# Protecting Cryptographic Keys from Memory Disclosure Attacks[1]

Giovanni Del Valle, T. Paul Parker, Keith Harrison, and Shouhuai Xu
University of Texas at San Antonio

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 secret. This assumption is very difficult to guarantee in real life because computers, on which cryptographic keys are stored and utilized, may be compromised relatively easily. Moreover, compromise of cryptographic keys may not be detected (and therefore the compromised keys being revoked) until after a long period of time. In this paper we investigate memory disclosure attacks, which exploit memory disclosure vulnerabilities to expose some amount of computer memory (RAM) and thus cryptographic keys. We demonstrate that the threat is real by formulating attacks that exposed the private keys of an OpenSSH server and an Apache server in their entirety. The attack experiments demonstrate that the private keys are somewhat flooding in RAM. We explain this phenomenon by showing that private keys are not carefully dealt with in the software stack, which motivates us to propose a set of software-based countermeasures that can effectively mitigate the damage of memory disclosure attacks at essentially no performance penalty. We also report lessons learned through this study, which should be taken into consideration in the design and development of future systems that have cryptographic components.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: cryptographic key, OS security, memory disclosure

## 1. INTRODUCTION

The utility of cryptography is based on the assumption that cryptographic keys are kept secret. This assumption is very difficult to guarantee in real-life systems because an attacker can exploit various vulnerabilities in the software stack to expose them. Despite recent progress in designing cryptosystems that can tolerate *bounded* exposure of partial information about cryptographic keys, the threat of software attacks against cryptographic keys will remain relevant because of the following. First, it may take a long time before such special cryptosystems are deployed in real life, and the currently deployed cryptosystems will continue to be used for a long time. Second, it is not easy in practice to estimate the non-trivial bound of partial information exposure about cyptographic keys. Third, as we show in this paper, attacks such as memory disclosure ones (if successful) will expose

[1]This paper extends [Harrison and Xu 2007].

Author's address: The authors are with the Department of Computer Science, University of Texas at San Antonio. {gdelvall,tparker,kharriso,shxu}@cs.utsa.edu. Corresponding author: Shouhuai Xu.

cryptographic keys in their entirety.

### 1.1 Our Contributions

In this paper, we investigate *memory disclosure attacks* that exploit software vulnerabilities to disclose some amount of computer memory (RAM) and thus cryptographic keys. Specifically, we make three contributions. First, we assess the damage of memory disclosure attacks against private keys of OpenSSH servers, Apache servers, OpenSSH clients, and Firefox web browsers running on top of Linux operating system (OS). As case studies, we focus on private keys of the arguably most widely deployed RSA cryptosystem [Rivest et al. 1978]. The attacks exploit three reported vulnerabilities to disclose some portions of RAM content, while not having the `root` privilege on a victim machine. Our experiments show that from the disclosed memory, these attacks almost always expose *many copies* of the private key of the aforementioned application in Linux OS.

Second, in order to explain the phenomenon that many copies of private keys are disclosed, we implemented a software tool to help analyze the content of RAM. Through the tool, we found that many copies of private keys remain in both *allocated* memory and *unallocated* memory even in the most recent Linux kernel 2.6.28.9. This is surprising because the literature has reiterated many times the importance of clearing unallocated memory (cf. Viega et al. [Viega 2001; Viega and McGraw 2002] and Chow et al. [Chow et al. 2005]). This investigation has two important implications: (i) While the attacks are made possible by the memory disclosure vulnerabilities, their successes are much amplified by the fact that the private keys are somewhat *flooding* in computer RAM. (ii) Although the particular vulnerabilities we exploited have been fixed, any future (zero-day) exploit of this kind (disclosing even a small amount of memory) will successfully recover private keys with high probability.

Third, our analyses on the attacks suggest the following countermeasures: We should ensure (i) there are no unnecessary copies of the private key in allocated memory, while degrading little the system performance, and (ii) unallocated memory does not have a copy of the private key. For this purpose, we proceed to propose a set of software-based countermeasures. We conduct case studies by applying our countermeasures to protect the RSA private keys of OpenSSH servers, of Apache servers, and of OpenSSH clients. Experimental results show that our countermeasures do not incur any significant performance penalty, can eliminate attacks that disclose unallocated memory, and can mitigate the damage due to attacks that disclose a small portion of allocated memory. Through experiments, we can actually quantify the effectiveness of the countermeasures.

**Remark**. (i) The attacks we will report are for the vulnerable Linux kernels (prior to 2.6.26.4), which have been fixed. This should not mean that the investigation presented in this paper is not useful. Indeed, we believe that investigations like the one we present here are much needed because the key-flooding phenomenon remains relevant in the very recent Linux kernel 2.6.28.9 (released in March 2009). As long as cryptographic keys are flooding in computer RAM, any future (zero-day) exploit could expose private keys in question. Because our countermeasures can defeat attacks that disclose unallocated memory, and can mitigate the damage of attacks that disclose allocated memory, they are still relevant in practice. (ii) In our experiments we assume the knowledge of private keys. This is consistent with our principal motivation of studying the behavior of cryptographic keys in computer RAM. In actual attacks, the attacker has to use other means to identify the keys in computer RAM, such as the method of Shamir and van Someren [Shamir and

van Someren 1999]. As a side note, it might be better not to make public such attack tools. (iii) Throughout the paper we assume that the private keys are not password-encrypted when stored on the harddisk as a `PEM` file. If the `PEM` file is encrypted with a password, then the ciphertext private key will be decrypted and the plaintext private key will be stored in a RSA data structure specified by OpenSSL. As a result, the difference in terms of the number of copies of a private key appearing in RAM is the following: If the `PEM` file is not encrypted with a password, we see the plaintext `PEM` file in computer RAM; otherwise, we see the password-encrypted ciphertext of `PEM` file. There are no other differences, which makes is reasonable to assume (for simplicity in presentation) that the `PEM` private key file is not password-encrypted for storage on the harddisk.

## 1.2 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 threshold cryptosystems [Desmedt and Frankel 1990], proactive cryptosystems [Ostrovsky and Yung 1991], forward-secure cryptosystems [Anderson 1997; Bellare and Miner 1999; Bellare and Yee 2003; Itkis and Reyzin 2001], key-insulated cryptosystems [Dodis et al. 2002], and intrusion-resilient cryptosystems [Itkis and Reyzin 2002]. Clearly, our mechanisms can be utilized to provide another layer of protection for the private keys of these 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. [Viega 2001; Viega and McGraw 2002]). Unfortunately, as confirmed by our experiments as well as an earlier one due to Chow et al. [Chow et al. 2004], this practice has not been widely or effectively enforced. Chow et al. [Chow et al. 2004] 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 [Bochs ]. More recently, Chow et al. [Chow et al. 2005] 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 of time. As a result, their countermeasure can successfully eliminate attacks that disclose unallocated memory. However, their countermeasure has no effect in countering attacks that can disclose portions of allocated memory. Whereas, our countermeasures can not only eliminate attacks that disclose unallocated memory, but also mitigate the damage due to attacks that disclose portions of allocated memory. That is, we provide strictly stronger protections.

Software-based memory disclosure attacks differ from the attacks explored by Halderman et al. [Halderman et al. 2008], who aim to recover cryptographic keys from *only* portions of cryptographic keys that can be recognized from the RAM chips after powering off a computer. There have been elegant investigations (following [Akavia et al. 2009]) into designing cryptosystems that can tolerate *bounded* disclosure of information about cryptographic keys. Memory disclosure attacks (if successful) will likely expose the keys in their entirety (as shown in our experiments), which also distinguish them from side-channel attacks [Kocher 1996].

There are some loosely related works. Broadwell et al. [Broadwell et al. 2004] explored the core dump problem, and inferred which data in a system is sensitive based on

programmer annotations. This was motivated to facilitate the shipment of crash dumps to application developers without revealing users' sensitive data. Provos [Provos 2000] investigated using swap encryption for processes in possession of confidential data. Gutmann [Gutmann 1996] showed the difficulty of removing all remnants of sensitive data once written to a disk. A cryptographic treatment on securely erasing sensitive data via a small erasable memory was presented by Jakobsson et al. [Crescenzo et al. 1999].

**Outline**. The rest of the paper is organized as follows. 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 general method for countering memory disclosure attacks, and the details on how to apply them to OpenSSH server and Apache server. In Section 5 we report the effectiveness of the countermeasures against the memory disclosure attacks we experimented with and the effectiveness on reducing the key-flooding in computer memory. In Section 6, we study whether OpenSSH clients and `HTTP` browsers exhibit the key-flooding phenomenon. In Section 8, we highlight the lessons learned, which should be useful for both practitioners and designers. We conclude this paper in Section 9.

## 2. EXPERIMENTING WITH MEMORY DISCLOSURE ATTACKS

### 2.1 RSA in OpenSSL

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 $q$ (e.g., $|p| = |q| = 512$). In practice, a variation of the Chinese Remainder Theorem (CRT) is used for 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 $q^{-1} \mod p$. There is a special `PEM`-encoded private key file, which contains these parts. For simplicity, we only consider $d$, $p$, $q$, and the whole `PEM`-encoded file in the sense that disclosure of any of them immediately causes to the compromise of the private key. We call any appearance of any of them "a copy of the private key."

### 2.2 Memory Disclosure Vulnerabilities and Experimental System Settings

**Memory disclosure vulnerabilities**. We consider the following three vulnerabilities that can be exploited (*without* requiring `root` privilege) to launch memory disclosure attacks.

—The first vulnerability was reported in [Lafon and Francoise 2005], 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. We call it the `ext2` vulnerability, and the resulting attack the `ext2` attack. In this attack, the attacker needs to have a normal user account at the server, but neither the `root` privilege nor the account that owns the private key, and to have physical access to the server. Therefore, the attack can be launched by some non-privileged insider.

—The second vulnerability was reported in [Guninski 2005], 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. We call it the `tty` vulnerability, and the resulting attack the `tty` attack. For this attack, the attacker only needs to have a normal account on the server, neither `root` nor the private key owner. The attacker does not even need to have physical access to the server.

—The third vulnerability was reported in [CVE 2008] and states that Linux kernels prior to 2.6.24.4 are susceptible to disclosure of an arbitrarily-sized portion of kernel memory. This disclosure was possible due to an unsanitized user supplied size variable provided to the `sctp_getsockopt_hmac_ident` function in `net/sctp/socket.c`. This function is in the Linux implementation of Stream Control Transmission Protocol (`SCTP`). For a successful disclosure of memory, a vulnerable system should have the `SCTP-AUTH` extension enabled. We call it the `socket` vulnerability, and the resulting attack the `socket` attack. This attack also only requires the attacker to have a normal account on the server, neither `root` nor the private key owner. The attacker is also not required to have physical access to the server.

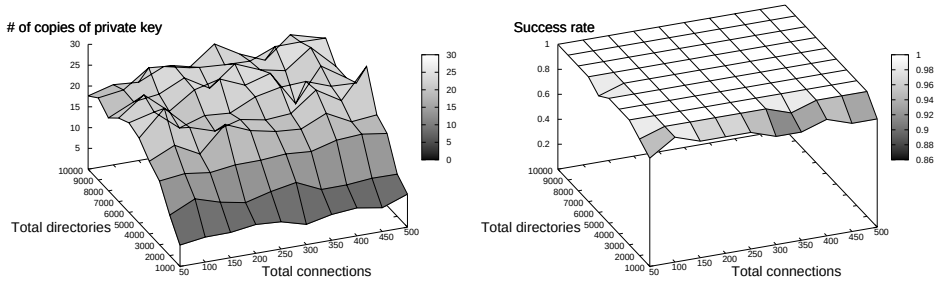
**Experimental system settings**. Our experiments (conducted in 2006) for exploiting the `ext2` and the `tty` vulnerabilities ran in the following setting: the server machine has a 3.2GHz Intel Pentium 4 CPU and 256MB memory; the server operating system is Gentoo Linux 2.6.10; the OpenSSH server is OpenSSH 4.3_p2; the Apache server is Apache 2.0.55 (compiled using the `mpm-prefork` option); the OpenSSL library version is 0.9.7i.

Our experiment (conducted in 2009-2010, at which time the computers used for the 2006 experiments are not available any more) for exploiting the `socket` vulnerability ran in the following setting: the server machine has an Intel Pentium 4 running at 2.26 GHz with 512 MB memory; the server and client are connected via a gigabit switch; the server operating system is Ubuntu 9.04 with a 2.6.24.1 kernel. The OpenSSH Server is OpenSSH 5.1_p1 with a configuration change to allow large numbers of connections. SSH traffic was generated with a custom Perl script that acted as a wrapper to spawn multiple OpenSSH *secure copy* (`scp`) connections concurrently. The Apache server is 2.2.13 (compiled with the `mpm-prefork` option). `HTTPS` traffic is generated with version 2.68 of siege [Siege ], a popular `HTTP`/`HTTPS` benchmarking and regression testing utility. The OpenSSL library version is 0.9.8l.

### 2.3 Experimental Results with the `ext2` Attack

Our experimental attack proceeded as follows. (i) We plugged a small 16MB USB storage device into the computer running OpenSSH (or Apache) 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 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.

**Experimental result with the OpenSSH server**. Figure 1(a) depicts the average (over 15 attack runs) number of copies of the private key found from the disclosed memory on the USB device, with respect to the number of `localhost` `SSH` connections and the number of created directories. For example, by establishing 500 total connections and creating 1,000 directories (i.e., disclosing up to about 4 MB memory), we were able to recover about 8 copies of the private key. From a different perspective, Figure 1(b) depicts the average success rates of attacks (i.e., the rates of the number of successful attacks over the total number of 15 attacks), which clearly states that an attack almost always succeeds.

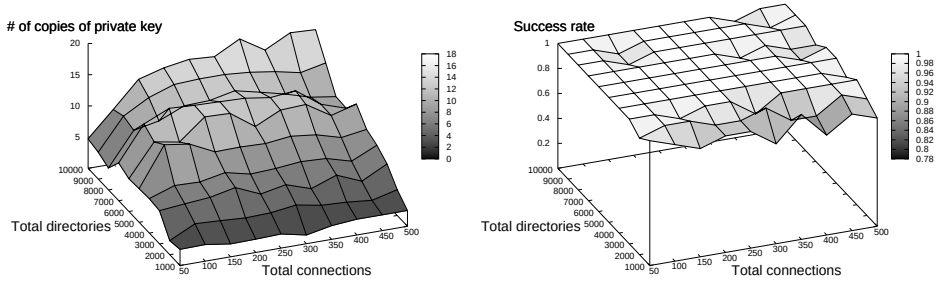



(a) # of copies of the private key found (b) Success rate

Fig. 1. Experimental result of the `ext2` attack against the vulnerable OpenSSH server

Excluding the time for scanning the disclosed memory, an attack run took *less than one minute*.

