Protecting Cryptographic Keys From Memory Disclosure Attacks

Keith Harrison and Shouhuai Xu
Department of Computer Science, University of Texas at San Antonio
{kharriso,shxu}@cs.utsa.edu

Abstract

Cryptography has become an indispensable mechanism for securing systems, communications and applications. While offering strong protection, cryptography makes the assumption that cryptographic keys are kept absolutely secret. In general this assumption is very difficult to guarantee in real life because computers may be compromised relatively easily. In this paper we investigate a class of attacks, which exploit memory disclosure vulnerabilities *to expose cryptographic keys. We demonstrate that the threat is real by formulating an attack that exposed the private key of an OpenSSH server within* **1** *minute, and exposed the private key of an Apache HTTP server within* **5** *minutes. We propose a set of techniques to address such attacks. Experimental results show that our techniques are efficient (i.e., imposing no performance penalty) and effective — unless a large portion of* allocated *memory is disclosed.*

Keywords: cryptographic key security, memory disclosure.

1 Introduction

The utility of cryptography is based on the assumption that cryptographic keys are kept absolutely secret. This assumption is very difficult to guarantee in real-life systems due to various software bugs in operating systems and applications. In this paper we focus on a class of attacks that exploit memory disclosure vulnerabilities, called *memory disclosure attacks*. Such an attack can expose the content of (a portion of) computer memory, and thus cryptographic keys in the disclosed memory.

Our contributions. First, we thoroughly assess (Section 2) the damage of memory disclosure attacks against the private keys of OpenSSH servers and Apache HTTP servers. The attacks exploit two reported vulnerabilities. Our experiments show that such attacks effectively expose the RSA private keys of the servers.

Second, we propose a method (Section 3) for helping understand the attacks (e.g., why are they so powerful?). The core of the method is a software tool we developed to help analyze the content of computer memory. Through our software tool, we found that disclosure a portion of either *allocated memory* or *unallocated memory* would effectively expose cryptographic keys. This is interesting because existing literature often emphasized the importance of clearing unallocated memory (cf. Viega et al. [18, 19] and Chow et al. [6]), but not necessarily taking care of allocated memory.

Third, our analyses on the attacks suggest that one should ensure (i) a cryptographic key only appears in allocated memory a minimal number of times (e.g., one), and (ii) unallocated memory does not have a copy of cryptographic keys. We thus proceed to propose a set of concrete solutions. In particular, our method for minimizing the number of copies of a private key in allocated memory, to our knowledge, is novel in the sense that it takes full advantage of the operating system "copy on write" memory management policy [17] – a technique that was not originally motivated for security purpose. We conduct case studies by applying our solutions to protect the private keys of OpenSSH servers and of Apache HTTP servers. Experimental results show that our solutions can eliminate attacks that disclose unallocated memory, and can mitigate the damage due to attacks that disclose a small portion of allocated memory. It is stressed, however, that if the portion of disclosed memory is large (e.g., about 50% as shown in our case study), the key is still exposed in spite of the fact that our solutions can minimize the number of key copies in memory. Therefore, our investigation may serve as an evidence that in order to completely avoid key exposures due to memory disclosures, special hardware is necessary.

Related work. The problem of ensuring the secrecy of cryptographic keys (and their functionalities thereof) has been extensively investigated by the cryptography community. There have been many novel cryptographic methods that can mitigate the damage caused by the compromise of cryptographic keys. Notable results include the notions of threshold cryptosystems [8], proactive cryptosystems [15], forward-secure cryptosystems [1, 2, 11], key-insulated cryptosystems [9], and intrusion-resilient cryptosystems [12]. The present paper falls into an approach that is orthogonal to the cryptographic approach. Clearly,

our mechanisms can be deployed to secure traditional cryptosystems, as evidently shown in this paper. Equally, our mechanisms can be utilized to provide another layer of protection for the afore-mentioned advanced cryptosystems.

