FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Yuval Yarom Katrina Falkner
The University of Adelaide

Abstract

Sharing memory pages between non-trusting processes is a common method of reducing the memory footprint of multi-tenanted systems. In this paper we demonstrate that, due to a weakness in the Intel X86 processors, page sharing exposes processes to information leaks. We present FLUSH+RELOAD, a cache side-channel attack technique that exploits this weakness to monitor access to memory lines in shared pages. Unlike previous cache side-channel attacks, FLUSH+RELOAD targets the Last-Level Cache (i.e. L3 on processors with three cache levels). Consequently, the attack program and the victim do not need to share the execution core.

We demonstrate the efficacy of the FLUSH+RELOAD attack by using it to extract the private encryption keys from a victim program running GnuPG 1.4.13. We tested the attack both between two unrelated processes in a single operating system and between processes running in separate virtual machines. On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.

1 Introduction

To reduce the memory footprint of a system, the system software shares identical memory pages between processes running on the system. Such sharing can be based on the source of the page, as is the case in shared libraries [13, 26, 42]. Alternatively, the sharing can be based on actively searching and coalescing identical contents [6, 55]. To maintain the isolation between non-trusting processes, the system relies on hardware mechanisms that enforce read only or copy-on-write [13, 40] semantics for shared pages. While the processor ensures that processes cannot change the contents of shared memory pages, it sometimes fails to block other forms of inter-process interference.

One form of interference through shared pages results from the shared use of the processor cache. When a process accesses a shared page in memory, the contents of the accessed memory location is cached. Gullasch et al. [29] describes a side channel attack technique that utilises this cache behaviour to extract information on access to shared memory pages. The technique uses the processor's `clflush` instruction to evict the monitored memory locations from the cache, and then tests whether the data in these locations is back in the cache after allowing the victim program to execute a small number of instructions.

We observe that the `clflush` instruction evicts the memory line from all the cache levels, including from the shared Last-Level-Cache (LLC). Based on this observation we design the FLUSH+RELOAD attack—an extension of the Gullasch et al. attack. Unlike the original attack, FLUSH+RELOAD is a cross-core attack, allowing the spy and the victim to execute in parallel on different execution cores. FLUSH+RELOAD further extends the Gullasch et al. attack by adapting it to a virtualised environment, allowing cross-VM attacks.

Two properties of the FLUSH+RELOAD attack make it more powerful, and hence more dangerous, than prior micro-architectural side-channel attacks. The first is that the attack identifies access to specific memory lines, whereas most prior attacks identify access to larger classes of locations, such as specific cache sets. Consequently, FLUSH+RELOAD has a high fidelity, does not suffer from false positives and does not require additional processing for detecting access. While the Gullasch et al. attack also identifies access to specific memory lines, the attack frequently interrupts the victim process and as a result also suffers from false positives.

The second advantage of the FLUSH+RELOAD attack is that it focuses on the LLC, which is the cache level furthest from the processors cores (i.e., L2 in processors with two cache levels and L3 in processors with three). The LLC is shared by multiple cores on the same processor die. While some prior attacks do use the

LLC [47, 60], all of these attacks have a very low resolution and cannot, therefore, attain the fine granularity required, for example, for cryptanalysis.

To demonstrate the power of FLUSH+RELOAD we use it to mount an attack on the RSA [48] implementation of GnuPG [27]. We test the attack in two different scenarios. In the same-OS scenario both the spy and the victim execute as processes in the same operating system. In the cross-VM scenario, the spy and the victim execute in separate, co-located virtual machines. Both scenarios were tested in a local lab settings on otherwise idle machines.

By observing a single signing or decryption round, the attack extracts 98.7% of the bits on average in the same-OS scenario and 96.7% in the cross-VM scenario, with a worst case of 95% and 90%, respectively.

The rest of this paper is organised as follows. The next section presents background information on page sharing, cache architecture and the RSA encryption. Section 3 describes the FLUSH+RELOAD technique, followed by a description of our attack on GnuPG in Section 4. Mitigation techniques are presented in Section 5, and the related work in Section 6.

2 Preliminaries

2.1 Page Sharing

Sharing memory between processes can serve two different aims. It can be used as an inter-process communication mechanisms between two co-operating processes and it can be used for reducing memory footprint by avoiding replicated copies of identical contents. This paper focuses on the latter use.

When using *content-aware sharing*, identical pages are identified by the disk location the contents of the page is loaded from. This is the traditional form of sharing in an operating system, which is used for sharing the text segment of executable files between processes executing it and when using shared libraries [26]. Context-aware sharing has been suggested in early operating systems, such as Multics [42] and TENEX [13], and is implemented in all current major operating systems. This approach has also been suggested within the context of virtualisation hypervisors, such as Disco [15] and Satori [39].

Content-based page sharing, also called *memory de-duplication*, is a more aggressive form of page sharing. When using de-duplication, the system scans the active memory, identifying and coalescing unrelated pages with identical contents. De-duplication is implemented in the VMware ESX [54, 55] and PowerVM [17] hypervisors, and has also been implemented in Linux [6] and in Windows [33].

As memory pages can be shared between non co-operating processes, the system must protect the contents of the pages to prevent malicious processes from modifying the shared contents. To achieve this, the system maps shared pages as copy-on-write [13, 40]. Read operations on copy-on-write pages are permitted whereas write operations cause a CPU trap. The system software, which gains control of the CPU during the trap, copies the contents of the shared page, maps the copied page into the address space of the writing process and resumes the process.

While copy-on-write protects shared pages from modifications, it is not fully transparent. The delay introduced when modifying a shared page can be detected by processes, leading to a potential information leak attack. Such attacks have been implemented within virtualised environments for creating covert channels [58], for OS fingerprinting [44] and for detection of applications and data in other guests [49].