**Experimental result with the Apache server**. Figure 2(a) shows the average (over 15 attack runs) number of copies of the private key found on the USB device, with respect to the number of connections and the number of created directories. For example, by




(a) # of copies of the private key found (b) Success rate

Fig. 2. Experimental result of the `ext2` attack against the vulnerable Apache server

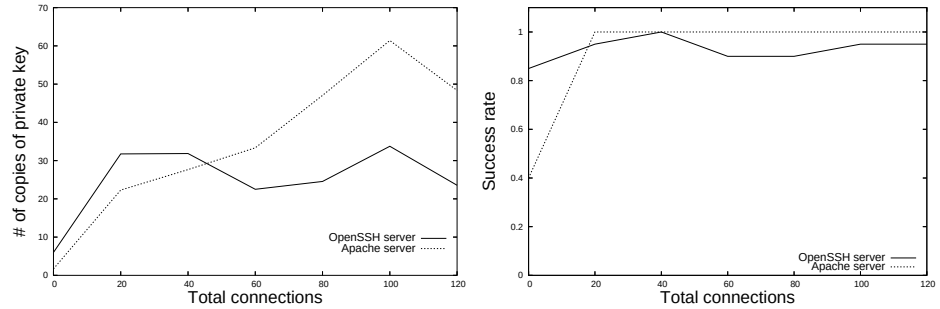
establishing 500 connections and creating 1,000 directories (i.e., disclosing up to 4 MB memory), we were able to recover about 5 copies of the private key. Figure 2(b) depicts the average success rates of attacks, namely the percentage of experiments in which there was at least one instance of the private key in the memory disclosed. It clearly states that an attack almost always succeeds. Excluding the time spent for scanning the disclosed memory, an attack took *less than five minutes*.

## 2.4 Experimental Results with the `tty` Attack

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 server, it instructed a remote computer to establish a large number of `HTTPS` connections to the server. (ii) The script executed a program (due to [Guninski

2005]) to disclose 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. In our experiment, the exploit disclosed about 50% of the memory (i.e., 128 MB) on average. Experimental results with both OpenSSH server and Apache server are summarized as follows.

Figure 3(a) shows the average (over 20 attack runs) number of copies of the private key found from the disclosed memory with respect to the number of connections. We observe that, in the case of OpenSSH server, the number of exposed private keys is not necessarily proportional to the number of concurrent connections; whereas, in the case of Apache server, the number of exposed private keys is proportional to the number of concurrent connections, until we reach 100 concurrent connections. Figure 3(b) shows the success




(a) # of copies of the private key found (b) Success rate

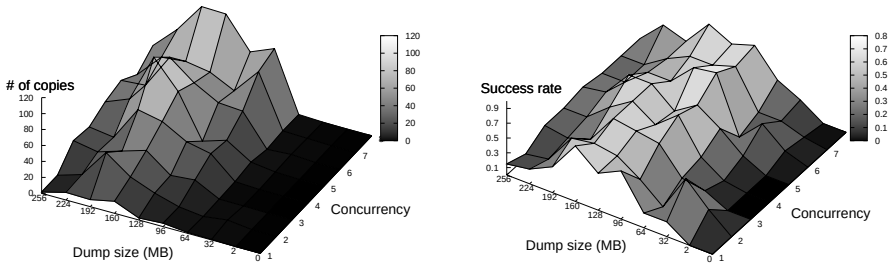
Fig. 3. Experimental result of the `tty` attack against the vulnerable OpenSSH server and Apache server

rates of attacks (i.e., the rates of the number of successful attacks over the total number of 20 attacks), which clearly states that in the case of OpenSSH server an attack almost always succeeds, and in the case of Apache server an attack always succeeds when 20 or more connections are established. In both cases, the attacks (excluding the time spent for scanning the disclosed memory) took *less than one minute*.

## 2.5 Experimental Results with the `socket` Attack

We used [Milworm 2008] to build an exploit. In both attacks, the basic steps were to create a flurry of network activity that would require use of the private key and then dump some specified amount of memory. Specifically, the attacks proceed as follows. (i) Create some local processes that request large chunks of memory, retain them and continually access them for the duration of the experiment. Create some local processes that randomly request small chunks of memory and access them for some random periods of time. These steps create memory pressure and may not be needed on a machine running other services and programs that use memory as well. (ii) Create many `scp` (or `HTTPS`) connections concurrently. (iii) Run the exploit to dump memory when `scp` (or `HTTPS`) connections are still ongoing. (iv) Scan the memory dump. It is important to note that the attacker specifies the amount of memory to attempt to disclose and, if the attacker specifies too much memory, the attack can fail to disclose any memory. The experimental results follow.

**Experimental result with the OpenSSH server**. Figure 4(a) shows the average (over 20 attack runs with successful memory disclosures) number of copies of the private key found from the disclosed memory with respect to the dumped memory size and the number of concurrent `scp` connections. Figure 4(b) shows the success rates of attacks, namely the




(a) # of copies of the private key found (b) Success rate

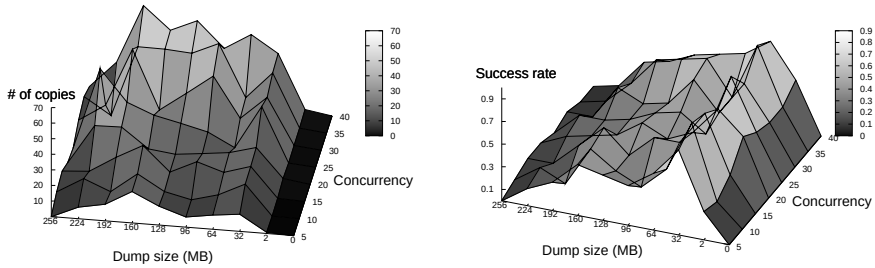
Fig. 4. Experimental result of the `socket` attack against the vulnerable OpenSSH server

percentage of experiments in which there was a successful dump of memory and there was at least one instance of the private key in the memory disclosed. Note that in cases where the exploit failed to dump memory, the key count was treated as zero. On average, an attack takes no more than 3 minutes. We can draw the following observations. (i) Lower dump sizes all the way up to 64 MB have minimal returns as compared to the higher dump sizes. (ii) Higher rates of concurrency coupled with larger sizes of dumped memory up to 192 MB prove to be an extremely successful combination that result in most numbers of copies of the private key found. The general trend — higher dump sizes in conjunction with higher concurrency rates result in higher values for keys found — continues until 192 MB, then the number of disclosed keys tails off rapidly. With this in mind, dumps of sizes larger than 256 MB were not attempted for this experiment.

**Experimental result with the Apache server**. Figure 5(a) shows the average (over 20 attack runs) number of copies of the private key found in the disclosed memory with respect to the dumped memory size and the number of concurrent `scp` connections. Figure 5(b) shows the success rates of attacks. In this case, an attack takes no more than 3 minutes. We can draw the following observations. Regardless of how the concurrency varied, very few keys were found for dumps of size 2 MB. At a dump size of 32 MB, there was a general trend upward as concurrency increased. There was also a noticeable increase in keys found at 32 MB from the previous case of 2 MB. The general trend — higher dump sizes in conjunction with higher concurrency rates result in higher values for keys found — continues until 192 MB, then the number of disclosed keys tails off rapidly. With this in mind, dumps of sizes larger than 256 MB were not attempted for this experiment.

## 2.6 Summary

Our experiments show that private keys can be easily compromised by memory disclosure attacks. Because the attacks are so successful, we suspect that private keys were somewhat




(a) # of copies of the private key found (b) Success rate

Fig. 5. Experimental result of the `ext2` attack against the vulnerable Apache server

somehow flooding in computer RAM. This motivates us to thoroughly examine, in the next section, the behavior of private keys in RAM.

## 3. EXPLAINING THE SUCCESS OF THE ATTACKS

### 3.1 A Tool for Locating Cryptographic Keys in Memory

In order to explain why the attacks were so successful, 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, called `scanmemory`, is about 260 lines. As highlighted in the pseudocode below, its functionality is to search for copies of a private key, `privkey`, within memory in a linear fashion. Thus, its time complexity is $O(M)$, where $M$ is the size of memory. In our experiments, it often took no more than 25 seconds to scan 512MB memory, which is sufficient for our purpose.

```
procedure scanmemory(string array privkey, integer numkeys) {
  foreach address from 0 to MEMSIZE
    foreach i from 0 to numkeys
      if *address equals privkey[i]
         Print 'Key (privkey[i]) found at address
            (address) owned by processes:';
         printOwingProcesses(address)
         Print newline
}
procedure printOwningProcesses(void pointer address){
  pageFrameNumber := address << PAGE_SHIFT
  page := page_frame_number_to_page(pageFrameNumber)
  foreach Anonymous_VMA associated with the page
    foreach process in the system
      if the given process has the given vma in its memory mapping
         Print '(process pid) '
  if no vma's are associated with the page
        but page_count(page) is > 0
     Print '0' #For the kernel
```

```
}
```

The `scanmemory` is implemented as a loadable kernel module (LKM). In addition to fulfilling the functionality of `scanmemory`, the LKM creates a `/proc` file system entry to facilitate communications between `scanmemory` and a user process. The `scanmemory` is invoked whenever the newly created `/proc` file system entry is read. For every copy of the private key found, `scanmemory` further searches through all processes to determine which processes, if any, have this physical memory page in their logical address space. To speed up this search, `scanmemory` takes full advantage of the reverse mapping functionality introduced in the 2.6 Linux kernel series. The output of `scanmemory` to the `/proc` file system includes locations of the `privkey` in memory, and identities of the processes that have the `privkey` in their logical address space.

### 3.2 Explaining the Success of the Attacks

We conducted experiments that ran on a pair of 2.26 GHz P4 machines, each with 512 MB RAM. The RAM size was kept low because it allows to scan RAM within about 25 seconds. Each machine had a 1 Gbit NIC, connected to a dedicated gigabit switch, and ran Ubuntu 9.04 Linux distribution with a 2.6.28.9 kernel, which is newer than the ones that were attacked against. The intent of experimenting with a newer version of OS, which was *not* known to be subject to the aforementioned three attacks, was to confirm or deny that the suspected phenomenon — cryptographic keys were somewhat flooding in the memory — is *still* relevant in newer OS.

**Experiment results with the OpenSSH server**. We use OpenSSH 5.1_p1 server, which is bundled with Ubuntu 9.04 distribution, with OpenSSL 0.9.8l being linked at run-time. The experiment proceeds as follows (time unit: 2 minutes).

—Time t=0: The experiment is started.

—Time t=2: The OpenSSH server is started.

—Time t=4: 5 processes that run `scp` are launched, each process initiating single sequential 2 KB file transfers to the server continuously.

—Time t=5: The 5 `scp` processes continue to transfer files to the server.

—Time t=6: Another 5 `scp` instances are launched, each initiating single sequential file transfers to the server continuously. It is now the case that we have 10 `scp` processes transferring files concurrently.

—Time t=7: The 10 `scp` processes continue to transfer files to the server.

—Time t=8: The first 5 `scp` processes are terminated.

—Time t=9: The remaining 5 `scp` processes continue to transfer files to the server.

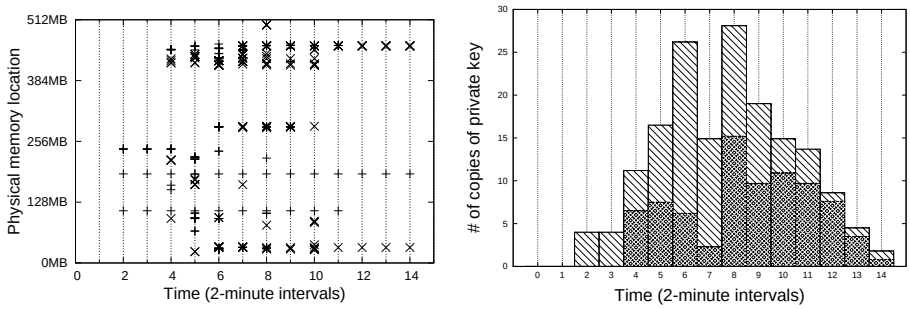
—Time t=10: The remaining 5 `scp` processes are terminated. At this point, the OpenSSH server is still running but there are no connected clients.

—Time t=12: The OpenSSH server is stopped.

—Time t=14: The experiment is completed.

Figure 6(a) plots the locations of copies of the OpenSSH server's RSA private key in RAM. Figure 6(b) depicts the average (over 10 runs) number of copies of the private key in RAM with respect to time. As expected, no keys were present in RAM before the OpenSSH server was started. At time step 2, approximately 3 keys were visible in memory

as well as 1 copy of the PEM-encoded file. When contrasted to the Apache case (demonstrated below), we see much fewer keys. This is attributed to the fact that OpenSSH only launches a process at start-up and spawns new processes as connections arrive; whereas, Apache launches 6 processes at start-up, each with a key copy. At time step 4, there is




(a) Locations of keys in memory (single instance) (b) Key frequency vs. Time (avg. over 10 runs)

Fig. 6. Experimental result with the OpenSSH server. (For left-hand figure, "+" represents a key copy in allocated memory, "×" represents a key copy in unallocated memory. For right-hand figure, light bar portions represent keys found in allocated memory, and dark bar portions represent keys found in unallocated memory.)

a significant increase in keys present. This is due to the first wave of 5 scp connections that were just started. Each new connection would result in a key present in memory. At time step 6, a second wave of scp connections further cause the keys to increase sharply again. At time step 8, we see a notable increase in the number of keys. At time step 10, the scp processes are stopped and the key count begins to decrease. The keys at this point do not belong to any process but rather are just in unallocated kernel memory and the page cache. At time steps 11 through 14, we see a continued decrease in the number of keys in memory. Again, we attribute the decrease to the kernel re-purposing free pages. Overall, we see that there are many keys in memory during periods of server load as well as after the traffic has subsided. From a performance point of view, it is desirable to have keys resident in memory so as to allow for rapid connection establishment and for preventing processes from being in an I/O wait state unnecessarily.

**Experimental result with the Apache server**. For the case of Apache, Apache server was compiled from Apache 2.2.13 source with the mpm-prefork option and OpenSSL 0.9.8l linked at runtime. The experiment proceeds as follows (time unit: 2 minutes).

—Time t=0: The experiment is started without Apache running.

—Time t=2: Apache server is started.

—Time t=4: 10 Firefox instances are launched, each initiating a single HTTPS request.

—Time t=5: The 10 Firefox instances on the client are terminated.

—Time t=6: 10 more Firefox instances are launched, each initiating a single HTTPS request.

—Time t=7: The 10 Firefox instances on the client are terminated.

