. In fact, the implementation in OpenBSD reseeds the ARC4 every time enough new entropy has been accumulated.

The kernel provides the "arc4random(3)" function to obtain a 32-bit word from the pseudo-random number generator.

The volatile key of a section is created by filling it with the output from "arc4random(3)." We hope that between the time the system has been booted and the first swap encryption sufficient randomness is available in the kernel entropy pool to ensure good randomness in the RC4 output. Nonetheless, it should be noted that this construction does not create a provably pseudo-random generator as described in the beginning of this section.

5 Performance Evaluation

In the following, we analyse the effect of swap encryption on the paging behavior. We look at page encryption and decryption times, and assess the runtime of applications with large working sets.

All measurements were performed on an OpenBSD 2.6 system with 128 MByte main memory and a 333 MHz Celeron processor. The swap partition was on a 6 GByte Ultra-DMA IDE disk, IBM model DBCA-206480 running at 4200 revolutions per minute. The operating system can sustain an average block write rate of 7.5 MByte/s and a block read rate of 6.3 MByte/s. OpenBSD uses the UVM [7] virtual memory system.

5.1 Micro Benchmark

Our micro benchmark measures the time it takes to encrypt one page. A test program allocates 200 MByte of memory, and fills the memory sequentially with zeros. Afterwards, it reads the allocated memory from the beginning in sequential order. The process is repeated three times.

We use kernel profiling to measure page encryption frequency, and the cumulative time of the encryption function. The kernel function "swap_encrypt()" is called 155336 times with a cumulative running time of 67.96 seconds. One 4 KByte page could be encrypted in 0.44 ms, resulting in an encryption bandwidth of 8.9 MByte/s. The total amount of memory encrypted is 600 MByte.

In UVM, writes to the backing store are asynchronous and reads are synchronous. To determine if I/O is still the bottleneck of the swapping process, we measured the runtime of the test program for different memory sizes, with and without swap encryption. We measure an increase in runtime of about 14% with encryption. To measure asynchronous writes, we modified the test program to write only to memory. The runtime increase of 26% - 36% is due to allocation of new pages that store the encrypted pages until they are written to disk, thus causing the system to swap more often. Figure 3 shows a graph of the results.

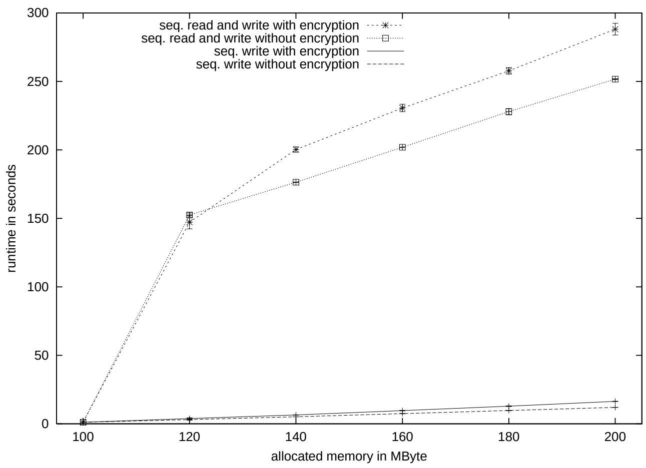

Figure 3: Performance difference between swap encryption and normal swapping when pages are accessed sequentially, illustrating the difference between asynchronous write and synchronous reads.

5.2 Macro Benchmark

To judge the impact of swap encryption on application programs, we used ImageMagick to process a 960 × 1280 image with a 16-bit colorspace. The image was magnified and then rotated by 24^o. The runtimes for different magnification factors are shown in Table 1.

	No Encryption		Encryption	
Magnification	Major Faults	Runtime (in sec)	Major Faults	Runtime (in sec)
2.30×	0.4 10^3	49s	0.4 10^3	49s
2.35×	19 10^3	145s	18 10^3	147s
2.40×	22 10^3	169s	22 10^3	180s
2.50×	24 10^3	179s	24 10^3	276s

Table 1: Runtime of image processing tool for different magnification factors.

The table compares the major faults and program runtime for a system that does not use encryption against a system that does. A major fault is a page fault that requires I/O to service it, and does not take into account the pages that have been paged out by the paging daemon.