It has been deemed as a good practice in developing secure software to clear the sensitive data such as cryptographic keys, promptly after use (cf. Viega et al. [18, 19]). Unfortunately, as confirmed by our experiments as well as an earlier one due to Chow et al. [5], this practice has not been widely or effectively enforced. Chow et al. [5] investigated the propagation of sensitive data within an operating system by examining all places the sensitive data can reside. Their investigation was based on whole-system simulation via a hardware simulator, namely the open-source IA-32 simulator Bochs v2.0.2 [3]. More recently, Chow et al. [6] presented a strategy for reducing the lifetime of sensitive data in memory called "secure deallocation," whereby data is erased either at deallocation or within a short, predictable period afterwards in general system allocators. As a result, their solution can successfully eliminate attacks that disclose unallocated memory. However, their solution has no effect in countering attacks that may disclose portions of allocated memory. Whereas, our solutions can not only eliminate attacks that disclose unallocated memory, but also mitigate the damage due to attacks that disclose a small portion of allocated memory. That is, our solutions provide strictly better protections.

There is some loosely related work. Broadwell et al. [4] explored the core dump problem to infer which data is sensitive based on programmer annotations, so as to facilitate the shipment of crash dumps to application developers without revealing users' sensitive data. Provos [16] investigated a solution to use swap encryption for processes in possession of confidential data. A cryptographic treatment on securely erasing sensitive data via a small erasable memory was presented by Jakobsson et al. [7].

Outline. In Section 2 we evaluate the severity of the memory disclosure problem. In Section 3 we show how to understand the attacks in detail based on our software tool. In Section 4 we present a set of solutions to countering memory disclosure attacks, whose concrete instantiations to protect private keys of OpenSSH servers are explored in Section 5. We conclude the paper in Section 6. Due to the space limitation, we defer many details (including the treatment on Apache HTTP servers) to the full version of the present paper [13].

2 Threat Assessment: Initial Experiments

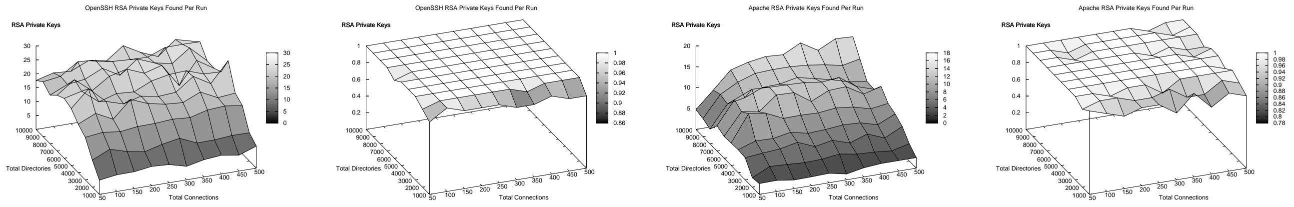
In this section we report our experiments that exploit two specific memory disclosure vulnerabilities to expose the RSA private keys of an OpenSSH Server and of an Apache HTTP server. The first vulnerability was reported in [14], which states that Linux kernels prior to 2.6.12 and prior to 2.4.30 are vulnerable to the following attack: directories created in the `ext2` file systems could leak up to 4072 bytes of (unallocated) kernel memory for every directory created. The second vulnerability was reported in [10], which states that a portion of memory of Linux kernels prior to 2.6.11 may be disclosed due to the misuse of signed types within `drivers/char/n_tty.c`. The disclosed memory may have a random location and may be of a random amount. Both vulnerabilities can be exploited *without* requiring the `root` privilege.

Recall that the RSA cryptosystem has a public key (e, N) and a private key (d, N), where $N = PQ$ for some large prime P and Q. In practice, a variation of the Chinese Remainder Theorem (CRT) is utilized to speed up the signing/decryption procedure, meaning that a RSA private key actually consists of 6 distinct parts: d, P, Q, $d \mod (P-1)$, $d \mod (Q-1)$, and $Q^{-1} \mod P$. Notice that there is a special PEM-encoded private key file, which contains the whole private key. For simplicity, we only consider d, P, Q, and the PEM-encoded file because disclose of any of them immediately leads to the compromise of the private key. Therefore, we call any appearance of any of them "a copy of the private key."

Our experiments ran in the following setting: the server machine has a 3.2GHz Intel Pentium 4 CPU and 256MB memory; the operating system is Gentoo Linux with a 2.6.10 Linux kernel; the OpenSSH server is OpenSSH 4.3_p2; the Apache HTTP server is Apache 2.0.55 (compiled using the `prefork` MPM); the OpenSSL library version is 0.9.7i.