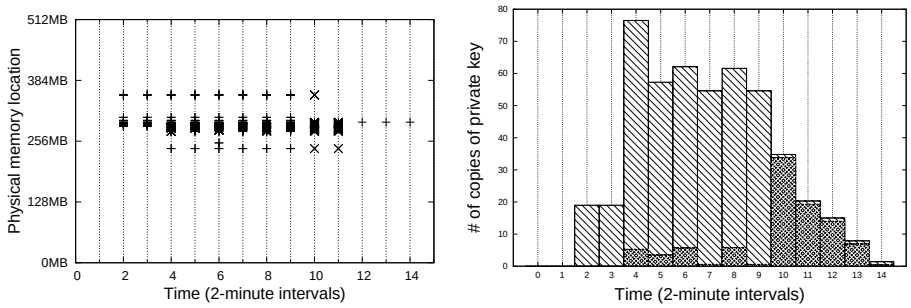
—Time t=8: 10 more Firefox instances are launched, each initiating a single HTTPS request.

—Time t=9: The 10 Firefox instances are terminated.

—Time t=10: Apache server is stopped.

—Time t=14: The experiment is completed.

Figure 7(a) plots the locations of copies of the Apache server's RSA private key in RAM. Although there is 512 MB of RAM available, memory allocated for processes is typically in the upper half of memory. There were rare instances where copies of the private key spread over the whole memory space. Figure 7(b) depicts the average (over 10 runs) number of copies of the private key in system memory with respect to time. We observe the following: As expected no keys were present in memory before the Apache processes were started. At time step 2, approximately 18 keys were visible in memory as well as 1 copy of the




(a) Locations of keys in memory (single instance) (b) Key frequency vs. Time (avg. over 10 runs)

Fig. 7. Experimental result with the Apache server. (For left-hand figure, "+" represents a key copy in allocated memory, "×" represents a key copy in unallocated memory. For right-hand figure, light bar portions represent keys found in allocated memory, and dark bar portions represent keys found in unallocated memory.)

PEM-encoded file. At time step 4, the number of keys spikes sharply. This is because 10 Firefox clients were launched from a remote host connecting to the web server. At this point several processes are launched to handle the requests and as such several copies of the key appear in memory for the SSL negotiation phase of the SSL session. At time steps 6 and 8, 10 connections are initiated each time. The key frequency varies somewhat, but the main observation is that it stays consistently high. At time step 10, the Apache processes are stopped and the key count begins to decrease. The keys at this point do not belong to any process, but rather are just in unallocated memory and in the page cache (in the form of the PEM file). At time steps 11 and 12, we see a continued decrease in the number of keys in memory.

### 3.3 Summary

While the attacks were made possible by the software vulnerabilities, the number of exposed copies of private keys and the attack success rates are amplified by that private keys are somewhat flooding in computer RAM, namely that there are many copies of private keys appearing in both allocated and unallocated memory. This would explain why the memory disclosure attacks were so successful.

## 4. COUNTERMEASURES AGAINST MEMORY DISCLOSURE ATTACKS

### 4.1 Transient Key Copies Versus Persistent Key Copies

When considering copies of a private key in RAM, we draw a distinction between *persistent* copies, which can have an extended lifetime (e.g., may exist for the entire lifetime of the process to which they belong), and *transient copies*, which exist briefly to facilitate low-level cryptographic computations. To see the need for considering transient copies, we use Algorithm 1 to highlight the basic operations in the OpenSSL library function `RSA_eay_mod_exp`, which is Eric Young's modular exponentiation routine and adopted by OpenSSL and used by both Apache and OpenSSH. As shown, OpenSSL uses the Chinese Remainder Theorem for RSA decryption/signing.

**Algorithm 1** Pseudocode for RSA signing/decryption in OpenSSL

1: $I \leftarrow padding(message)$ {according to `PKCS #1 v2.0 EMSA-PKCS1-v1_5` (PKCS #1 v1.5 block type 1)}
2: $r_1 \leftarrow I \mod q$
3: $m_1 \leftarrow r_1^{d \mod q-1} \mod q$
4: $r_0 \leftarrow I \mod p$
5: $r_1 \leftarrow r_1^{d \mod p-1} \mod p$
6: $r_0 \leftarrow r_0 + p$
7: $r_1 \leftarrow r_0 \cdot (q^{-1} \mod p)$
8: $r_0 \leftarrow r_1 \mod p$
9: $r_0 \leftarrow r_0 + p$
10: $r_1 \leftarrow r_0 \cdot q$
11: $r_0 \leftarrow r_1 + m_1$

In Figure 8 we plot the event sequence that caused the transient copies of $p$ and $q$ in RAM. Specifically, instruction 2 in Algorithm 1 caused one transient copy of $q$ in RAM, instruction 3 in Algorithm 1 caused another transient copy of $q$ in RAM, while preserving the existing copy of $q$ (these two transient copies are connected with a dashed line), etc. Although there are, in total, five transient copies of $p$ and $q$, there are two transient

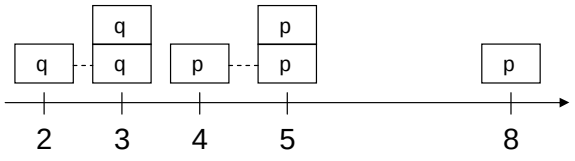



Fig. 8. Transient copies of $q$ and $p$ in computer RAM caused by RSA decryption/signing

copies of $q$ corresponding to instruction 3 in Algorithm 1, and two transient copies of $p$ corresponding to instruction 5 in Algorithm 1. This means that a memory disclosure at intrustion 3 or 5 might cause a higher probability of key exposure when compared with a memory disclosure at instruction 2, 4, or 8 because there is, in the former case, one more transient copy in RAM.

Looking into the source code, we found that the transient copies of $p$ and $q$ are due to the following. Because the numbers are much larger than computers natively support

in the processor, and so arbitrary-precision arithmetic libraries are used for cryptographic computations. OpenSSL's arbitrary precision arithmetic is known as the `bignum` library, which is abbreviated as `BN`, and works by representing numbers using a high radix (e.g., base 65536 instead of base 10 or base 2). `BN_mod` is implemented as `BN_div`, i.e,. a divide operation. `BN_div` begins by normalizing the divisor and dividend, a typical operation in arbitrary precision division. (The normalization is seemingly to ensure that the intermediate results in the division operations fit within machine words so as to allow efficient CPU processing, while avoiding bit-by-bit arithmetic operations.) This normalization operation is performed by shifting the number left (using `BN_lshift`), which of course modifies the number being shifted. Thus, OpenSSL creates temporary copies of $p$ and $q$ in order to avoid mutating the real copies, and these are the transient copies we observe. Note that the addition and multiplication operations involving $p$ and $q$ did not require numbers to be mutated, and thus did not incur extra transient copies.

### 4.2 Basic Idea of the Countermeasures

To mitigate the damage of memory disclosure attacks, we want to eliminate unnecessary persistent key copies in RAM, while it is seemingly impossible to dismiss the transient copies without significant performance penalty. For this purpose, we present a set of countermeasures that can be deployed at different layers of the software stack, from application down to OS kernel. Their strengths and limitations will be examined, while we recommend the integrated countermeasure.

* **Application-level countermeasure**: At the application level, the following can be enforced.
  (1) Utilize the "copy on write" memory management policy [Silberschatz et al. 2001] to avoid unnecessary copies of a private key. This policy says that when a process is forked, its physical memory will be copied only after one process attempts to write to that memory region. Specifically, we propose placing a private key into a special memory region, and ensuring that no process will write to that special memory region.
  (2) Ensure that the private key is not explicitly copied by the application or any involved libraries.
  (3) Disable swapping of the memory that contains the private 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.

  These ensure that, in addition to the `PEM`-encoded private key file, there are no unnecessary persistent copies of the private key in allocated memory, no matter how many processes are forked. Note that the application-level countermeasure can prevent private keys from appearing in memory other than the aforementioned special region (which exists in user-space) and the `PEM`-encoded private key file (which exists in kernel-space). Nevertheless, the application-level countermeasure alone cannot guarantee that there are always no keys appearing in unallocated memory, unless special care has been taken to clear the aforementioned special memory region before the application itself dies.
* **Library-level countermeasure:** At the library level, we propose eliminating unnecessary persistent key copies in allocated memory using the same countermeasure as suggested for the application level. This suffices to prevent private keys from appearing in memory other than the aforementioned special region and the `PEM`-encoded private key

file. However, the library-level countermeasure alone cannot guarantee that there are always no key copies in unallocated memory, unless special care has been taken to clear the special memory region before the application dies.

* **Kernel-level countermeasure:** At the kernel level, we propose ensuring that the unallocated memory does not contain any copy of the private key. This can be fulfilled by having the kernel clear any physical pages before they become unallocated. However, kernel-level countermeasure alone cannot prevent unnecessary copies of the private key in allocated memory.
* **Integrated countermeasure:** We propose a crossing-layer countermeasure, which is *not* just the combination of the some or all of the above individual countermeasures. This way, unnecessary copies of the private key in allocated memory and copies of the private key in unallocated memory are simultaneously eliminated. Moreover, this countermeasure can even remove the PEM-encoded private key file from allocated memory, provided that the library instructs the kernel not to cache the PEM-encoded private key file.

The above discussion also suggests that we recommend the integrated countermeasure because it can achieves better security. This countermeasure also has an appealing advantage: It can achieve the desired protection in a fashion transparent to the applications, meaning that the applications do not have to be changed. This application-transparency is important because it is often difficult to make changes to many applications.

### 4.3 Applying the Countermeasures to OpenSSH Server

**Application-level countermeasure**. At the application level, we instantiate the above general countermeasure as a new function, `RSA_memory_align()`, which should be called as soon as OpenSSL's RSA data structure (we call `RSA struct` for conciseness) contains the private key. This ensures that no unnecessary copy of the private key appears in allocated memory, in addition to the PEM-encoded private key file. Specifically:

(1) `RSA_memory_align()` uses `posix_memalign()` to request one or more memory pages for fulfilling a 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 pointer in `RSA struct` 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) `RSA_memory_align()` 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. This function is normally called once per connection when establishing a session key. Moreover, `RSA_memory_align()` disables swapping of memory that contains the private key by calling `mlock()` on the memory allocated by `posix_memalign()`. This is necessary for dealing with memory disclosure attacks because memory that is swapped out is not immediately cleared. As an added benefit this measure helps prevent swap space based attacks.

It should be noted that the functionality of `RSA_memory_align()` cannot be fulfilled by OpenSSL's `RSA_memory_lock()`, which may seemingly fulfill what `RSA_memory_align()` is doing at a first glance.

**Library-level countermeasure**. At the library level, we modify the OpenSSL function `d2i_PrivateKey()`. This function is responsible for translating a `PEM`-encoded private key file into the RSA key parts by calling `d2i_RSAPrivateKey()` to fill in the RSA data structure. The modification is that when the `d2i_RSAPrivateKey()` method returns, we immediately call the function `RSA_memory_align()` mentioned above.
**Kernel-level countermeasure**. At the kernel level, 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. This way, it is guaranteed that there are no private key copies in unallocated memory. As mentioned before, this countermeasure was well known but somehow not widely adopted in practice.
**Integrated countermeasure**. In addition to the modifications made by the library-level countermeasure and the kernel-level countermeasure 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 (i.e., OpenSSH client and server in this paper) 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()`. Note that this countermeasure is *not* the combination of the application-level, the library-level, and the kernel-level countermeasures.

### 4.4 Applying the Countermeasures to Apache Server

**Application-level countermeasure**. This is the same as in the application level countermeasure for OpenSSH server. Our function `RSA_memory_align()` is used just as before to align the RSA private keys within memory as soon as they are allocated by OpenSSL. Recall that this function clears the bit flag `RSA_FLAG_CACHE_PRIVATE` and aligns all the keys on a single page of memory allocated via `posix_memalign()`, after that it locks the page in memory via a call to `mlock()`. The `RSA_memory_align()` function was added to the Apache `mod_ssl` source code, and called from the appropriate place.
**Library-level countermeasure**. This is the same as its counterpart in the case of OpenSSH server. Specifically, the RSA private key is kept on a single page, the `RSA_FLAG_CACHE_PRIVATE` flag was cleared so as to disable replications of the RSA private key, the keys are prevented from being swapped out by a call to `mlock()`.
**Kernel-level countermeasure**. This is the same as its counterpart in the case of OpenSSH server. Specifically, the `free_hot_cold_page()` function is modified to ensure that memory released by a process is cleared before it can be allocated again.
**Integrated countermeasure**. This is even simpler than its counterpart in the case of OpenSSH server because no modifications to Apache server are required. Note that this countermeasure is neith the combination of the above three countermeasures, nor the combination of the above library-level and kernel-level countermeasures.

### 4.5 Summary

Because we might have to telerate the apperances of transient copies of a private key (for at least preserving the performance of the current software implementation), we propose a general method for eliminating the unnecessary apperances of the persistent copies of a private key in computer RAM. This draws questions such as their effectinvess in resisting

memory disclosure attacks, their effectiveness in dealing with the key-flooding problem, and their impact on the performance of the resulting systems. These questions are addressed in the next section.

## 5. EFFECTIVENESS OF THE COUNTERMEASURES

To be concise, we report the effectiveness of our integrated countermeasure against the attacks mentioned above. We recommend this countermeasure because it leads to better security. When the need arises, we also mention the effectiveness of other countermeasures.

### 5.1 Effectiveness Against the `ext2` Attack

**The case of OpenSSH server**. We re-examined the `ext2` attack against the same vulnerable 2.6.10 Linux kernel except that the system is now patched with our integrated countermeasure, which we recommend. In no case were we able to recover the private key because there are no copies of the private key in unallocated memory.

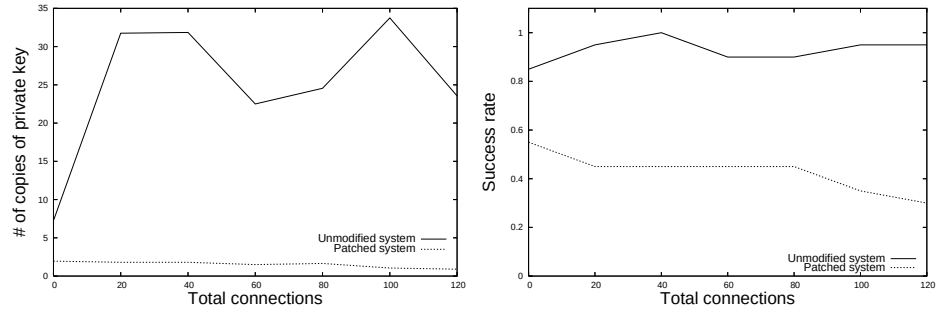
**The case of Apache server**. We re-examined the `ext2` attack against the same vulnerable 2.6.10 Linux kernel except that the system is now patched with our integrated countermeasure. In no case were we able to recover the private key because there are no key copies in unallocated memory.

Note that in both cases, the kernel-level countermeasure can also completely eliminate this attack because it ensures that no key copies of a private key appear in unallocated memory. Note also that neither the application-level countermeasure nor the library-level countermeasure by itself can completely eliminate this attack because the private key may still appear in unallocated memory once.