2.2 Cache Architecture

In addition to sharing memory pages, processes running on the same processor share the processor caches. Processor caches bridge the gap between the processing speed of modern processors and the data retrieval speed of the memory. Caches are small banks of fast memory in which the processor stores values of recently accessed memory cells. Due to locality of reference, recently used values tend to be used again. Retrieving these values from the cache saves time and reduces the pressure on the main memory.

Modern processors employ a cache hierarchy consisting of multiple caches. For example, the cache hierarchy of the Core i5-3470 processor, shown in Fig. 1, consists of three cache levels: L1, L2 and L3.

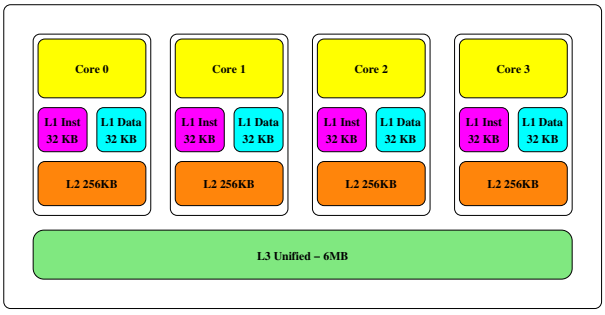

Figure 1: Intel Ivy Bridge Cache Architecture

The Core i5-3470 processor has four processing units called *cores*. Each core has a 64KB L1 cache, divided into a 32KB data cache and a 32KB instruction cache. Each core also has a 256KB L2 cache. The four cores

share a 6MB L3 cache, also known as the Last-Level Cache, or LLC.

The unit of memory in a cache is a *line* which contains a fixed number of bytes. A cache consists of multiple *cache sets* each of which stores a fixed number of cache lines. The number of cache lines in a set is the cache *associativity*. Each memory line can be cached in any of the cache lines of a single cache set. The size of cache lines in the Core i5-3470 processor is 64 bytes. The L1 and L2 caches are 8-way associative and the L3 cache is 12-way associative.

An important feature of the LLC in modern Intel processors is that it is an inclusive cache. That is, the LLC contains copies of all of the data stored in the lower cache levels. Consequently, flushing or evicting data from the LLC also remove said data from all other cache levels of the processor. Our attack exploits this cache behaviour.

Retrieving data from memory or from cache levels closer to memory takes longer than retrieving it from cache levels closer to the core. This difference in timing has been exploited for side-channel attacks. Side-channel attacks target information that an implementation of an algorithm leaks through its interaction with its environment. To exploit the timing difference, an attacker sets the cache to a known state prior to a victim operation. It can, then, use one of two methods to deduce information on the victim's operation [43]. The first method is measuring the time it takes for the victim to execute the operation. As this time depends on the state of the cache when the victim starts the operation, the attacker can deduce the cache sets accessed by the victim and, therefore, learn information on the victim [5, 9, 57]. The second approach is for the attacker to measure the time it takes for the attacker to access data after the victim's operation. This time is dependent on the cache state prior to the victim operation as well as on the changes the victim operation caused in the cache state [1, 2, 4, 14, 19, 47, 61].

Most prior work on cache side-channel attacks relies on the victim and spy executing within the same processing core. One reason for that is that many of the attacks suggested require the victim to be stopped while the spy performs the attack. To that aim, the attack is combined with an attack on the scheduler that allows the spy process to interrupt and block the victim.

Another reason for attacking within the same core is that the attacks focus on the L1 cache level, which is not shared between cores. The large size of the LLC hinders attacks both because setting it to a known state takes longer than with smaller caches and because the virtual memory used by the operating system masks the mapping of memory addresses to cache sets. Furthermore, as most of the memory activity occurs at the L1 cache level, less information can be extracted from LLC activity. Some prior works do use the LLC as an information leak channel [46, 47, 60]. However, due to the cache size, these channels have a low bandwidth.

We now proceed to describe the RSA encryption.

2.3 RSA

RSA [48] is a public-key cryptographic system that supports encryption and signing. Generating an encryption system requires the following steps:

- Randomly selecting two prime numbers p and q and calculating $n = pq$.
- Choosing a public exponent e. GnuPG uses $e = 65537$.
- Calculating a private exponent $d \equiv e^{-1} \pmod{(p-1)(q-1)}$.

The generated encryption system consists of:

- The public key is the pair (n, e).
- The private key is the triple (p, q, d).
- The encrypting function is $E(m) = m^e \bmod n$.
- The decrypting function is $D(c) = c^d \bmod n$.

CRT-RSA is a common optimisation for the implementation of the decryption function. It splits the secret key d into two parts $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, computes two parts of the message: $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. m is then computed from m_p and m_q using Garner's formula [25]:

$$h = (m_p - m_q)(q^{-1} \bmod p) \bmod p$$
$$m = m_q + hq$$

To compute the encryption and decryption functions, GnuPG versions before 4.1.14 and the related `libgcrypt` before version 1.5.3 use the square-and-multiply exponentiation algorithm [28]. Square-and-multiply computes $x = b^e \bmod m$ by scanning the bits of the binary representation of the exponent e. Given a binary representation of e as $2^{n-1}e_{n-1} + \cdots 2^0 e_0$, square-and-multiply calculates a sequence of intermediate values $x_{n-1}, \ldots, x_0$ such that $x_i = b^{\lfloor e/2^i \rfloor} \bmod m$ using the formula $x_{i-1} = {x_i}^2 b^{e_{i-1}}$. Figure 2 shows a pseudo-code implementation of square-and-multiply.

As can be seen from the implementation, computing the exponent consists of sequence of Square and Multiply operations, each followed by a Modulo Reduce. This sequence corresponds directly with the bits of the exponent. Each occurrence of Square-Reduce-Multiply-Reduce within the sequence corresponds to a bit whose value is 1. Occurrences of Square-Reduce that are not

```
 1 function exponent(b, e, m)
 2 begin
 3   x ← 1
 4   for i ← |e| − 1 downto 0 do
 5     x ← x^2
 6     x ← x mod m
 7     if (e_i = 1) then
 8       x ← xb
 9       x ← x mod m
10     endif
11   done
12   return x
13 end
```

Figure 2: Exponentiation by Square-and-Multiply

followed by a Multiply correspond to bits whose values are 0. Consequently, a spy process that can trace the execution of the square-and-multiply exponentiation algorithm can recover the exponent.