On the power of attacks exploiting the vulnerability reported in [14]. Our experimental attacks proceeded as follows. (i) We plugged a small 16MB USB storage device into the computer running OpenSSH (or Apache HTTP) server. (ii) We wrote a script to fulfill the following. In the case of OpenSSH server, it first created a large number of SSH connections to `localhost`; whereas in the case of Apache HTTP server, it first instructed a remote client machine to create a large number of HTTP connections to the server. Then, the script immediately closed all connections. Finally, the script created a large number of directories on the USB device, where each directory created revealed less than 4,072 bytes of memory onto the USB device. (iii) We removed the USB device, and then simply searched the USB device for copies of the private key. Experimental results are summarized as follows.

The case of OpenSSH server: Figure 1(a) depicts the average (over 15 attacks) number of copies of private keys found from the disclosed memory on the USB device, with respect to the number of `localhost` SSH connections (the x-axis) and the number of created directories (the y-axis). For example, by establishing 500 total connections



(a) OpenSSH: # of key copies found (b) OpenSSH: success rate of attacks (c) Apache: # of key copies found (d) Apache: success rate of attacks

Figure 1. OpenSSH vs. Apache with respect to the vulnerability reported in [14]

and creating 1,000 directories (i.e., disclosing up to about 4 MBytes memory), we were able to recover about 8 copies of the private key. From a different perspective, Figure 1(b) depicts the average success rate of attacks (i.e., the rate of the number of successful attacks over the total number of 15 attacks), which clearly states that an attack almost always succeeds. In this case, an attack took *less than 1 minute*.

The case of Apache HTTP server: Figure 1(c) shows the average (over 15 attacks) number of copies of private keys found on the USB device, with respect to the number of connections (the x-axis) and the number of created directories (the y-axis). For example, by establishing 500 connections and creating 1,000 directories (i.e., disclosing up to 4 MBytes memory), we were able to recover about 5 copies of the private key. From a different perspective, Figure 1(d) depicts the average success rate of attacks, which clearly states that an attack almost always succeeds. In this case, an attack took *less than 5 minutes*.

On the power of attacks exploiting the vulnerability reported in [10]. Our experimental attack was orchestrated by a script that fulfills the following: (i) In the case of OpenSSH server, it created a large number of SSH connections to `localhost`. In the case of Apache HTTP server, it instructed a remote computer to establish a large number of HTTP connections to the server. (ii) The script executed a program (due to [10]) to dump a piece of memory to a file, which was then searched for the private key. The size and location of the disclosed memory varied, dependent on the terminal running the exploit. The exploit disclosed about 50% of the memory (i.e., 128 MBytes) on average. Experimental results are summarized as follows.

The case of OpenSSH server: Figure 2(a) shows the average (over 20 attacks) number of copies of private keys found in the disclosed memory with respect to the number of connections (the x-axis). From a different perspective, Figure 2(b) shows the success rate of attacks (i.e., the rate of the number of successful attacks over the total number of 20 attacks) with respect to the number of connections (the x-axis). It is clear that an attack almost always succeeds. Moreover, an attack took *less than 1 minute*.

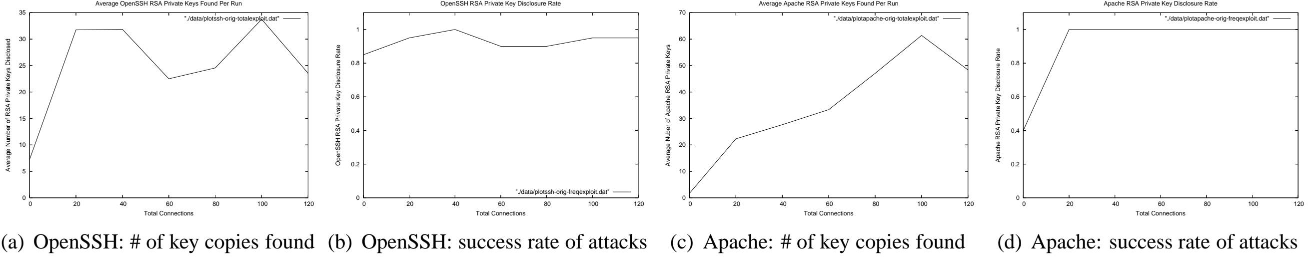
The case of Apache HTTP server: Figure 2(c) shows the average (over 20 attacks) number of private keys found in the disclosed memory with respect to the number of connections (the x-axis). Figure 2(d) shows the success rate of attacks, which clearly states that an attack always succeeds when 30 or more connections are established. In this case, an attack took *less than 1 minute*.