### 5.2 Effectiveness Against the `tty` Attack

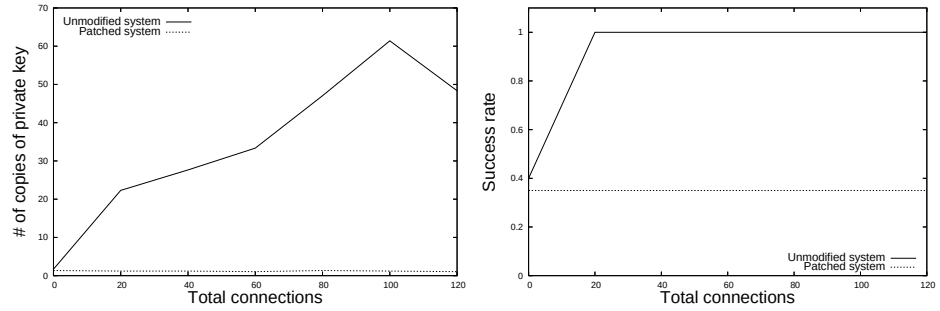
**The case of OpenSSH server**. We re-examined the `tty` attack against the same vulnerable 2.6.10 Linux kernel, except that the system is now patched with our integrated countermeasure. Figure 9(a) compares the average (over 20 attack runs) number of copies of the private key found in the disclosed memory *before* and *after* deploying our countermeasure. It shows that the number of copies of the private key recovered is significantly reduced. This is because our countermeasure ensured that no unnecessary copies of the private key appear in allocated memory, and no copies of the private key appear in unallocated memory. Figure 9(b) compares the success rates of attacks *before* and *after* deploying our countermeasure. While our countermeasure significantly reduces the success rates of attacks, the attack still succeeds with a probability about 50%. The reason is that the attack discloses on average about 50% of the memory, which means that with about 50% probability an attack would still successfully expose the private key.

**The case of Apache server**. We re-examined the `tty` attack against the same vulnerable 2.6.10 Linux kernel, except that the system is now patched with our integrated countermeasure. Figure 10(a) compares the average (over 20 attack runs) number of copies of the private key found in the disclosed memory *before* and *after* deploying our countermeasure. It clearly shows that the number of copies of the private key recovered is significantly reduced. This is because our countermeasure ensured that no unnecessary copies of the private key appear in allocated memory, and no copies of the private key appear in unallocated memory. Figure 10(b) compares the success rates of attacks *before* and *after* deploying our countermeasure. While our countermeasure substantially reduces the success




(a) # of copies of private key found (b) Success rates

Fig. 9. Effectiveness of the integrated countermeasure against the `tty` attack: The case of OpenSSH server




(a) # of copies of private key found (b) Success rates

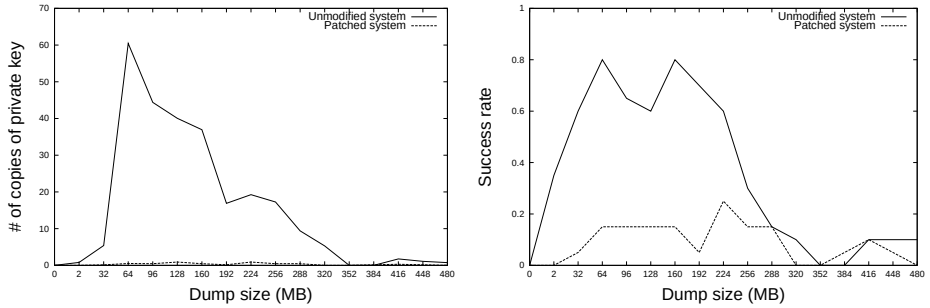
Fig. 10. Effectiveness of the integrated countermeasure against the `tty` attack: The case of Apache server

rate of attack, the attack can still succeed with a probability about 38% as a consequence of disclosing on average about 50% of the RAM content.

### 5.3 Effectiveness Against the `socket` Attack

**The case of OpenSSH server**. We re-examined the `socket` attack against the same vulnerable 2.6.24.1 kernel, with the system now patched with our integrated countermeasure. Figure 11(a) depicts the number of private keys disclosed (again averaged over 20 attack runs) in the kernel memory dump file *before* and *after* deploying our countermeasure. We notice a significant decrease in the number of key copies exposed. This can be attributed to the fact that our countermeasure cleared unallocated memory and avoid any unnecessary copies of the private key in allocated memory. Figure 11(b) depicts the success rates of attacks *before* and *after* deploying our countermeasure. It shows that in the best case an attacker has a 25% chance of disclosing a portion of memory that has a copy of the private key. Note that with a dump size 384 MB the attack success rate is actually higher against the patched system (1 success) than against the unpatched/unmodified (0 successes). This is because in the former case the attacker indeed successfully disclosed the required memory, but was not successful in the latter case.

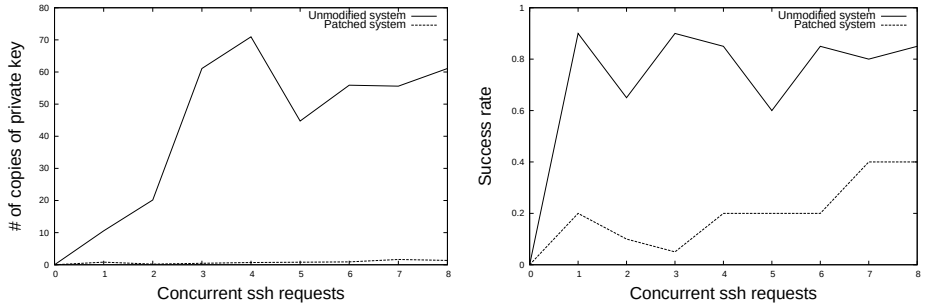
Figure 12 plots the impact of concurrent OpenSSH sessions. Figure 12(a) shows that the




(a) # of copies of private key found (b) Success rates

Fig. 11. Effectiveness of the countermeasure against the `socket` attack with respect to disclosed memory size: The case of OpenSSH server

integrated countermeasure significantly reduces the number of private key copies disclosed, regardless of the volume of traffic the OpenSSH server receives. Figure 12(b) depicts the

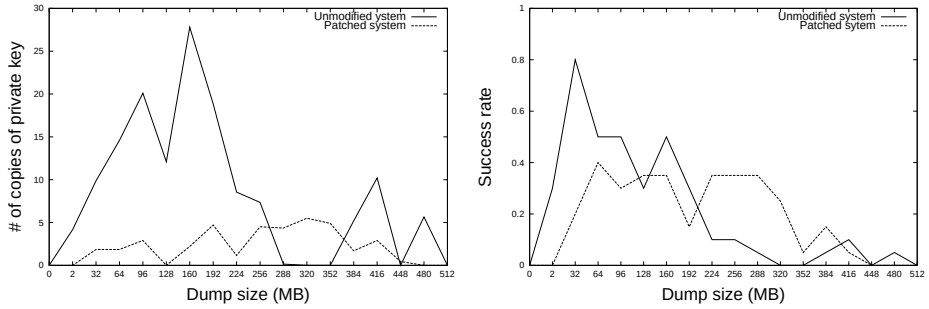



(a) # of copies of private key found (b) Success rates

Fig. 12. Effectiveness of the integrated countermeasure against the `socket` attack with respect to concurrent sessions: The case of OpenSSH server

success rates of attacks *before* and *after* deploying our countermeasure. It shows that the attacker's probability of success decreased from 90% to 40% in the best case.

**The case of Apache server**. We re-examined the `socket` attack against the same vulnerable 2.6.24.1 kernel, but with the system now patched with our integrated countermeasure. Figure 13(a) depicts the number of copies of the private key disclosed (again averaged over 20 attack runs) in the kernel memory dump file *before* and *after* deploying our countermeasure. We observe that on the average our countermeasure significantly decreases the number of copies of the private key disclosed. Notice that system patched with our countermeasure can sometimes cause the leakage of a private key, but the same system before patching with our countermeasure did not. This is specifically caused by the fact that the attack does not always successfully disclose the required memory, which can happen for both cases. This also explains why sometimes the attack success rate in system patched with our countermeasure can be higher than its counterpart before applying our

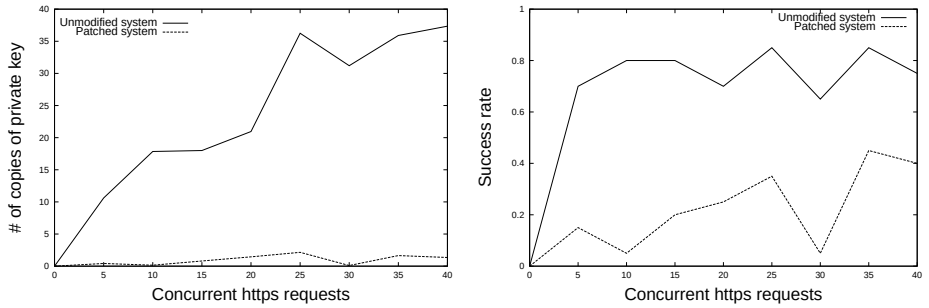



(a) # of copies of private key found (b) Success rates

Fig. 13. Effectiveness of the integrated countermeasure against the `socket` attack with respect to disclosed memory size: The case of Apache server

countermeasure. Figure 13(b) demonstrates that the attacker does not have better than a 40% chance of a successful key disclosure.

Figure 14 depicts the impact of concurrent HTTPS sessions. Figure 14(a) shows that the countermeasure is effective at reducing average key disclosure, irrespective of HTTPS traffic applied to the Apache server. Figure 14(b) depicts the success rates of the attacks



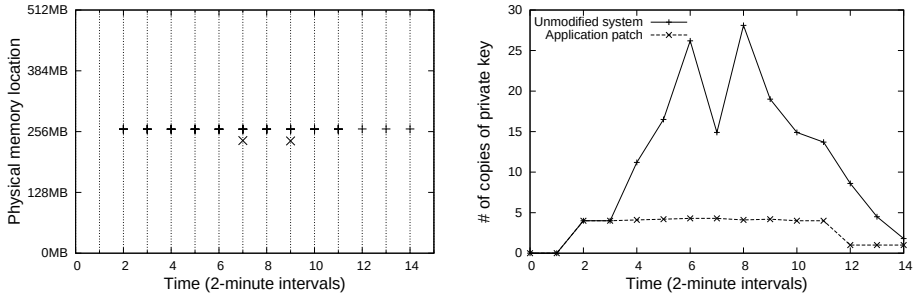

(a) # of copies of private key found (b) Success rates

Fig. 14. Effectiveness of the integrated countermeasure against the `socket` attack with respect to concurrent sessions: The case of Apache server

*before* and *after* deploying the countermeasure. It shows that the attacker has at best a 40% chance of disclosing a key; whereas in the unpatched case the attacker would almost always be guaranteed a successful key disclosure.

## 5.4 Effectiveness on Mitigating the Key-Flooding Problem in OpenSSH Server

5.4.1 *Previous Kernel + Previous OpenSSL.* **Effectiveness of the application-level countermeasure**. We investigated the system after patching with the application-level countermeasure. Figure 20(a) shows the locations of the copies of the private key found in memory. It is immediately clear that under traffic the key count is stable. This is clearly not the case with the unpatched/unmodified system (as shown in Figure 6). Moreover, when the server running in the unpatched system is stopped, a solitary copy of the key remains in




(a) Location of keys in memory (b) # of keys in memory

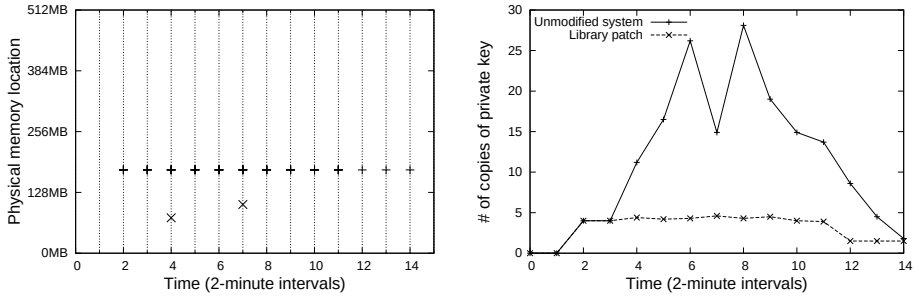
Fig. 15. Effectiveness of the application-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

memory — the PEM-encoded private key copy. In time steps 1 through 3 , we see that both systems behave basically the same when the server sits idle. At time step 6, we see the largest difference in key count between the patched case and the unmodified case. What should be seen here is that in most cases the patched system has fewer keys resident in memory while the OpenSSH server is handling many connections. In both cases we do see that key count is still closely related to the number of server processes, as expected. From time step 12 through 14, when the server is stopped, we see that the patched case only has the PEM-encoded private key file in memory while the unmodified system has the PEM-encoded private key file as well as some keys that are the residue of the processes that were active. Because the unpatched system still has keys that need not be in memory, our application-level is a clear improvement as shown in Figure 20(b), which compares numbers of copies of the private key found in the patched system and in the unmodified system.

**Effectiveness of the library-level countermeasure**. Figure 21(a) shows the locations of the copies of the private key found in memory after adopting the library-level countermeasure. We observe similar result as in the case of after adopting the application-level countermeasure. Moreover, Figure 21(b) shows that the library-level countermeasure can indeed substantially reduce the number of copies of the private key in memory.

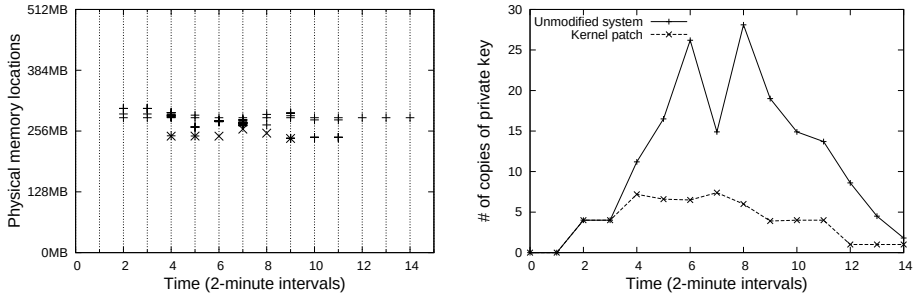
**Effectiveness of the kernel-level countermeasure**. Figure 22(a) plots the locations of copies of the private key found in memory after adopting the kernel-level countermeasure. Figure 22(b) compares the case of unmodified system and the case of the patched system. We observe very similar effect as in the cases of both application-level and library-level countermeasures.

**Effectiveness of the integrated countermeasure**. Figure 23(a) shows the locations of copies of the private key found in memory after patching with the integrated countermeasure. Figure 23(b) compares the number of copies of the private key found in the memory of the patched and the unmodified systems. We observe that the countermeasure performs exceedingly well in this case. At any one time, we do not have more keys in memory than are needed because the transient copies of the private key for RSA operations are used they are cleared, and the PEM-coded private key file is flushed from the page cache after it is read. When the process is terminated, we no longer have keys in memory.