As GnuPG uses the CRT-RSA optimisation, the spy process can only hope to extract d_p and d_q. However, for an arbitrary message m, $(m - m^{ed_p})$ is a multiple of p. Hence, knowing d_p (and, symmetrically, d_q) is sufficient for factoring n and breaking the encryption [16].

3 The FLUSH+RELOAD Technique

The FLUSH+RELOAD technique is a variant of PRIME+PROBE [51] that relies on sharing pages between the spy and the victim processes. With shared pages, the spy can ensure that a specific memory line is evicted from the whole cache hierarchy. The spy uses this to monitor access to the memory line. The attack is a variation of the technique suggested by Gullasch et al. [29], which include adaptations for use in multi-core and in virtualised environments.

A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the spy reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer. Figure 3 (A) and (B) show the timing of the attack phases without and with victim access.

As shown in Fig. 3 (C), the victim access can overlap the reload phase of the spy. In such a case, the victim access will not trigger a cache fill. Instead, the victim will use the cached data from the reload phase. Consequently, the spy will miss the access.

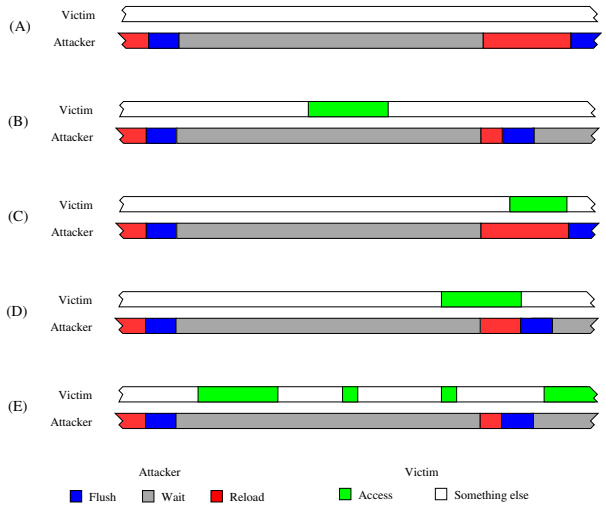

Figure 3: Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses

A similar scenario is when the reload operation partially overlaps the victim access. In this case, depicted in Fig. 3 (D), the reload phase starts while the victim is waiting for the data. The reload benefits from the victim access and terminates faster than if the data has to be loaded from memory. However, the timing may still be longer than a load from the cache.

As the victim access is independent of the execution of the spy process code, increasing the wait period reduces the probability of missing the access due to an overlap. On the other hand, increasing the wait period reduces the granularity of the attack.

One way to improve the resolution of the attack without increasing the error rate is to target memory accesses that occur frequently, such as a loop body. The attack will not be able to discern between separate accesses, but, as Fig. 3 (E) shows, the likelihood of missing the loop is small.

Several processor optimisations may result in false positives due to speculative memory accesses issued by the victim's processor [34]. These optimisations include data prefetching to exploit spatial locality and speculative execution [52]. When analysing the attack results, the attacker must be aware of these optimisations and develop strategies to filter them.

Our implementation of the attack is in Figure 4. The code measures the time to read the data at a memory address and then evicts the memory line from the cache. This measurement is implemented by the inline assembly code within the `asm` command.

The assembly code takes one input, the address, which is stored in register `%ecx`. (Line 16.) It returns the time to read this address in the register `%eax` which is stored

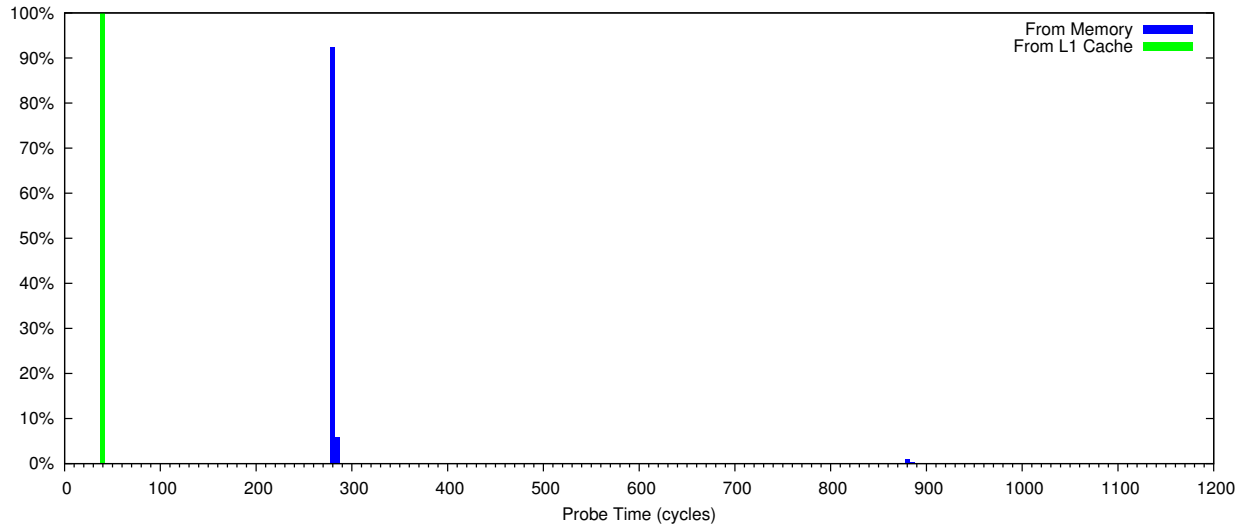

Figure 5: Distribution of Load Times.

```
 1  int probe(char *adrs) {
 2    volatile unsigned long time;
 3
 4    asm __volatile__ (
 5      "  mfence             \n"
 6      "  lfence             \n"
 7      "  rdtsc              \n"
 8      "  lfence             \n"
 9      "  movl %%eax, %%esi  \n"
10      "  movl (%1), %%eax   \n"
11      "  lfence             \n"
12      "  rdtsc              \n"
13      "  subl %%esi, %%eax  \n"
14      "  clflush 0(%1)      \n"
15      : "=a" (time)
16      : "c" (adrs)
17      :  "%esi", "%edx");
18    return time < threshold;
19  }
```

Figure 4: Code for the FLUSH+RELOAD Technique

in the variable `time`. (Line 15.)