In summary, our experiments showed that cryptographic keys can be easily compromised by attacks that exploit memory disclosure vulnerabilities. Since the attacks are so powerful, we suspect that copies of the cryptographic keys were somehow flooding the memory to some extent. This motivates our thorough examination in Section 3.

3 Understanding the Attacks

Supporting tool: locating cryptographic keys in memory. In order to understand the attacks, we needed a tool to capture the "snapshots" of memory, and to bookkeep information such as "which processes have access to which memory pages that contain copies of private keys". We developed a software tool for this purpose. The C code of our tool is about 260 lines, and is implemented as a loadable kernel module (LKM). The detail of the code is deferred to the full version of this paper [13]. In our experiments, it took about 5 seconds to scan the 256MB memory.

Understanding the attacks: the case of OpenSSH server. Equipped with our software tool, we conducted another experiment with the same hardware and software setting as in our experiments mentioned above, except that the operating system was replaced by Gentoo Linux with a 2.6.16.1 Linux kernel. The intent of experimenting with a newer version of operating system, which was *not* known to be subject to the afore-mentioned two vulnerabilities, was to validate whether the suspected phenomenon is still relevant in newer operating systems. Note that the new experiments are run with the root privilege, whereas the above attack experiments are not. Specifically, we let two other machines act as clients for issuing SSH requests to the server via a 100Mb/s switch network. We wrote a Perl script to automatically trigger events at the following predefined points in time (unit: 2 minutes).

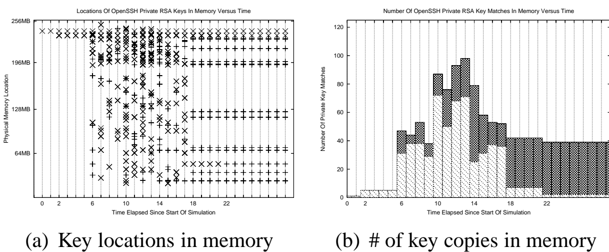


(a) OpenSSH: # of key copies found (b) OpenSSH: success rate of attacks (c) Apache: # of key copies found (d) Apache: success rate of attacks

Figure 2. OpenSSH vs. Apache with respect to the vulnerability reported in [10]

- Time t=0: The simulation is started without OpenSSH running.
- Time t=2: The OpenSSH server is started via the command `/etc/init.d/sshd start`.
- Time t=6: The first client machine begins issuing SSH requests and maintains 8 concurrent `scp` transfers. Each transfer lasts about 4 seconds.
- Time t=10: The second client machine initiates an additional 8 concurrent `scp` transfers. This brings the number of concurrent connections to 16 in total.
- Time t=14: The first client machine stops all file transfers. This reduces the total number of concurrent file transfers to 8.
- Time t=18: The second client machine stops all file transfers, and thus all network traffic ceased.
- Time t=22: The OpenSSH server is stopped via the command `/etc/init.d/sshd stop`.
- Time t=29: The experiment is finished.

Corresponding to the above events, outputs of the LKM are plotted in Figures 3(a) and 3(b) from two different perspectives. Both pictures have the time as the x-axis.



(a) Key locations in memory (b) # of key copies in memory

Figure 3. OpenSSH case

Figure 3(a) shows the locations of copies of the private key in memory, where "$\times$" represents a copy in allocated user or kernel space, and "$+$" represents a copy in unallocated memory. From this picture we draw the following observations. (1) The OpenSSH private key is in memory at time t=0, even though the OpenSSH server is not started until time t=2. This is because the PEM-encoded file has been loaded into memory by the Reiser file system. (2) When the OpenSSH server is started at time t=2, the newly appearing $\times$'s are actually the d, P, and Q of the private key. (3) When OpenSSH client requests begin to be issued at time t=6, the number of copies of the private key increases abruptly. We also begin to see copies of the private key in unallocated memory. (4) When the client machines stop issuing requests at time t=18, the number of copies of the private key in allocated memory drops abruptly. We also observe that many copies of the private key are not erased before entering unallocated memory. (5) When the OpenSSH server stops at time t=22, d, P and Q exist only in unallocated memory, except the PEM-encoded private key file that remains in the Linux kernel's page cache.

Figure 3(b) shows the total number of copies of the private key in memory, where lightly shaded bars correspond to copies of the private key in allocated memory, and dark shaded bars correspond to copies of the private key in unallocated memory.