(a) Location of keys in memory (b) # of keys in memory

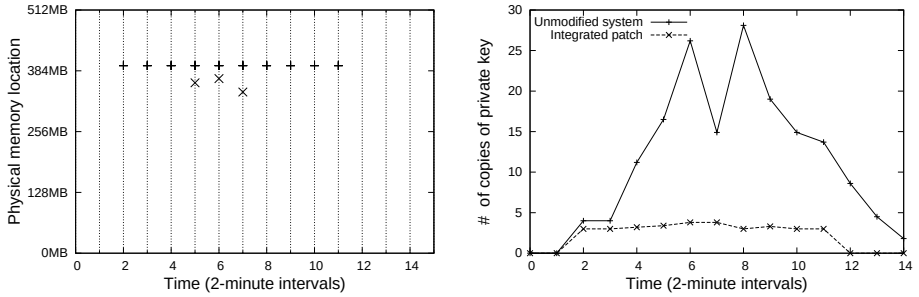
Fig. 16. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.




(a) Location of keys in memory (b) # of keys in memory

Fig. 17. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

Because we recommend the integrated countermeasure, now we discuss the impact of our integrated countermeasure on system performance. Since we are not aware of any benchmark for measuring the performance of OpenSSH servers, we wrote a simple Perl script to do a performance analysis. The Perl script is run on a client machine and maintains 20 concurrent `scp` connections to the OpenSSH server. The 20 concurrent connections repeatedly transfer 10 different files until 4000 file transfers have been completed. The 10 different files vary in size from 1 KB to 512 KB, with an average size of 102.3 KB. This analysis was repeated 20 times in order to obtain average measurements. We considered two metrics: *transaction rate*, namely the number of files transferred per second, and *throughput* in terms of Mbits per second transferred. Figure 24 plots the comparison. In each metric case, the left bar corresponds to the measurement before adopting the countermeasure, and the right bar corresponds to the measurement after adopting the countermeasure. It is apparent that the countermeasure has little impact on performance.




(a) Location of keys in memory (b) # of keys in memory

Fig. 18. Effectiveness of the integrated countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

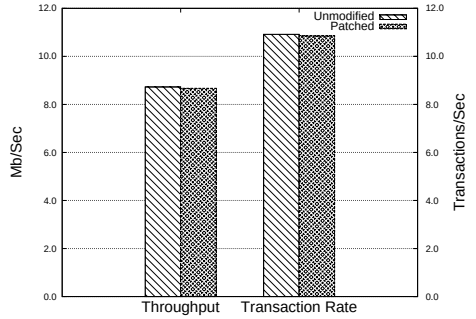
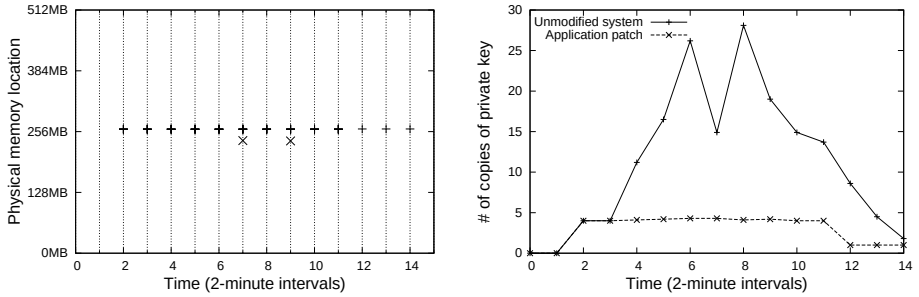



Fig. 19. OpenSSH server performances before and after adopting the integrated countermeasure

5.4.2 *Newest Kernel + Newest OpenSSL: The content is just placeholder right now.* **Effectiveness of the application-level countermeasure**. We investigated the system after patching with the application-level countermeasure. Figure 20(a) shows the locations of the copies of the private key found in memory. It is immediately clear that under traffic the key count is stable. This is clearly not the case with the unpatched/unmodified system (as shown in Figure 6). Moreover, when the server running in the unpatched system is stopped, a solitary copy of the key remains in memory — the `PEM`-encoded private key copy. In time steps 1 through 3 , we see that both systems behave basically the same when the server sits idle. At time step 6, we see the largest difference in key count between the patched case and the unmodified case. What should be seen here is that in most cases the patched system has fewer keys resident in memory while the OpenSSH server is handling many connections. In both cases we do see that key count is still closely related to the number of server processes, as expected. From time step 12 through 14, when the server is stopped, we see that the patched case only has the `PEM`-encoded private key file in memory while the unmodified system has the `PEM`-encoded private key file as well as some keys that are the residue of the processes that were active. Because the unpatched system still has keys that need not be in memory, our application-level is a clear improvement as shown

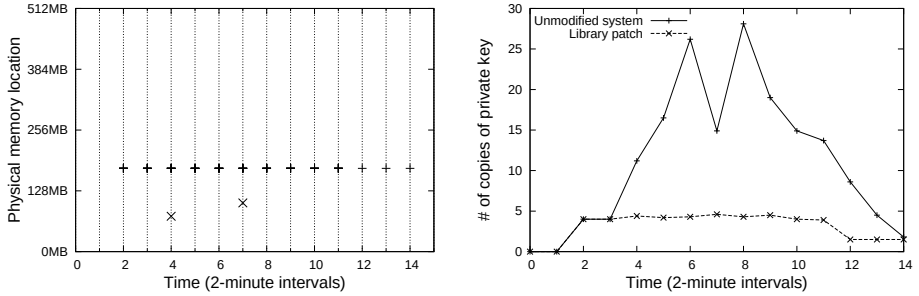



(a) Location of keys in memory (b) # of keys in memory

Fig. 20. Effectiveness of the application-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

in Figure 20(b), which compares numbers of copies of the private key found in the patched system and in the unmodified system.

**Effectiveness of the library-level countermeasure**. Figure 21(a) shows the locations of the copies of the private key found in memory after adopting the library-level countermeasure. We observe similar result as in the case of after adopting the application-level countermeasure. Moreover, Figure 21(b) shows that the library-level countermeasure can



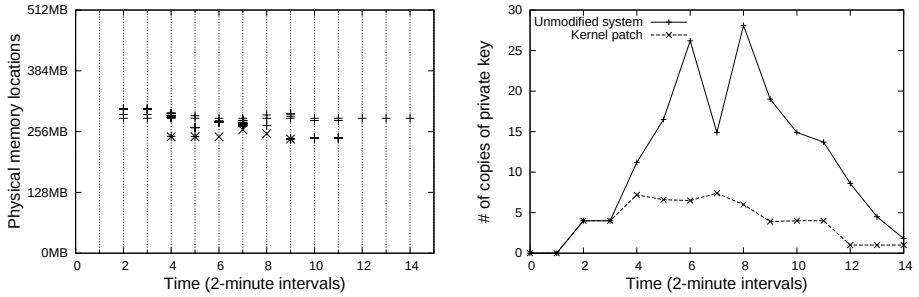

(a) Location of keys in memory (b) # of keys in memory

Fig. 21. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

indeed substantially reduce the number of copies of the private key in memory.

**Effectiveness of the kernel-level countermeasure**. Figure 22(a) plots the locations of copies of the private key found in memory after adopting the kernel-level countermeasure. Figure 22(b) compares the case of unmodified system and the case of the patched system. We observe very similar effect as in the cases of both application-level and library-level countermeasures.

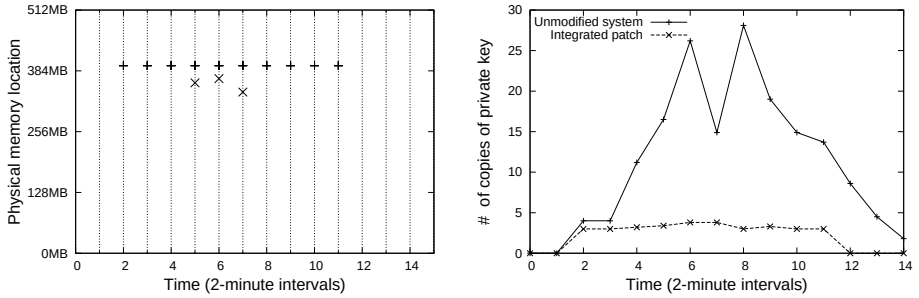
**Effectiveness of the integrated countermeasure**. Figure 23(a) shows the locations of copies of the private key found in memory after patching with the integrated countermea-




(a) Location of keys in memory (b) # of keys in memory

Fig. 22. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

sure. Figure 23(b) compares the number of copies of the private key found in the memory of the patched and the unmodified systems. We observe that the countermeasure performs




(a) Location of keys in memory (b) # of keys in memory

Fig. 23. Effectiveness of the integrated countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH server. For left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

exceedingly well in this case. At any one time, we do not have more keys in memory than are needed because the transient copies of the private key for RSA operations are used they are cleared, and the `PEM`-coded private key file is flushed from the page cache after it is read. When the process is terminated, we no longer have keys in memory.

Because we recommend the integrated countermeasure, now we discuss the impact of our integrated countermeasure on system performance. Since we are not aware of any benchmark for measuring the performance of OpenSSH servers, we wrote a simple Perl script to do a performance analysis. The Perl script is run on a client machine and maintains 20 concurrent `scp` connections to the OpenSSH server. The 20 concurrent connections repeatedly transfer 10 different files until 4000 file transfers have been completed. The 10 different files vary in size from 1 KB to 512 KB, with an average size of 102.3 KB.

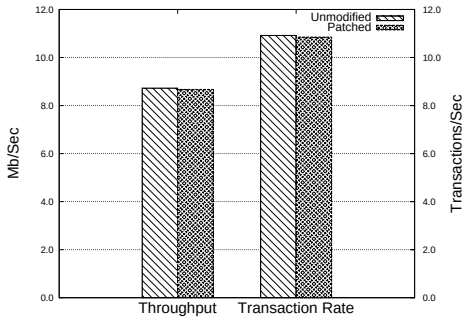
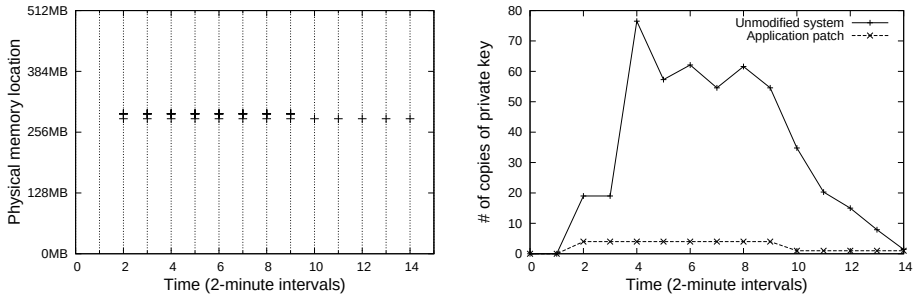



Fig. 24. OpenSSH server performances before and after adopting the integrated countermeasure

This analysis was repeated 20 times in order to obtain average measurements. We considered two metrics: *transaction rate*, namely the number of files transferred per second, and *throughput* in terms of Mbits per second transferred. Figure 24 plots the comparison. In each metric case, the left bar corresponds to the measurement before adopting the countermeasure, and the right bar corresponds to the measurement after adopting the countermeasure. It is apparent that the countermeasure has little impact on performance.

## 5.5 Effectiveness on Mitigating the Key-Flooding Problem in Apache Server

5.5.1 *Previous Kernel + Previous OpenSSL.* **Effectiveness of the application-level countermeasure**. Figure 30(a) plots the location of copies of the private key found in memory after incorporating the application-level countermeasure. Figure 30(b) shows the dramatic



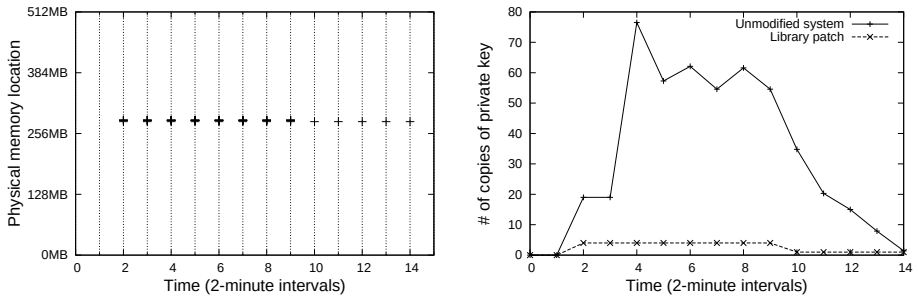

(a) Location of keys in memory (b) # of keys in memory

Fig. 25. Effectiveness of the application-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

contrast in terms of the numbers of copies of the private key found in memory between the case where the system is unmodified and the case where the system is patched. What is most interesting to note here is that during time steps 4 through 9, when traffic was varied, there is no increase in key count. Essentially, key count is no longer related to the number

of processes in memory. Again, the only key that remains in memory is the PEM-encoded private key key in the page cache.

**Effectiveness of the library-level countermeasure**. Figure 31(a) shows the location of copies of the private key found in memory after adopting the library-level countermeasure. Figure 31(b) shows the the comparison of key frequency between the unmodified case and

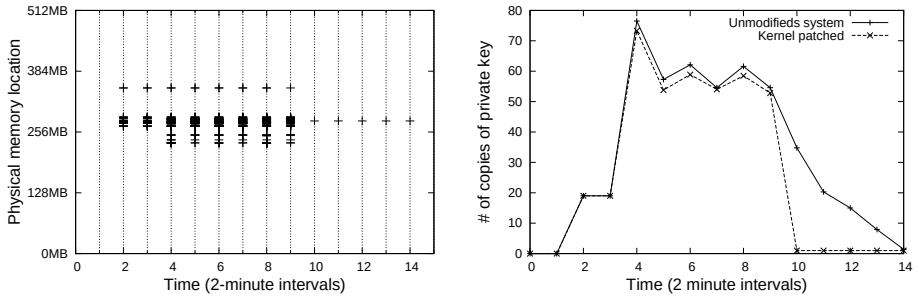



(a) Location of keys in memory (b) # of keys in memory

Fig. 26. Effectiveness of the library countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

the patched case. The results here are similar to that of the case in which the application-level countermeasure was adopted.