Line 10 reads 4 bytes from the memory address in `%ecx`, i.e. the address pointed by `adrs`. To measure the time it takes to perform this read, we use the processor's time stamp counter.

The `rdtsc` instruction in line 7 reads the 64-bit counter, returning the low 32 bits of the counter in `%eax` and the high 32 bits in `%edx`. As the times we measure are short, we treat it as a 32 bit counter, ignoring the 32 most significant bits in `%edx`. Line 9 copies the counter to `%esi`.

After reading the memory, the time stamp counter is read again. (Line 12.) Line 13 subtracts the value of the counter before the memory read from the value after the read, leaving the result in the output register `%eax`.

The crux of the technique is the ability to evict specific *memory lines* from the cache. This is the function of the `clflush` instruction in line 14. The `clflush` instruction evicts the specific memory line from *all* the cache hierarchy, including the L1 and L2 caches of all cores. Evicting the line from all cores ensures that the next time the victim accesses the memory line it will be loaded into L3.

The purpose of the `mfence` and `lfence` instructions in lines 5, 6, 8 and 11 is to serialise the instruction stream. The processor may execute instructions in parallel or out of order. Without serialisation, instructions surrounding the measured code segment may be executed within that segment.

The `lfence` instruction performs partial serialisation. It ensures that load instructions preceding it have completed before it is executed and that no instruction following it executes before the `lfence` instruction. The `mfence` instruction orders all memory access, fence instructions and the `clflush` instruction. It is not, however, ordered with respect to other instructions and is, therefore, not sufficient to ensure ordering.

Intel recommends using the serialising instruction `cpuid` for that purpose [45]. However, in virtualised environments the hypervisor emulates the `cpuid` instruction. This software emulation takes too long (over 1,000 cycles) to provide the fine granularity required for the attack.

Line 18 compares the time difference between the two `rdtsc` instructions against a predetermined threshold. Loads shorter than the threshold are presumed to be served from the cache, indicating that another process has accessed the memory line since it was last flushed

from the cache. Loads longer than the threshold are presumed to be served from the memory, indicating no access to the memory line.

The threshold used in the attack is system dependent. To find the threshold for our test systems, we used the measurement code of the probe in Listing 4 to measure load times from memory and from the L1 cache level. (To measure the L1 times we removed the `clflush` instruction in line 14.) The results of 100,000 measurements of each on an HP Elite 8300 with an i5-3470 processor, running CentOS 6.5 are presented in Figure 5.

Virtually all loads from the L1 cache measure 44 cycles. (Note that this measure includes an overhead for the `rdtsc` and the fence instructions and is, therefore, much longer than a single load instruction.) Loads from memory show less constant timing. Over 98% of those take between 270 and 290 cycles. The rest are mostly spread around 880 cycles with about 200 loads measured 1140–1175 cycles. No loads from memory measured less than 200 cycles.

The timings of load operations depend on both the system architecture and the software environment. For example, on a Dell PowerEdge T420 with Xeon E5-2430 processors, loads from L1 take between 33 and 43 cycles and loads from memory take around 230 cycles. On the same architecture, within a KVM [37] guest, about 0.02% of the loads from memory take over 6,000 cycles. We believe these are caused by hypervisor activity.

The L1 measurements underestimate the probe time for data that the victim accesses. In an attack, data the victim accesses is read from the L3 cache. Intel documentation [34] states that the difference is between 22 and 39 cycles. Based on the measurement results and the Intel documentation we set the threshold to 120 cycles.

To use the FLUSH+RELOAD technique the spy and the victim processes need to share both the cache hierarchy and memory pages. In a non-virtualised environment, to share the cache hierarchy, the attacker needs the ability to execute software on the victim machine. The attacker, however, does not need elevated privileges on the victim machine. For a virtualised environment, the attacker needs access to a guest co-located on the same host as the victim guest. Techniques for achieving co-location are described by Ristenpart et al. [47]. Identifying the OS and software version in co-resident guests has been dealt with in past research [44, 49].

For sharing memory pages in system that use content-aware sharing, the attacker needs read access to the attacked executable or shared libraries. In systems that support de-duplication the attacker needs access to a copy of the attacked files. De-duplication will coalesce pages from these copies with pages from the attacked files.

4 Attacking GnuPG

In this section we describe how we use the FLUSH+RELOAD technique to extract the components of the private key from the GnuPG implementation of RSA.

We tested the attack on two hardware platforms: an HP Elite 8300, which features an Intel Core i5-3470 processor and 8GB DDR3-1600 memory and a Dell PowerEdge T420, with two Xeon E5-2430 processors and 32GB DDR3-1333 memory. On each hardware platform we experimented with two scenarios. The same-OS scenario tests the attack between two unrelated processes in the same operating system while the cross-VM scenario demonstrates that the attack works across the virtual machine isolation boundary in virtualised environments.

The same-OS tests use CentOS 6.5 Linux running on the hardware. The spy and the victim execute as two processes within that system. To achieve sharing, the spy `mmaps` the victim's executable file into the spy's virtual address space. As the Linux loader maps executable files into the process when executing them, the spy and the victim share the memory image of the mapped file. On the Dell machine we set the CPU affinity of the processes to ensure that both the victim and the spy execute on the same physical processor. We do let the processes float between the cores of the processor.