Summary. In both OpenSSH and Apache HTTP servers (details of the Apache case is deferred to [13]), many copies of the private key can be found in both allocated memory and unallocated memory. This confirms our suspicion that copies of cryptographic keys somewhat flooded in memory when the number of SSH / HTTP connections increases – even in newer operating systems, and explains why the afore-experimented attacks were so powerful.

4 Countering Memory Disclosure Attacks

Analyses in the last section naturally suggest the following countermeasures: We should ensure (i) a cryptographic key only appears in allocated memory a minimal number of times (e.g., one) as long as this does not downgrade system performance, and (ii) unallocated memory (or any other place with a disclosure potential such as swap space) does not have a copy of a cryptographic key. For this purpose, now we present a set of solutions at different layers, from

application down to operating system kernel.

Application level solution: First, utilize the "copy on write" memory management policy [17] to avoid unnecessary duplications of cryptographic keys. Specifically, we propose placing the private key into a special memory region, and guaranteeing that no process will write to that memory region. This ensures that the private key will only exist once in physical memory (in addition to the PEM-encoded private key file), no matter how many processes are forked. Second, avoid appearances of cryptographic keys as follows: (1) Ensure the private key is not explicitly copied by the application or any involved libraries. (2) Disable swapping of the memory that contains the key using the appropriate system calls. This is because when memory is swapped to disk, the memory is not immediately cleared and the private key may appear in unallocated memory.

Library level solution: We suggest eliminating unnecessary duplications of cryptographic keys in allocated memory using the same measures suggested in application level solution. This suffices to prevent private keys from appearing in memory other than the afore-mentioned special region and the PEM-encoded private key file.

Kernel level solution: We propose ensuring that the unallocated memory does not contain any private keys. This can be fulfilled by having the kernel zero any physical pages before they become unallocated.

Integrated library-kernel solution: We propose integrating the library and kernel level mechanisms together to obtain strictly stronger protection. This way, unnecessary duplications of private keys in allocated memory and any appearances of private keys in unallocated memory are simultaneously eliminated. Moreover, this solution can even remove the PEM-encoded private key from allocated memory, provided that the library instructs the kernel not to cache the PEM-encoded private key file. Therefore, whenever possible, this solution should be adopted.

5 Protecting Keys of OpenSSH Servers

Implementing the application level solution. We instantiate the above general solution with a function, `RSA_memory_align()`, which should be called as soon as OpenSSL's `RSA` data structure contains the private key. This ensures that exactly one copy of the private key appears in allocated memory, in addition to the PEM-encoded file. Notice that we need to start OpenSSH with the undocumented `-r` option to prevent the OpenSSH server from re-executing itself after every incoming connection.

Specifically, `RSA_memory_align()` can be characterized as follows. (1) It takes advantage of the "copy on write" memory management policy as follows. First, it uses `posix_memalign()` to request one or more memory pages for fulfilling the afore-mentioned special memory region. Then, it copies the private key into the special memory region, and zeros and frees the memory originally containing the private key. Then, it updates the pointers in the `RSA` data structure to point to the new location of the private key. Finally, it sets the `BN_FLG_STATIC_DATA` flag to inform OpenSSL that the private key is now exclusively located at the special region. (2) It prevents OpenSSL's `RSA_eay_mod_exp()` from caching the private key by unsetting the `RSA_FLAG_CACHE_PRIVATE` flag in the `flags` member of the associated `RSA` data structure. Moreover, `RSA_memory_align()` disables swapping of memory that contains the private key by calling `mlock()` on the memory allocated by `posix_memalign()`.

Implementing the library level solution. We modify the OpenSSL function `d2i_PrivateKey()`, which is responsible for translating a PEM-encoded private key file into the RSA key parts by calling `d2i_RSAPrivateKey()`. The modification is that when the `d2i_RSAPrivateKey()` method returns, we immediately call the function `RSA_memory_align()` mentioned above.

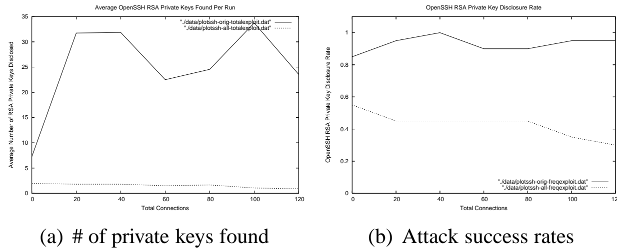
Implementing the kernel level solution. We modify the kernel function `free_hot_cold_page()` to enforce that memory pages are cleared, via `clear_highpage()`, before they are added to one of the lists of free pages. Thus, no private key will appear in unallocated memory.