**Effectiveness of the kernel-level countermeasure**. Figure 32(a) shows the locations of copies of the private key found in memory after patching with the kernel-level countermeasure. Figure 32(b) shows the comparison between the unmodified and patched systems. We



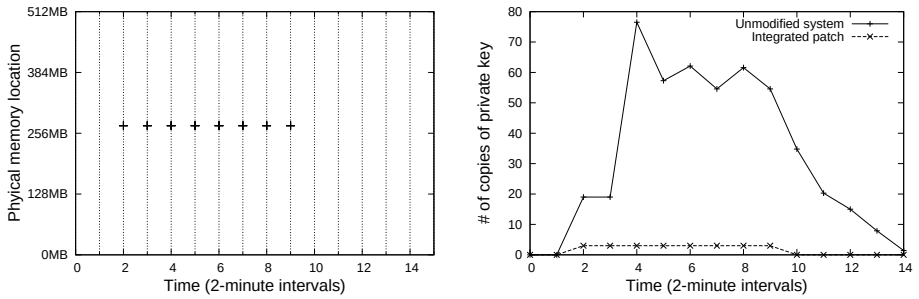

(a) Location of keys in memory (b) # of keys in memory

Fig. 27. Effectiveness of the kernel-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

observe that in the initial phase of the experiment, time steps 0 through 4, the frequency of keys in memory is almost identical. From time step 4 through time step 9, when varying

load is applied to the server, we see that the system with the kernel-level countermeasure behaves slightly better than the unmodified system, but in both cases we see that the number of keys in memory is still tightly coupled to the number of server processes. The slightly lower key count in the patched case is probably due to server processes that complete and then have their memory cleared immediately. Significant improvement can be seen from step 10 though the completion of the experiment at time step 14. In this phase, the server has been terminated. Our countermeasure proves to be very effective in clearing keys from memory as the memory pages become free. The only copy of the key present in memory is the PEM-encoded private key file residing in the page cache, where it may remain indefinitely.

**Effectiveness of the integrated countermeasure**. Figure 35(a) shows the locations of copies of the private key in memory after incorporating the integrated countermeasure. Figure 35(b) depicts the comparison between the unmodified and the patched systems.

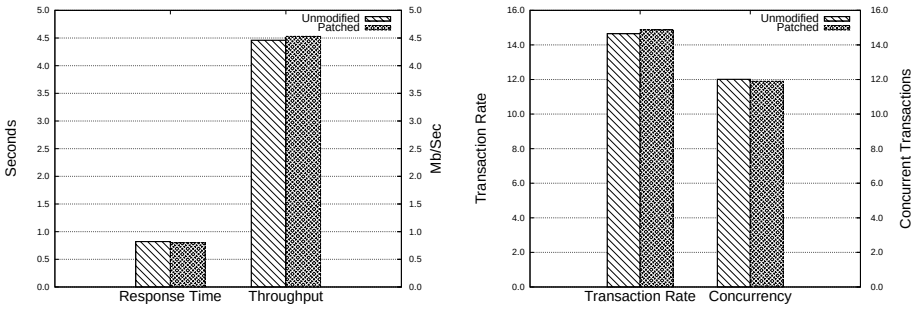



(a) Location of keys in memory (b) # of keys in memory

Fig. 28. Effectiveness of the integrated countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

Again, it is important to note that key frequency is no longer affected by process count, and that at time step 10 when the server is terminated, there are no copies of the private key in memory because the private key was flushed from the page cache immediately after it is read into the server's virtual memory.

Because we recommend the integrated countermeasure, we here report its impact on system performance. Benchmarks were taken using the open source Siege benchmarking utility, which ran on a separate client machine to initiate 4000 HTTPS transactions while attempting to maintain 20 concurrent connections throughout the experiment. CPU utilization was observed to guarantee that the benchmarks were not influenced by the client machine. We considered four metrics: *response time*, namely the average time the Apache server took to respond to each request; *throughput*, namely the average number of bytes transferred by Apache every second; *transaction rate*, namely the average number of transactions the Apache was able to handle per second; *concurrency*, namely the average number of concurrent connections throughout the experiment. Figure 34(a) plots the average response time (seconds) and throughput (bytes per second) metrics output by the Siege stress tester. Figure 34(b) plots the average transaction rate (transactions per second) and concurrency (processes) output by the Siege stress tester. These pictures clearly show that

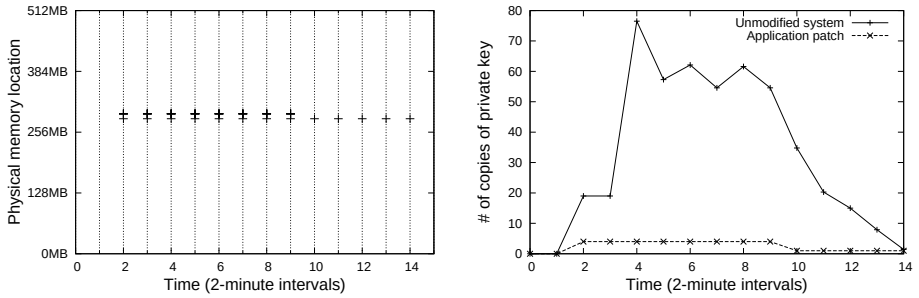



(a) *average response time* and *throughput* (b) *average transaction rate* and *concurrency*

Fig. 29. Apache server performances before and after adopting the integrated countermeasure

our modifications to the library and kernel have negligible performance impacts. We actually observe some improvements in both throughput and transaction rate, this is likely caused by our modification to the "copy on write" because we put the private key into a special memory region, which might incur less duplications of memory pages.

5.5.2 *Current Kernel + Current OpenSSL: This is just place holder right now.* **Effectiveness of the application-level countermeasure**. Figure 30(a) plots the location of copies of the private key found in memory after incorporating the application-level countermeasure. Figure 30(b) shows the dramatic contrast in terms of the numbers of copies of the



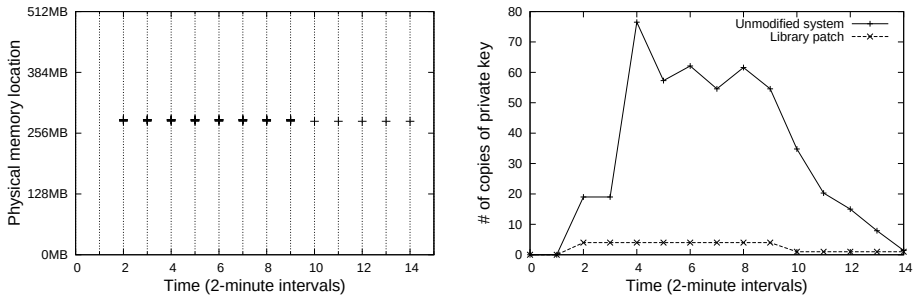

(a) Location of keys in memory (b) # of keys in memory

Fig. 30. Effectiveness of the application-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

private key found in memory between the case where the system is unmodified and the case where the system is patched. What is most interesting to note here is that during time steps 4 through 9, when traffic was varied, there is no increase in key count. Essentially, key count is no longer related to the number of processes in memory. Again, the only key that remains in memory is the PEM-encoded private key key in the page cache.

**Effectiveness of the library-level countermeasure**. Figure 31(a) shows the location of copies of the private key found in memory after adopting the library-level countermeasure.

Figure 31(b) shows the the comparison of key frequency between the unmodified case and

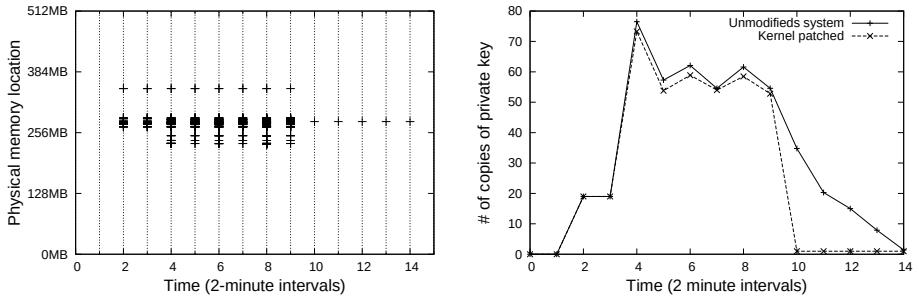



(a) Location of keys in memory (b) # of keys in memory

Fig. 31. Effectiveness of the library countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

the patched case. The results here are similar to that of the case in which the application-level countermeasure was adopted.

**Effectiveness of the kernel-level countermeasure**. Figure 32(a) shows the locations of copies of the private key found in memory after patching with the kernel-level countermeasure. Figure 32(b) shows the comparison between the unmodified and patched systems. We



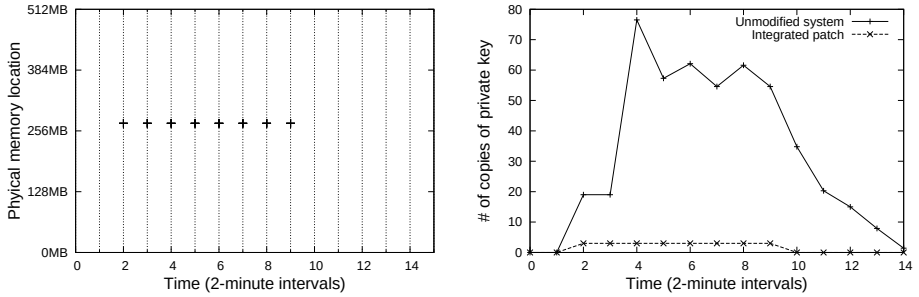

(a) Location of keys in memory (b) # of keys in memory

Fig. 32. Effectiveness of the kernel-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

observe that in the initial phase of the experiment, time steps 0 through 4, the frequency of keys in memory is almost identical. From time step 4 through time step 9, when varying load is applied to the server, we see that the system with the kernel-level countermeasure behaves slightly better than the unmodified system, but in both cases we see that the number of keys in memory is still tightly coupled to the number of server processes. The slightly lower key count in the patched case is probably due to server processes that complete and then have their memory cleared immediately. Significant improvement can be

seen from step 10 though the completion of the experiment at time step 14. In this phase, the server has been terminated. Our countermeasure proves to be very effective in clearing keys from memory as the memory pages become free. The only copy of the key present in memory is the PEM-encoded private key file residing in the page cache, where it may remain indefinitely.

**Effectiveness of the integrated countermeasure**. Figure 35(a) shows the locations of copies of the private key in memory after incorporating the integrated countermeasure. Figure 35(b) depicts the comparison between the unmodified and the patched systems.

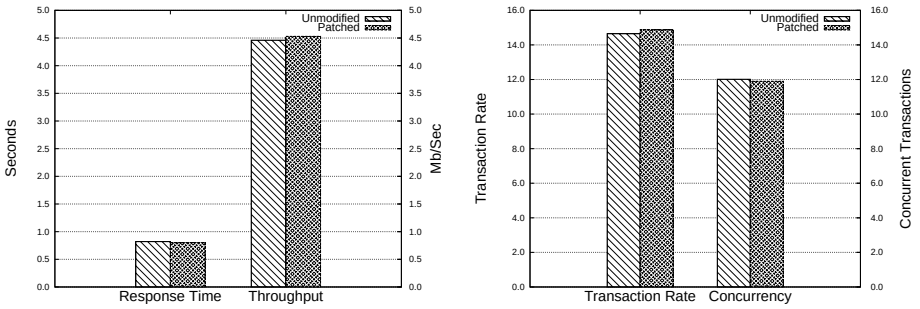



(a) Location of keys in memory (b) # of keys in memory

Fig. 33. Effectiveness of the integrated countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of Apache server. Left-hand figure: "+" denotes a copy in allocated memory and "×" denotes a copy of private key in unallocated memory.

Again, it is important to note that key frequency is no longer affected by process count, and that at time step 10 when the server is terminated, there are no copies of the private key in memory because the private key was flushed from the page cache immediately after it is read into the server's virtual memory.

Because we recommend the integrated countermeasure, we here report its impact on system performance. Benchmarks were taken using the open source Siege benchmarking utility, which ran on a separate client machine to initiate 4000 HTTPS transactions while attempting to maintain 20 concurrent connections throughout the experiment. CPU utilization was observed to guarantee that the benchmarks were not influenced by the client machine. We considered four metrics: *response time*, namely the average time the Apache server took to respond to each request; *throughput*, namely the average number of bytes transferred by Apache every second; *transaction rate*, namely the average number of transactions the Apache was able to handle per second; *concurrency*, namely the average number of concurrent connections throughout the experiment. Figure 34(a) plots the average response time (seconds) and throughput (bytes per second) metrics output by the Siege stress tester. Figure 34(b) plots the average transaction rate (transactions per second) and concurrency (processes) output by the Siege stress tester. These pictures clearly show that our modifications to the library and kernel have negligible performance impacts. We actually observe some improvements in both throughput and transaction rate, this is likely caused by our modification to the "copy on write" because we put the private key into a special memory region, which might incur less duplications of memory pages.

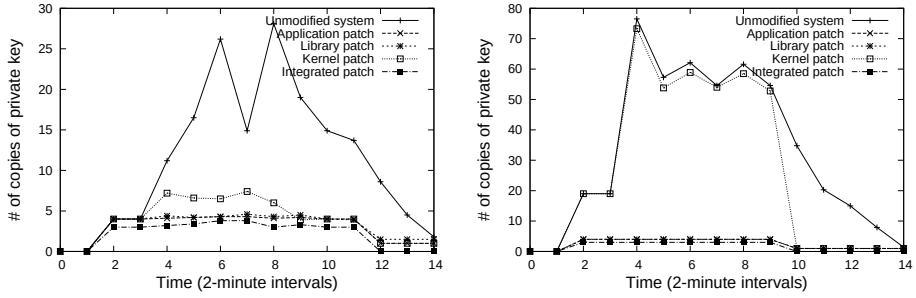



(a) *average response time* and *throughput* (b) *average transaction rate* and *concurrency*

Fig. 34. Apache server performances before and after adopting the integrated countermeasure

## 5.6 Summary

The recommended integrated countermeasure can significantly reduce the number of exposed key copies (from roughly 20-70 to no more than 3 persistent copies and no more than 2 transient copies in the experiments), when launching the attacks mentioned above. Because exposure of any copy is sufficient to compromise the private key, it is perhaps more important to look into the success rates of the attacks. Experimental results show that




(a) Location of keys in memory (b) # of keys in memory

Fig. 35. Effectiveness of the countermeasures on mitigating the key-flooding problem (averaged over 10 runs)

the countermeasure can reduce the success rates of the attacks, from roughly 60-100% to roughly 20-40% (i.e., it can drop 40-60% success rates).

# 6. ARE OPENSSH CLIENTS AND BROWSERS VULNERABLE?

## 6.1 Does the Key-Flooding Problem Apply to the Client-Ends?

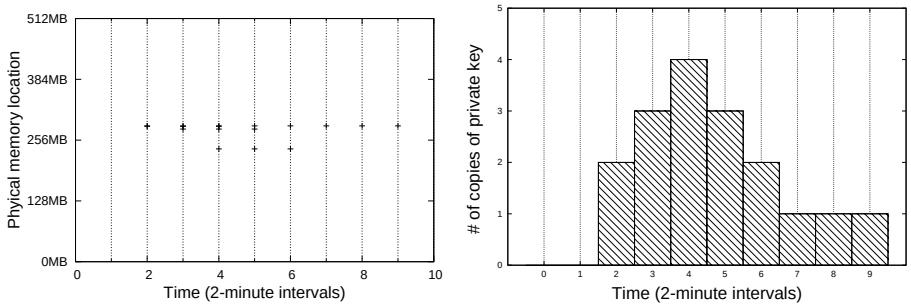
6.1.1 *Previous Kernel + Previous OpenSSL.* We want to know whether the OpenSSH client and `HTTP` browsers (with Firefox in our case study) also exhibit the key-flooding phenomenon. For this purpose, we conducted experiments with the following servers settings (the same as the above investigation of the `socket` attack): the server machine has an Intel Pentium 4 running at 2.26 GHz with 512 MB memory; the server and client are

connected via a gigabit switch; the server operating system is Ubuntu 9.04 with a 2.6.24.1 kernel. The OpenSSH server is OpenSSH 5.1_p1. The Apache server is 2.2.13.