For the cross-VM scenario we used two different hypervisors: VMware ESXi 5.1 on the HP machine and Centos 6.5 with KVM on the Dell machine. In each hypervisor we created two virtual machines, one for the victim and the other for the spy. The virtual machines run CentOS 6.5 Linux. In this scenario, the spy `mmaps` a copy of the victim's executable file. Sharing is achieved through the page de-duplication mechanisms of the hypervisors. As in the same-OS scenario, on the Dell machine we set the CPU affinity of the virtual machines to ensure execution on the same physical processor.

When a pages is shared, all of the page entries in the virtual address spaces of the sharing processes map to the same physical page. As the LLC is physically tagged, entries in the cache depend only on the physical address of the shared page with no dependency on the virtual addresses in which the page is mapped. Consequently, we do not need to take care of the virtual to physical address mapping and the attack is oblivious to some diversification techniques, such as Address Space Layout Randomization (ASLR) [50].

The approach we take is to trace the execution of the victim program. For that, the spy program applies the FLUSH+RELOAD technique to memory locations within the victim's code segment. This, effectively, places probes within the victim program that are triggered whenever the victim executes the code in the probed memory lines. Tracing the execution allows the

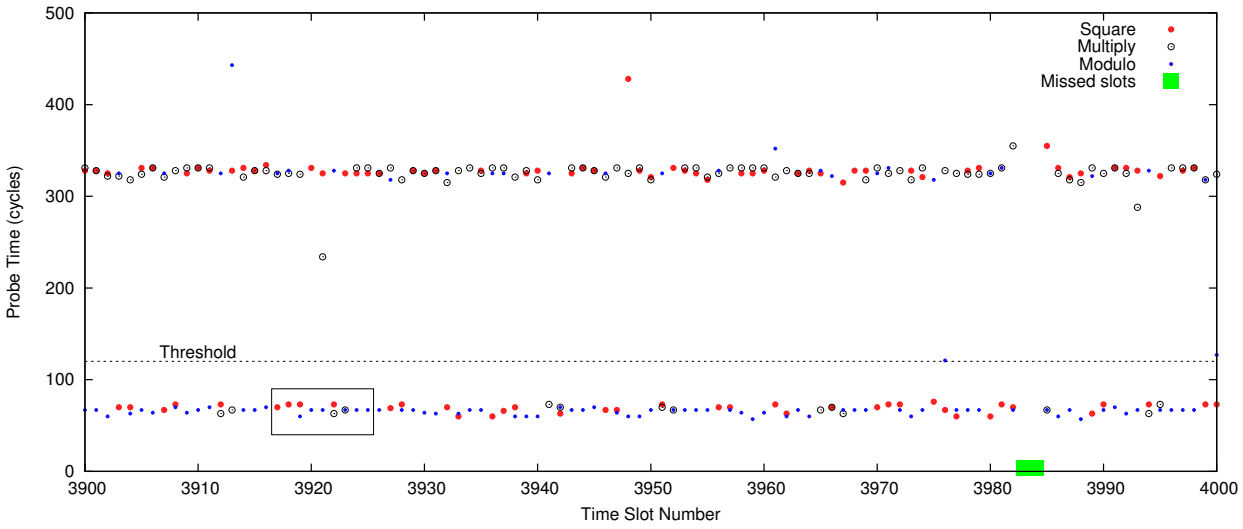

Figure 6: Time measurements of probes

spy program to infer the internal state of the victim program.

To implement the trace, the spy program divides time into fixed slots of 2,500 cycles each. In each slot it probes one memory line of the code of each of the square, multiply and modulo reduce calculations. To increase the chance of a probe capturing the access, we selected memory lines that are executed frequently during the calculation. Furthermore, to reduce the effect of speculative execution, we avoided memory lines near the beginning of the respective functions. After probing the memory lines, the spy program flushes the lines from the cache and busy waits to the end of the time slot.

We used the default build of the gpg program, which includes optimisation at -O2 level and which leaves the debugging symbols in the executable. We use the debugging symbols to facilitate the mapping of source code lines to memory addresses. In most distributions, the GnuPG executable is stripped and does not include these symbols. Attacks against stripped executables would require some reverse engineering [20] to recover this mapping. As the debugging symbols are not loaded in run time, these do not affect the victim's performance.

Measurement times for 100 time slots of the GnuPG signing with a 2,048 bit key are displayed in Figure 6. In each time slot, the spy flushes and then measures the time to read the memory lines in the Square, Multiply and Reduce functions. Measurements under the threshold indicate victim access to the respective memory lines. The exponentiations for signing takes a total of 15,690 slots or about 18ms. The CRT components used for exponentiation are 1,022 and 1,023 bits long.

Figure 7 is an enlarged view of the boxed section

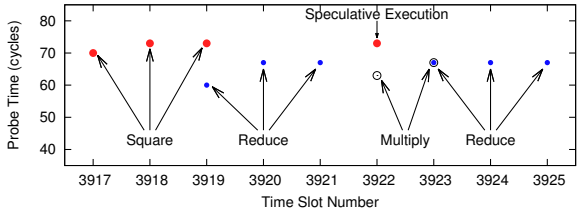

Figure 7: Section of Fig. 6

in Fig. 6. As the displayed area is below the threshold, the diagram only displays the memory lines that were retrieved from the cache, showing the activity of the GnuPG encryption. The steps of the exponentiation are clearly visible in the diagram. For example, between time slots 3,917 and 3,919 the victim was calculating a square, Time slots 3,919–3,921 are for modulo reduce calculation, multiplication in slots 3,922–3,923, and another modulo reduce in 3,923–3,925. A sequence of Square-Reduce-Multiply-Reduce indicates that during these time slots the victim was processing a set bit.