With increasing magnification factor, the working set size of the program grows larger. We measure a sharp increase of the running time with swap encryption for a magnification factor of 2.5. However, for the other magnification factors the program runtime is not affected that much, even though nearly half of the program's memory was on backing store. Thus, we believe that the overhead caused by encryption is tolerable.

6 Conclusion

Confidential data can remain on backing store long after the process to which the data originally belonged has terminated. This is contrary to a user's expectations that all confidential data is deleted with the termination of the process. An investigation of secondary storage of machines at the Center for Information Technology Integration revealed very confidential information, such as the author's PGP pass phrase.

We investigate several alternative solutions to prevent confidential data from remaining on backing store, *e.g.* erasing data physically from the backing store after pages on it become unreferenced. However, we find that encryption of data on the backing store with volatile random keys has several advantages over other approaches:

- The content of a page disappears when its respective encryption key is deleted, a very fast operation.

- Encryption provides protection against physical attacks, *e.g.* an attacker stealing the disk that contains the swap partition

Encryption enables us to make the guarantee that unreferenced pages on the backing store become unreadable after a suitable time period upper bounded by system uptime has passed.

We have demonstrated that the performance of our encryption system is acceptable, and it proves to be a viable solution.

The software is freely available as part of the OpenBSD operating sytem and can also be obtained by contacting the author.

7 Acknowledgments

I thank Patrick McDaniel and my advisor Peter Honeyman for careful reviews and helpful comments on the organization of this paper. I also thank Chuck Lever for getting me interested in swap encryption, Artur Grabowski for improving my understanding of UVM and David Wagner for helpful feedback on cipher selection.

References

[1] R. Anderson, R. Needham, and A. Shamir. The Steganographic File System. In *Proceedings of the Information Hiding Workshop*, April 1998.

[2] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] Alex Biryukov and Eyal Kushilevitz. From Differential Cryptanalysis to Ciphertext-Only Attacks. In *Proceedings of the Advances in Cryptology — CRYPTO '98*, pages 72–88. Springer-Verlag, August 1998.

[4] Matt Blaze. A Cryptographic Filesystem for Unix. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 9–16, November 1993.

[5] Matt Blaze. Key Management in an Encrypting File System. In *Proceedings of the 1994 USENIX Summer Technical Conference*, pages 27–35, June 1994.

[6] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic Filesystem for Unix. Unpublished Technical Report, July 1997. `ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz`.

[7] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 117–130, June 1999.

[8] Joaen Daemen and Vincent Rijmen. AES Proposal: Rijndael. AES submission, June 1998. `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

[9] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[10] Fred Douglis. The Compression Cache: Using On-Line Compression to Extend Physial Memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, 1993.

[11] Berke Durak. Hidden Data Transmission by Controlling Electromagnetic Emanations of Computers. Webpage. `http://altern.org/berke/tempest/`.

[12] Niels Ferguson, John Kelsey, Mike Stay, David Wagner, and Bruce Schneier. Improved Cryptanalysis of Rijndael. In *Fast Software Encryption Workshop 2000*, April 2000.

[13] Brian Gladman. AES Algorithm Efficiency. Webpage. `http://www.btinternet.com/~brian.gladman/cryptography_technology/aes/index.html`.

[14] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness.* Springer-Verlag, 1999.

[15] Jovan Dj. Golić. Linear Statistical Weakness of Alleged RC4 Keystream Generator. In *Proceedings of the Advances in Cryptology — Eurocrypt '97*, pages 226–238. Springer-Verlag, May 1997.

[16] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX Security Symposium*, pages 77–89, July 1996.

[17] Peter Gutmann. Software Generation of Practially Strong Random Numbers. In *Proceedings of the Seventh USENIX Security Symposium*, pages 243–255, June 1998.

[18] J. Hastad, R. Impagliazzo, L. Levin, and M. Luby. Construction of Pseudorandom Generator from any One-Way Function, 1993.

[19] J. Heidemann and G. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[20] Maurice P. Herlihy and J. D. Tygar. How to Make Replicated Data Secure. In *Proceedings of the Advances in Cryptology - CRYPTO '87*, pages 379–391. Springer-Verlag, 1988.

[21] Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 1999.

[22] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992.

[23] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[24] Bruce Schneier and John Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the Seventh USENIX Security Symposium*, pages 53–62, January 1998.

[25] RSA Data Security. The RC4 Encryption Algorithm, March 1992.