**Experiment result with the OpenSSH client**. The OpenSSH client used a 2048-bit RSA key to authenticate to the OpenSSH server. In this experiment the OpenSSH client initiates one session to each of three different OpenSSH servers. Specifically, the experiment proceeds as follows (time unit: 2 minutes):

—Time t=0: The experiment is started.
—Time t=1: An OpenSSH client is launched.
—Time t=2: An OpenSSH connection is made to the first server.
—Time t=3: An OpenSSH connection is made to the second server.
—Time t=4: An OpenSSH connection is made to the third server.
—Time t=5: The first OpenSSH connection is terminated.
—Time t=6: The second OpenSSH connection is terminated.
—Time t=7: The third OpenSSH connection is terminated.
—Time t=8: All OpenSSH client processes are terminated.
—Time t=9: The experiment is finished.

Figure 37(a) plots the typical case of the spatial distribution of OpenSSH client's RSA private key in RAM. Figure 37(b) depicts the average (over 10 runs) number of copies of the private key in RAM with respect to time. We observe the following. At time step 0, when no OpenSSH client is launched, no key instances are found. At time step 1, when an OpenSSH client process is launched, we again do not see any instance of the private key. At time step 2, we launch an OpenSSH connection to the first the server using the private key. At this point we see an appearance of the private key, 1 in the PEM-encoded form and as well another copy. The PEM-encoded copy is held in the page cache. The other copy is an uncleared copy left after a key was used in a signing operation, which is likely overlooked in the code of OpenSSH client software. At time steps 3 and 4, we




(a) Locations of keys in memory (single instance) (b) Key frequency vs. Time (averaged over 10 runs)

Fig. 36. Experimental result with the OpenSSH client (all copies are found in allocated memory)

see an increase in the number of copies of the private key, proportional to the number of SSH connections. At time steps 5, 6 and 7, we stop the SSH connections one at a time, beginning with the oldest. This had the effect of decreasing the keys found proportionally

to the number of active connections. At time steps 8 and 9, we see that 1 copy of the key remained. It is the PEM-encoded key copy in the page cache.

The above suggests that the OpenSSH client attempts to clear the private key immediately after using it. Indeed, we observe that the OpenSSH client does clear the private key before returning the memory page to the kernel, meaning that no private key appears in the unallocated memory. However, there is consistently 1 copy of $p$ (in addition to the PEM file) in RAM even after the key is not needed. Source code analysis reveals that the OpenSSH client code overlooked the additional copy of $p$ that remains in RAM even after the authentication phase of the SSH negotiation. Specifically, this unnecessary copy of $p$ can be traced to OpenSSL library caching a copy to be used in its Montgomery calculations and, more specifically, the copy is made in BN_MONT_CTX_set with a call to BN_copy, namely RSA_eay_mod_exp -> BN_MONT_CTX_set_locked -> BN_MONT_CTX_set calls a BN_copy at line 580 of crypto/rsa/rsa_eay.c. As we will see, our application, library, and integrated countermeasures eliminated this unnecessary appearance of $p$.

**Experimental result with Firefox browser**. The latest stable binary version of Firefox that was available when the experiment was conducted was Firefox 3.5.1. Firefox was unique from the other applications in that it uses the Netscape Security Service (NSS) library rather than OpenSSL. This library has its origins in the Netscape browser and has been updated for use with Firefox. In our experiment, Firefox browser has a user's RSA private key. The iMacros plug-in was used to record and replay certain steps within the experiment, such as the opening and closing of new tabs to Apache servers at predetermined times. RAM content is searched for the RSA private key used for SSL negotiation at each of the following steps (time unit: 2 minutes).

—Time t=0: The experiment is started.

—Time t=1: Firefox client is started. It connects to an Apache server that does not require the client to authenticate (i.e., does not require the use of the private key).

—Time t=2: An HTTPS connection to a server requiring client authentication is initiated.

—Time t=3: A second HTTPS connection to another Apache server requiring client authentication is initiated in a new tab.

—Time t=4: A third HTTPS connection to yet another Apache server requiring client authentication is initiated in a new tab.

—Time t=5: The third connection is closed.

—Time t=6: The second connection is closed.

—Time t=7: The first connection is closed.

—Time t=8: The Firefox process is terminated.

—Time t=9: The experiment is completed.

No copies of the client RSA private key were present in RAM at the points in time that we searched RAM, which is possible because RAM was searched at the end of 2 minute intervals. To confirm that the client private key was indeed used, we considered a variant experiment that ran with a debugger attached to Firefox. We found the key inside cryptographic routines but only for a brief time interval. Source code analysis indicates that Firefox does in fact clear the key immediately after use, which is consistent with our experimental result that Firefox does a very good job of minimizing exposure of the RSA

private key in memory. It is nevertheless noted that this is easier for a client because it can afford to load a key only when necessary. In contrast, servers are expected to operate with high loads and low latencies, so for performance reasons we expect them to be designed to keep at least one copy of the key in memory.

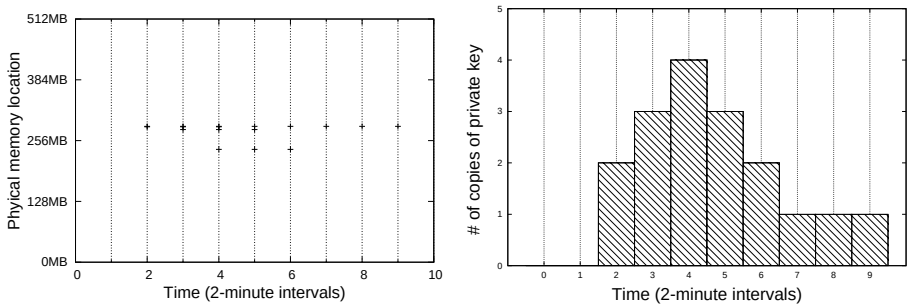
Because Mozilla Firefox browser already did an excellent job of clearing copies of a private key in memory, namely only allowing them to exist briefly when necessary, it does not appear to be much value to apply our countermeasures to it. On the other hand, the OpenSSH client did not appear to have done a good job in clearing copies of a private key in memory, it makes good sense to investigate (in what follows) whether our countermeasures can help.

6.1.2 *Current Kernel + Current OpenSSL: The content below is just place holder right now.* We want to know whether the OpenSSH client and HTTP browsers (with Firefox in our case study) also exhibit the key-flooding phenomenon. For this purpose, we conducted experiments with the following servers settings (the same as the above investigation of the socket attack): the server machine has an Intel Pentium 4 running at 2.26 GHz with 512 MB memory; the server and client are connected via a gigabit switch; the server operating system is Ubuntu 9.04 with a 2.6.24.1 kernel. The OpenSSH server is OpenSSH 5.1_p1. The Apache server is 2.2.13.

**Experiment result with the OpenSSH client**. The OpenSSH client used a 2048-bit RSA key to authenticate to the OpenSSH server. In this experiment the OpenSSH client initiates one session to each of three different OpenSSH servers. Specifically, the experiment proceeds as follows (time unit: 2 minutes):

—Time t=0: The experiment is started.
—Time t=1: An OpenSSH client is launched.
—Time t=2: An OpenSSH connection is made to the first server.
—Time t=3: An OpenSSH connection is made to the second server.
—Time t=4: An OpenSSH connection is made to the third server.
—Time t=5: The first OpenSSH connection is terminated.
—Time t=6: The second OpenSSH connection is terminated.
—Time t=7: The third OpenSSH connection is terminated.
—Time t=8: All OpenSSH client processes are terminated.
—Time t=9: The experiment is finished.

Figure 37(a) plots the typical case of the spatial distribution of OpenSSH client's RSA private key in RAM. Figure 37(b) depicts the average (over 10 runs) number of copies of the private key in RAM with respect to time. We observe the following. At time step 0, when no OpenSSH client is launched, no key instances are found. At time step 1, when an OpenSSH client process is launched, we again do not see any instance of the private key. At time step 2, we launch an OpenSSH connection to the first the server using the private key. At this point we see an appearance of the private key, 1 in the PEM-encoded form and as well another copy. The PEM-encoded copy is held in the page cache. The other copy is an uncleared copy left after a key was used in a signing operation, which is likely overlooked in the code of OpenSSH client software. At time steps 3 and 4, we see an increase in the number of copies of the private key, proportional to the number of SSH connections. At time steps 5, 6 and 7, we stop the SSH connections one at a time,




(a) Locations of keys in memory (single instance) (b) Key frequency vs. Time (averaged over 10 runs)

Fig. 37. Experimental result with the OpenSSH client (all copies are found in allocated memory)

beginning with the oldest. This had the effect of decreasing the keys found proportionally to the number of active connections. At time steps 8 and 9, we see that 1 copy of the key remained. It is the PEM-encoded key copy in the page cache.

The above suggests that the OpenSSH client attempts to clear the private key immediately after using it. Indeed, we observe that the OpenSSH client does clear the private key before returning the memory page to the kernel, meaning that no private key appears in the unallocated memory. However, there is consistently 1 copy of $p$ (in addition to the PEM file) in RAM even after the key is not needed. Source code analysis reveals that the OpenSSH client code overlooked the additional copy of $p$ that remains in RAM even after the authentication phase of the SSH negotiation. Specifically, this unnecessary copy of $p$ can be traced to OpenSSL library caching a copy to be used in its Montgomery calculations and, more specifically, the copy is made in `BN_MONT_CTX_set` with a call to `BN_copy`, namely `RSA_eay_mod_exp -> BN_MONT_CTX_set_locked -> BN_MONT_CTX_set` calls a `BN_copy` at line 580 of `crypto/rsa/rsa_eay.c`. As we will see, our application, library, and integrated countermeasures eliminated this unnecessary appearance of $p$.

**Experimental result with Firefox browser**. The latest stable binary version of Firefox that was available when the experiment was conducted was Firefox 3.5.1. Firefox was unique from the other applications in that it uses the Netscape Security Service (NSS) library rather than OpenSSL. This library has its origins in the Netscape browser and has been updated for use with Firefox. In our experiment, Firefox browser has a user's RSA private key. The iMacros plug-in was used to record and replay certain steps within the experiment, such as the opening and closing of new tabs to Apache servers at predetermined times. RAM content is searched for the RSA private key used for SSL negotiation at each of the following steps (time unit: 2 minutes).

—Time t=0: The experiment is started.

—Time t=1: Firefox client is started. It connects to an Apache server that does not require the client to authenticate (i.e., does not require the use of the private key).

—Time t=2: An HTTPS connection to a server requiring client authentication is initiated.

—Time t=3: A second HTTPS connection to another Apache server requiring client authentication is initiated in a new tab.

—Time t=4: A third HTTPS connection to yet another Apache server requiring client authentication is initiated in a new tab.

—Time t=5: The third connection is closed.

—Time t=6: The second connection is closed.

—Time t=7: The first connection is closed.

—Time t=8: The Firefox process is terminated.

—Time t=9: The experiment is completed.

No copies of the client RSA private key were present in RAM at the points in time that we searched RAM, which is possible because RAM was searched at the end of 2 minute intervals. To confirm that the client private key was indeed used, we considered a variant experiment that ran with a debugger attached to Firefox. We found the key inside cryptographic routines but only for a brief time interval. Source code analysis indicates that Firefox does in fact clear the key immediately after use, which is consistent with our experimental result that Firefox does a very good job of minimizing exposure of the RSA private key in memory. It is nevertheless noted that this is easier for a client because it can afford to load a key only when necessary. In contrast, servers are expected to operate with high loads and low latencies, so for performance reasons we expect them to be designed to keep at least one copy of the key in memory.

Because Mozilla Firefox browser already did an excellent job of clearing copies of a private key in memory, namely only allowing them to exist briefly when necessary, it does not appear to be much value to apply our countermeasures to it. On the other hand, the OpenSSH client did not appear to have done a good job in clearing copies of a private key in memory, it makes good sense to investigate (in what follows) whether our countermeasures can help.

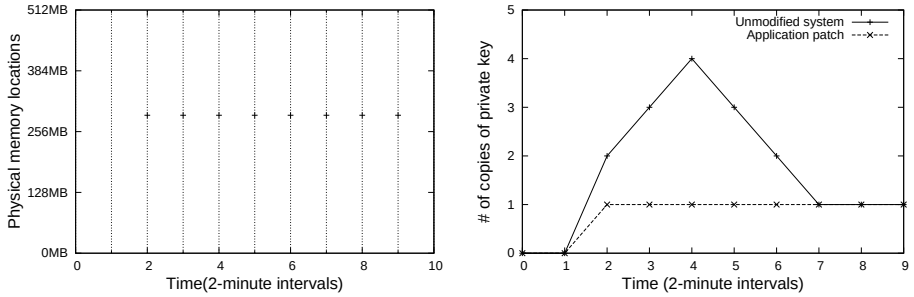
## 6.2 Mitigating the Key-Flooding Problem in OpenSSH Client

6.2.1 *Previous Kernel + Previous OpenSSL.* Because the unmodified OpenSSH client does not leave copies of a private key in unallocated memory, there was no need of the kernel-level countermeasure. However, there was evidence of keys existing in memory after they are no longer needed by OpenSSH client. Therefore the application-level, library-level and integrated countermeasures were adopted so as to determine their effectiveness on further reducing the number of copies of the private key in memory.

**Effectiveness of the application-level countermeasure**. Figure 41(b) clearly shows that we have limited the number of keys in memory to just one. It should be noted that the copy resident in memory is the PEM-encoded form of the key that is in the page cache. A key note is that the relationship between processes and keys present is no longer proportional as we had seen before. Figure 41(a) shows the location of the keys in memory.