Figure 7 also demonstrates the effects of speculative execution. To improve performance, the processors tries to predict future behaviour of the program. When predicting the behaviour of the test of the bit value (Line 7 in Fig. 2), the processor does not know the value of the bit. Instead of waiting for the value to be calculated, the processor speculates that the bit might be clear and starts bringing memory lines required for the square calculation into the cache. As a result, cache lines that are part of the square calculation in Line 5 are brought into the cache, and are captured by the spy.

We have witnessed speculative execution on both the

HP and the Dell machines. Moving the probes to cache lines closer to the end of the probed functions eliminates the effects of speculative execution on the HP machine. However, speculative execution is still evident on the Dell machine.

By recognising sequences of operations, an attacker can recover the bits of the exponent. Sequences of Square-Reduce-Multiply-Reduce indicate a set bit. Sequences of Square-Reduce which are not followed by Multiply indicate a clear bit. For example, in Fig. 6, between time slots 3,903 and 3,906 the calculated sequence is Square-Reduce, which is followed by a Square, indicating that in these time slots the victim was processing a clear bit.

Continuing throughout Fig. 6 we find that the bit sequence processed in this sample is 0110011010011. Table 1 shows the time slots corresponding to each bit.

Table 1: Time Slots for Bit Sequence

Seq.	Time Slots	Value
1	3,903–3,906	0
2	3,907–3,916	1
3	3,917–3,926	1
4	3,927–3,931	0
5	3,932–3,935	0
6	3,936–3,945	1
7	3,946–3,955	1

Seq.	Time Slots	Value
8	3,956–3,960	0
9	3,961–3,969	1
10	3,970–3,974	0
11	3,975–3,979	0
12	3,980–3,988	1
13	3,989–3,998	1

System activity may cause the spy to miss time slots. The spy identifies missed time slot by noting jumps in the cycle counter. For example, In the run used for generating Fig. 6, the spy missed time slots 3,983 and 3,984. In this instance, the missed bits were not enough to hide the information on the bit processed during these time slots. However, if more slots are missed, data on bits of the private key exponent will be lost resulting in capture errors.

To measure the prevalence of capture errors, we used our spy program to observe and capture 1,000 signatures on each of the test configurations. We used a single invocation of a spy program to capture all the signatures in each system configuration. The GnuPG victim was executed from a shell in another window. Except for ensuring that the spy executes while running the signatures, the executions of the spy and of GnuPG are not synchronised.

For each observed signature, the spy outputs a text line representing the observed probes in each time slot. We used a shell script to parse this output and compared the results against the ground truth. The results are summarised in Table 2 and in Fig. 8. (For clarity, we trim Fig. 8 at 30% and 100 erroneous bits. A total of 15 samples have capture errors of more than 100 bits and the probability of no errors for the HP-CentOS configuration is 33%.)

Table 2: Statistics on Bit Errors in Capture

Hardware	HP Elite 8300		Dell PowerEdge T420	
Software	CentOS	VMware	CentOS	KVM
Average	1.41	26.55	25.12	66.12
Median	1	25	24	65
Max	15	196	96	190

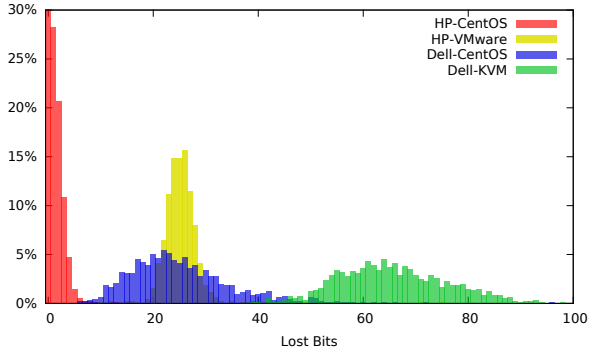

Figure 8: Distribution of Bit Errors in Capture

The shell script overestimates the number of errors. For example, due to the missing time slots, the script does not identify bit 12 in Table 1. We have manually inspected a few samples of capture output and estimate that manual inspection can reduce the number of errors by 25%-50%. Yet, the use of an automated script allows us to examine a large number of results.

On the HP machine we observe better results and significantly less noise than on the Dell machine. We believe this to be a consequence of the more advanced optimisations of the Xeon processor of the Dell machine. On each machine, results for the same-OS configuration are better than those for the cross-VM attack due to the added processing of the virtualisation layer.

Even accounting for the better results expected from manual inspection, the number of errors may be too big for a naïve brute force attack. Several strategies can be used to reduce the search space and to recover the private key. One such strategy is to rely on the nature of CRT-RSA exponentiation. As discussed in Section 2.3, an attacker only needs to recover one of the CRT components to break the encryption. By attacking the CRT component that has less errors, the attacker can reduce the search space to a more manageable size. Table 3 and Fig. 9 show the distribution of erroneous bits in the better captured CRT component in each signature. As these demonstrate, the search space is significantly reduced.

Several algorithms have been suggested for recovering the RSA exponent from partial information on the exponent bits [30, 31, 46]. These algorithms require between 27% and 70% of the bits of the exponent to recover the system key. While our attack reveals over 90% of the bits, it does not always recover the positions of

Table 3: Statistics on Bit Errors in the Better Captured CRT Component

Hardware	HP Elite 8300		Dell PowerEdge T420	
Software	CentOS	VMware	CentOS	KVM
Average	0.20	11.75	7.11	28.66
Median	0	12	6	28
Max	4	68	26	47

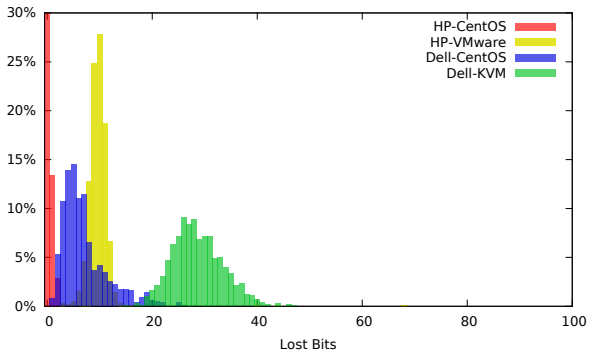

Figure 9: Distribution of Bit Errors in the Better Captured CRT Component

these bits. E.g. when a sequence of about 10 time slots is missed, this sequence can cover either one set bit or two clear bits. The attacker cannot, therefore, determine the bit positions of the following bits. Further research is required to determine whether these algorithms can be adapted to the data our attack recovers.