Implementing the integrated library-kernel solution. In addition to the modifications made in the library level solution and in the kernel level solution mentioned above, the PEM-encoded private key file can be removed from allocated memory. In order to do this, we introduce a new flag, `O_NOCACHE`, to allow an application to instruct the kernel to immediately remove this file from the "page cache". Specifically this is implemented as follows. Whenever the PEM-encoded private key file is read, the kernel gives the file contents to the requester and then checks if the `O_NOCACHE` flag is specified. If so, the kernel immediately deletes the corresponding "page cache" entry by calling `remove_from_page_cache()` before calling `free_page()`.

Experimental results. First, we re-examined the attack based on [14] against the same vulnerable 2.6.10 Linux Kernel, except that the system is now patched with our respective solutions. In no case were we able to recover any copy of the private key.

Second, we re-examined the attack based on [10] against the same vulnerable 2.6.10 Linux kernel, except that the system is now patched with our solutions. For conciseness, we only consider our integrated library-kernel solution. Figure 4(a) compares the average (over 20 attacks) number of

copies of the private key found in the USB device *before* and *after* deploying our solution. It shows that the number of copies of the private key recovered is reduced by our solution because only one copy of the private key appears in allocated memory, and no copies of the private key appear in unallocated memory. Figure 4(b) compares the success rate of attacks *before* and *after* deploying our integrated library-kernel solution. While our solution does reduce the attack success rate (from about 90% to about 50%), the attack still succeeds with a probability about 50% because the attack discloses on average about 50% of the memory. Thus, as mentioned before, completely eliminating such powerful attacks might have to resort to some special hardware devices.



(a) # of private keys found

(b) Attack success rates

Figure 4. OpenSSH effect of our library-kernel solution countering the attack of [10]

As we show in [13] our solution does not impose any performance penalty.

6 Conclusion

We investigated a set of mechanisms to deal with the exposure of cryptographic keys caused by memory disclosure attacks. Our mechanisms can eliminate attacks that disclose unallocated memory, and can mitigate the damage due to attacks that disclose a small portion of allocated memory. Our result suggests that in order to eliminate powerful attacks that can disclose a large portion of memory, one may have to resort to special hardware devices.

Acknowledgement. We thank the anonymous reviewers for their valuable comments, and our shepherd, Luigi Romano, for his constructive suggestions that improved the paper.

This work was supported in part by ARO, NSF and UTSA.

References

[1] R. Anderson. On the forward security of digital signatures. Technical report, 1997.

[2] M. Bellare and S. Miner. A forward-secure digital signature scheme. In *Proc. Crypto'99*, pages 431–448.

[3] Bochs. the bochs ia-32 emulator project. `http://bochs.sourceforge.net/`.

[4] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Usenix Security Symposium'03*.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Usenix Security Symposium'04*.

[6] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime. In *Proc.USENIX Security Symposium'05*.

[7] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to forget a secret. In *STACS'99*.

[8] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proc. Crypto'89*, pages 307–315.

[9] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Proc. EUROCRYPT'02*.

[10] Georgi Guninski. linux kernel 2.6 fun. windoze is a joke. `http://www.guninski.com/where_do_you_want_billg_to_go_today_3.html` (dated 15 February 2005).

[11] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Crypto'01*.

[12] G. Itkis and L. Reyzin. Sibir: Signer-base intrusion-resilient signatures. In *Crypto'02*.

[13] K. Harrison and S. Xu. Full version of the present paper available at `www.cs.utsa.edu/~shxu`.

[14] Mathieu Lafon and Romain Francoise. Information leak in the linux kernel ext2 implementation. `http://arkoon.net/advisories/ext2-make-empty-leak.txt` (Arkoon Security Team Advisory - dated March 25, 2005).

[15] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC'91*.

[16] N. Provos. Encrypting virtual memory. In *Proc. Usenix Security Symposium'00*.

[17] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts (sixth ed.)*. John Wiley & Sons.

[18] J. Viega. Protecting sensitive data in memory. `http://www.cgisecurity.com/lib/protecting-sensitive-data.html`, 2001.

[19] J. Viega and G. McGraw. *Building Secure Software*. Addison Wesley, 2002.