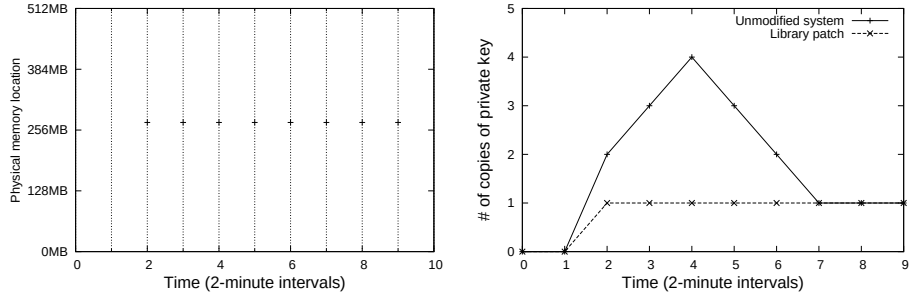
**Effectiveness of the library-level countermeasure**. In the case of the library-level countermeasure, see Figure 42(b), the results mirror that of the application-level countermeasure. This tells us that in terms of the efficacy of the patch, it does not matter where the fix is implemented. Figure 42(a) shows the location of the keys in memory.

**Effectiveness of the integrated countermeasure**. We do not plot figures for the integrated countermeasure because it effectively removes all copies of the private key from memory, at least at the sampling times. We used a debug tool to help examine the computer RAM content during the execution of OpenSSH client function `identity_sign`, we did find




(a) Location of keys in memory (b) # of keys in memory

Fig. 38. Effectiveness of the application kernel countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH client. Left-hand figure: "+" denotes a copy in unallocated memory and "×" denotes a copy of private key in allocated memory.




(a) Location of keys in memory (b) # of keys in memory

Fig. 39. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH client. Left-hand figure: "+" denotes a copy in unallocated memory and "×" denotes a copy of private key in allocated memory.

transient copies in memory (such copies appear in memory briefly). More specifically, after the unpatched OpenSSH client executes instruction `key_free`, there is still a copy of $p$ in memory while it is not needed any more. This orpha copy of $p$ is cleared by this countermeasure.

**Performance** When we consider the usage model for the OpenSSH client, performance is a much lesser concern, however we thought it prudent to evaluate how our patch affects the client's performance . Our rationale for doing performance evaluation on the integrated countermeasure still holds here. For the client, we modified the Perl script used in OpenSSH server performance evaluation. The Perl script is run on a client machine and maintains 1 process that spwans `scp` connections to transfer 10 different files the OpenSSH server consecutively until 4000 files are transferred. The 10 different files varying in size from 1 KB to 512 KB , with an average of 102.3 KB. This experiment was performed 20 times in order to obtain average measurements. We considered two metrics: *transaction rate*, namely the number of files transferred per second, and *throughput* in terms of Mbits per second transferred. Figure 43 plots the comparison. In each metric

case, the left bar corresponds to the measurement before adopting our countermeasure, and the right bar corresponds to the measurement after adopting our countermeasure. From this figure it is apparent that there is no notable performance loss or gain , when considering either metric, by implementing our countermeasure.

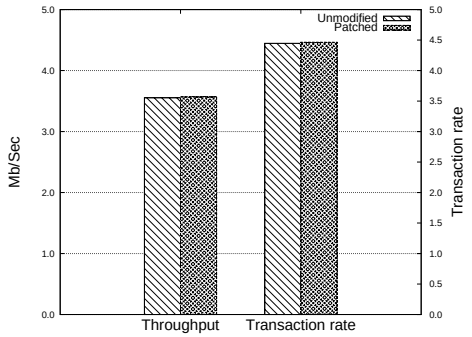
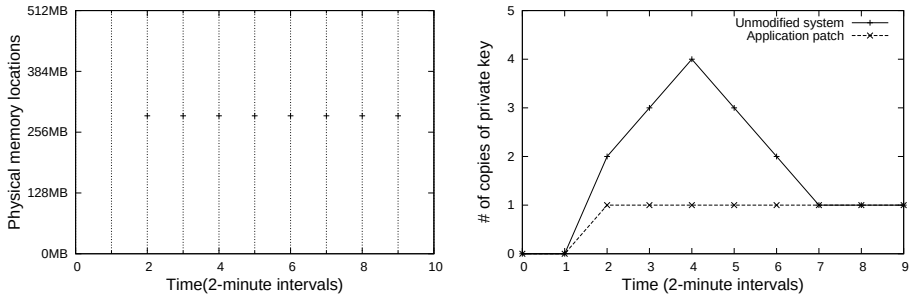



Fig. 40. OpenSSH client performances before and after adopting the integrated countermeasure

6.2.2 *Current Kernel + Current OpenSSL: The content below is just place holder right now.* Because the unmodified OpenSSH client does not leave copies of a private key in unallocated memory, there was no need of the kernel-level countermeasure. However, there was evidence of keys existing in memory after they are no longer needed by OpenSSH client. Therefore the application-level, library-level and integrated countermeasures were adopted so as to determine their effectiveness on further reducing the number of copies of the private key in memory.

**Effectiveness of the application-level countermeasure**. Figure 41(b) clearly shows that we have limited the number of keys in memory to just one. It should be noted that the copy resident in memory is the PEM-encoded form of the key that is in the page cache. A key

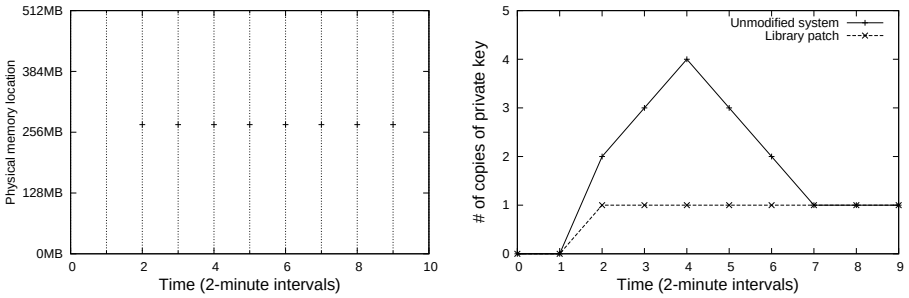



(a) Location of keys in memory (b) # of keys in memory

Fig. 41. Effectiveness of the application kernel countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH client. Left-hand figure: "+" denotes a copy in unallocated memory and "×" denotes a copy of private key in allocated memory.

note is that the relationship between processes and keys present is no longer proportional as we had seen before. Figure 41(a) shows the location of the keys in memory.

**Effectiveness of the library-level countermeasure**. In the case of the library-level countermeasure, see Figure 42(b), the results mirror that of the application-level countermeasure. This tells us that in terms of the efficacy of the patch, it does not matter where the fix




(a) Location of keys in memory (b) # of keys in memory

Fig. 42. Effectiveness of the library-level countermeasure on mitigating the key-flooding problem (averaged over 10 runs): The case of OpenSSH client. Left-hand figure: "+" denotes a copy in unallocated memory and "×" denotes a copy of private key in allocated memory.

is implemented. Figure 42(a) shows the location of the keys in memory.

**Effectiveness of the integrated countermeasure**. We do not plot figures for the integrated countermeasure because it effectively removes all copies of the private key from memory, at least at the sampling times. We used a debug tool to help examine the computer RAM content during the execution of OpenSSH client function `identity_sign`, we did find transient copies in memory (such copies appear in memory briefly). More specifically, after the unpatched OpenSSH client executes instruction `key_free`, there is still a copy of $p$ in memory while it is not needed any more. This orpha copy of $p$ is cleared by this countermeasure.

**Performance** When we consider the usage model for the OpenSSH client, performance is a much lesser concern, however we thought it prudent to evaluate how our patch affects the client's performance . Our rationale for doing performance evaluation on the integrated countermeasure still holds here. For the client, we modified the Perl script used in OpenSSH server performance evaluation. The Perl script is run on a client machine and maintains 1 process that spwans `scp` connections to transfer 10 different files the OpenSSH server consecutively until 4000 files are transferred. The 10 different files varying in size from 1 KB to 512 KB , with an average of 102.3 KB. This experiment was performed 20 times in order to obtain average measurements. We considered two metrics: *transaction rate*, namely the number of files transferred per second, and *throughput* in terms of Mbits per second transferred. Figure 43 plots the comparison. In each metric case, the left bar corresponds to the measurement before adopting our countermeasure, and the right bar corresponds to the measurement after adopting our countermeasure. From this figure it is apparent that there is no notable performance loss or gain , when considering either metric, by implementing our countermeasure.

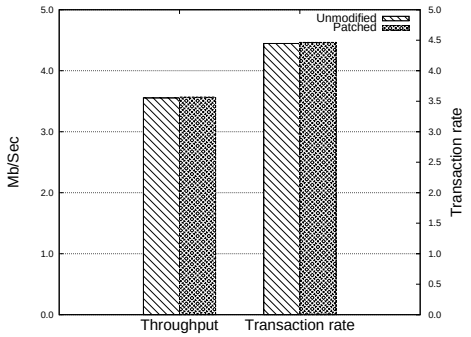



Fig. 43. OpenSSH client performances before and after adopting the integrated countermeasure

### 6.3 Summary

Because client-end is always less loaded than the server-end, it is not surprising that there are not many copues of private keys in client-end computer RAM. OpenSSH client did a good job at eliminating copies of private keys in unallocated memory, but still there are some unnecessary copies of $p$ that were not cleared. We suspect this copy is a persistent copy. On the other hand, Firefox browser did a much better job because we could not find any copy of the private key.

## 7. ELIMINATING THE TRANSIENT COPIES: PLACE HOLDER

## 8. DISCUSSION: LESSONS LEARNED

First, the flooding of persistent copies of a private key is due to the OS "copy on write" policy, which was seemingly motivated for performance purpose. Unfortunately, for applications involving cryptographic keys, this is not appropriate. Nor is it necessary because our investigation shows that the problem can be avoided essentially at no performance penalty, essentially by storing the private key in a special memory region that is not writable and thus will not be copied. The lesson learned is that when we design system for performance reason, decisions might have hidden security consequences.

Second, even in Linux kernel as recent as 2.6.28.9 (released in March 2009), computer memory is still not necessarily cleared before deallocation and private keys are still flooding in RAM.

Third, Netcape's NSS library (counterpart of OpenSSL) cleared the copies of private keys before deallocation in all cases we observed; whereas, OpenSSL was not as good as NSS library in this regard. The may be seen as evidence that open source software is not always better.

Fourth, memory disclosure attacks will always succeed if the portion of disclosed allocated memory is sufficiently large, despite our countermeasures (or any software-only countermeasure that abide by the current implementation without jeopadizing performance). Therefore, our investigation may serve as evidence that in order to completely avoid key exposures due to memory disclosure attacks, special hardware is necessary.

## 9. CONCLUSION

We showed that memory disclosure attacks are a severe threat in real-life systems. We then explored a set of mechanisms to deal with such attacks. The mechanisms have been successfully integrated into real-life systems including the OpenSSL library and the Linux kernel. Through simulation and benchmarking, we showed that our mechanisms are effective in eliminating memory disclosure attacks that disclose unallocated memory, and in mitigating the damage due to attacks that disclose portions of allocated memory.

**Acknowledgments**. We thank the anonymous reviewers of DSN'07 for their valuable comments, and our DSN'07 shepherd, Luigi Romano, for his constructive suggestions that improved the paper.

## REFERENCES

AKAVIA, A., GOLDWASSER, S., AND VAIKUNTANATHAN, V. 2009. Simultaneous hardcore bits and cryptography against memory attacks. In *6th Theory of Cryptography Conference (TCC'09)*. 474–495.

ANDERSON, R. 1997. On the forward security of digital signatures. Tech. rep.

BELLARE, M. AND MINER, S. 1999. A forward-secure digital signature scheme. In *Proc. CRYPTO 1999*. Springer-Verlag, 431–448. Lecture Notes in Computer Science No. 1666.

BELLARE, M. AND YEE, B. 2003. Forward-security in private-key cryptography. In *Proc. Cryptographer's Track - RSA Conference (CT-RSA)*. Springer-Verlag, 1–18. Lecture Notes in Computer Science No. 2612.

BOCHS. the bochs ia-32 emulator project. http://bochs.sourceforge.net/.

BROADWELL, P., HARREN, M., AND SASTRY, N. 2004. Scrash: A system for generating secure crash information. In *Proceedings of Usenix Security Symposium 2003*. ??–??

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of Usenix Security Symposium 2004*.

CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. 2005. Shredding your garbage: Reducing data lifetime. In *Proc. 14th USENIX Security Symposium*.

CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. 1999. How to forget a secret. In *Proceedings of STACS 1999*, C. Meinel and S. Tison, Eds. Lecture Notes in Computer Science, vol. 1563. Springer, 500–509.

CVE. 2008. Common vulnerabilities and exposures cve-2008-4113 (under review). `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4113`.

DESMEDT, Y. AND FRANKEL, Y. 1990. Threshold cryptosystems. In *Proc. CRYPTO 89*, G. Brassard, Ed. Springer-Verlag, 307–315. Lecture Notes in Computer Science No. 435.

DODIS, Y., KATZ, J., XU, S., AND YUNG, M. 2002. Key-insulated public key cryptosystems. In *Advances in Cryptology - EUROCRYPT 2002*, L. R. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2332. Springer, 65–82.

GUNINSKI, G. 2005. 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).

GUTMANN, P. 1996. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of Usenix Security Symposium 1996*. 77–90.

HALDERMAN, J., SCHOEN, S., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J., FELDMAN, A., APPELBAUM, J., AND FELTEN, E. 2008. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*. 45–60.

HARRISON, K. AND XU, S. 2007. Protecting cryptographic keys from memory disclosure attacks. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 137–143.

ITKIS, G. AND REYZIN, L. 2001. Forward-secure signatures with optimal signing and verifying. In *Proc. CRYPTO 2001*, J. Kilian, Ed. Lecture Notes in Computer Science, vol. 2139. Springer-Verlag, 332–354.

ITKIS, G. AND REYZIN, L. 2002. Sibir: Signer-base intrusion-resilient signatures. In *Proc. CRYPTO 2002*, M. Yung, Ed. Lecture Notes in Computer Science, vol. 2442. Springer-Verlag, 499–514.

KOCHER, P. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. CRYPTO 96*. Springer-Verlag, 104–113. LNCS 1109.

LAFON, M. AND FRANCOISE, R. 2005. 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).

MILWORM. 2008. Linux kernel ¡ 2.6.26.4 sctp kernel memory disclosure. `http://www.milw0rm.com/exploits/7618` (dated 2008).

OSTROVSKY, R. AND YUNG, M. 1991. How to withstand mobile virus attacks (extended abstract). In *Proc. ACM Principles of distributed computing*. 51–59.

PROVOS, N. 2000. Encrypting virtual memory. In *Proceedings of Usenix Security Symposium 2000*.

RIVEST, R., SHAMIR, A., AND ADLEMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2, 120–126.

SHAMIR, A. AND VAN SOMEREN, N. 1999. Playing 'hide and seek' with stored keys. In *Proceedings of Financial Cryptography 1999*. LNCS, vol. 1648. Springer, 118–124.

SIEGE. Homepage for siege tool. `http://www.joedog.org/index/siege-home`.

SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. 2001. *Operating System Concepts (sixth ed.)*. John Wiley & Sons.

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

VIEGA, J. AND MCGRAW, G. 2002. *Building Secure Software*. Addison Wesley.