Another approach for recovering the key is to combine data from multiple signatures. As the positions of errors in each capture are independent, there is a small likelihood that any two captures will have errors in the same bit positions. To test this approach we manually merged the output of several pairs of observations of the spy under the Dell cross-VM scenario. When merging random pairs, we had at most a one bit error in the merged results. When merging the worst capture for the Dell cross-VM scenario with a random capture, the merged results had six bit errors, all of them in one of the CRT components and all have been identified during the process as potential errors. We, therefore, conclude that by observing two signatures, the attacker can recover the private key.

While the attack is very effective in recovering exponent bits, it does have some limitations. For the attack to work, the spy and the victim must execute on the same physical processor. For our testing, we set the processor affinity on the multi-processor system. However, in a real attack scenario the attack depends on the system scheduler.

When performing the tests, the spy and the victim were the only load on the system. Such a scenario is not representative of a real system where multiple processes are running. We expect such load to create noise that will affect the quality of capture. Furthermore, for a load that includes multiple parallel instances of GnuPG, the spy will be unable to distinguish between memory access of each instance and will be unable to recover any data.

Another limitation is the length of the secret key. On the Dell machine, probing three memory locations takes about 2,200 cycles. Hence, the attack cannot work with time slots shorter than that. With shorter key lengths, time slots of 2,200 cycles or more do not provide enough resolution to trace the victim. Consequently, recovering the private key is more difficult with shorter keys, supporting the results of Walter [56].

5 Mitigation Techniques

The attack presented here is a real, immediate threat to computer security. It, therefore, raises the very pertinent question of countermeasures. The FLUSH+RELOAD attack relies on a combination of four factors for its operation: data flow from sensitive data to memory access patterns, memory sharing between the spy and the victim, accurate, high-resolution time measurements and the unfettered use of the `clflush` instruction. Preventing any of these blocks the attack.

The lack of permission checks for using the `clflush` instruction is a weakness of the X86 architecture. Consequently, the most complete solution to the problem is to limit the power of the `clflush` instruction. The main use of the `clflush` instruction is to enforce memory coherence, e.g. when using devices that do not support memory coherence [34]. Another potential use of the instruction is to control the use of the cache for improving program performance, e.g. by flushing lines that the program knows it will not require. However, we are not aware of any actual use of the instruction for this purpose.

As the first use is, clearly, a system function and the second is based on the assumption that no other process has access to the data, we suggest restricting the use of `clflush` to memory pages to which the process has write access and to memory pages to which the system allows `clflush` access. This access control could be implemented by adding memory types that restrict flush access to the PAT (Page Attribute Table) [35, chap. 11].

The ARM architecture [7] also includes instructions to evict cache lines. However, these instructions can only be used when the processor is in an elevated privilege mode. As such, the ARM architecture does not allow user process to selectively evict memory lines and the FLUSH+RELOAD is not applicable in this architecture.

Our attack seems not to work on contemporary AMD processors, such as the A10-6800K and Opteron 6348. The code in Fig. 5 returns the same result with and without the `clflush` instruction. Replacing the second

`rdtsc` instruction (Line 12) with the similar `rdtscp` instruction fixes this issue, however, two problems prevent the use of the technique. The first problem is that data seems to linger in the cache for some time after being evicted. The second problem is that the attack does not capture accesses from other processes. A possible explanation for this behaviour is that the AMD caches are non-inclusive, i.e. data in L1 does not need to also be in L2 or L3, as is the case with the Intel caches. Consequently, evicting data from the LLC does not, necessarily, evicts it from the L1 caches of other cores. Processes executing on other cores can access data in the L1 cache without triggering a load from memory to the LLC. The attack does work on older AMD processors, such as the Opteron 2212.

Hardware based countermeasures, such as those described above cannot provide an immediate solution to the problem. They will take time to develop and will not protect existing hardware. Consequently, for immediate mitigation of the attack, software-based solutions are required.

Another possible solution is preventing sharing between the spy and the victim. Preventing page sharing between processes provides protection against the FLUSH+RELOAD attack. However, this approach goes against the trend of increased sharing in operating systems and virtualisation hypervisors. Completely eliminating page sharing would significantly increase the memory requirements of modern operating systems and is, therefore, unlikely to be a feasible solution. As a partial solution, it may be possible to avoid sharing of sensitive code by changing the program loader. Another partial solution is disabling page de-duplication, which prevents using the FLUSH+RELOAD attack between co-hosted guests in a virtualised system. This approach is recommended for public compute clouds which offer the implied promise that guests cannot interfere with each other.

Software diversification [24] is a collection of techniques that permute the locations of objects within the address spaces of processes. While most of these techniques were originally developed as a protection against memory corruption attacks, some of them can be used to prevent sharing and, consequently, to mitigate the FLUSH+RELOAD attack. More specifically, in virtualised environments, static reordering of code and data [12,24,36] can be used to create unique copies of programs in each virtual machines. As these copies are not available outside the specific virtual machine, pages of the program are not de-duplicated and sharing is prevented. Diversifying the program at run time [22] can prevent sharing of the program text even when the attacker has access to the binary file. As discussed above, the FLUSH+RELOAD technique is oblivious to the virtual to physical address mapping. Consequently, diversification techniques that rely on permuting the virtual address mapping of code pages, such as [50, 59], do not provide any protection against the attack.

FLUSH+RELOAD, like other side-channel attacks, relies on the availability of a high-resolution clock. Reducing the resolution of the clock or introducing noise to clock measurement [32,53] can be an used as a countermeasure against the attack. The main limitation of this approach is that the attacker can use other methods for generating high resolution clocks. Examples include using data from the network or running a 'clock' process in a separate execution core.

Irrespective of the measures described above, cryptographic software should be protected against the attack. Following our disclosure [18, 38], the GnuPG team released GnuPG version 1.4.14 and `libgcrypt` version 1.5.3. These mitigate the attack using the square-and-multiply-always [21] algorithm, shown in Listing 10. The algorithm executes the square and the multiply steps for each bit, but ignores the result of the multiply step for bits of value 0.

```
function exponent(b, e, m)
begin
  x ← 1
  for i ← |e| − 1 downto 0 do
    x ← x^2
    x ← x mod m
    x' ← xb
    x' ← x' mod m
    if (e_i = 1) then
      x = x'
    endif
  done
  return x
end
```

Figure 10: Exponentiation by Square-and-Multiply-Always

When introducing instructions with no effect, care should be taken to prevent the compiler from optimising these away. In the case of the GnuPG fix, the optimiser cannot know that the added addition does not have side-effects. With the possibility of side-effects, the optimiser takes a conservative approach and invokes the function.

The implementation still contains a small section of code that depends on the value of the bit, which could, theoretically, be exploited by a cache side-channel attack. However, due to speculative execution, the processor is likely to access the section irrespective of the value of the bit. Furthermore, as this section is short and is smaller than a cache line, it is likely to fit within the same cache line as the preceding or following code. Hence, we believe that this implementation protects against the FLUSH+RELOAD attack.

This fix, however, does not protect against other forms of side-channel attack. In particular, the code is likely to be vulnerable to Branch Prediction Analysis [3]. Furthermore, as access patterns to data depend on the values of the exponent bits, the code is likely to be vulnerable to PRIME+PROBE attacks [51,61]. Like FLUSH+RELOAD, these side-channel attacks rely on data flow from secret exponent bits to memory access patterns. These attacks can be prevented by using *constant time* exponentiation, where the sequence of instructions and memory locations accessed are fixed and do not depend on the value of the exponent bits. Techniques for constant time computation have been explored in the NaCl cryptographic library [10]. The pattern of accesses to memory lines of the OpenSSL [41] implementation of RSA exponentiation is not dependent on secret exponent bits. Consequently, even though the implementation is not constant time [11], it is not vulnerable to our attack.

Constant time computation is not, however, a panacea for the problem of side-channel attacks. FLUSH+RELOAD can be applied no extract secret data from non cryptographic software. For such software, the performance costs of constant-time computation are unreasonable, hence other solutions are required.

6 Related Work

Several works have pointed out that page sharing exposes guests to information leakage, which can be exploited for implementing covert channels [58], OS fingerprinting [44] and for detecting applications and data in other guests [49]. These works exploit the copy-on-write feature of page sharing. Copy-on-write introduces a significant delay when a page is copied. Hence, by timing write operations on pages, a spy can deduce the existence of pages with identical contents in other guests. As page de-duplication is a slow process, all these attacks have a very low resolution.

Using a cache side-channel to trace the execution of a program is not a new idea [1, 2, 4, 14, 19, 29, 61]. In all of these attacks, the victim and the spy must share the execution core, either by using hyper-threading or by interleaving the execution of the victim and the spy on the same core.

Gullasch et al. [29] describes an attack on AES which traces the victim's access to the S-Boxes. Our work builds on the attack technique presented by Gullasch et al. and extends it in two ways. Gullasch et al. only applies the attack on a time-shared core and does not exploit the eviction from a shared LLC. Our attack exposes the use of a shared LLC and demonstrates that the technique can be used across cores. Additionally, Gullasch et al. uses the `cpuid` instruction to synchronise the instruction stream whereas we use fence instructions. In virtualised environments, the `cpuid` is emulated in software and this emulation takes over 1,000 cycles. With two `cpuid` instructions in each probe, the Gullasch et al. probe spans over 2,500 cycles. As our attack requires three probes within 2,500 cycles, the resolution of the Gullasch et al. code is is not high enough for implementing our cross-VM attack.

The attack in Zhang et al. [61] specifically targets virtualised environments, extracting the private ElGamal [23] key of a GnuPG decryption executing in another guest. The attack depends on a weakness in the scheduler of the Xen hypervisor [8]. The granularity of the attack is one probe in 50,000 cycles, limiting the minimum size of victim key that can be captured. The modulus in the paper is 4,096 bits long. The attack has low signal to noise ratio, and requires the use of filtering. Even with this filtering and the large modulus, the attack requires six hours of constant decryption to recover the key.

Weiß et al. [57] also describes cache timing attack in a virtualised environment. The attack is an adaptation of Bernstein's attack [9] that relies on the short constant communication time between domains in the L4 kernel.

7 Conclusions

In this paper we describe the FLUSH+RELOAD technique and how we use it to extract GnuPG private keys across multiple processor cores and across virtual machine boundaries.

It is hard to overstate the severity of the attack, both in virtualised and in non-virtualised environments. GnuPG is a very popular cryptographic package. It is used as the cryptography module of many open-source projects and is used, for example, for email, file and communication encryption. Hence, vulnerable versions of GnuPG are not safe for multi-tenant systems or for any system that may run untrusted code.

While significant, the attack on GnuPG is only a demonstration of the power of the FLUSH+RELOAD technique. The technique is generic and can be used to monitor other software. It can be used to devise other types of attacks on cryptographic software. It can also be used against other types of software. For example, it could be used to collect statistical data on network traffic by monitoring network handling code or it could monitor keyboard drivers to collect keystroke timing information.

Hence, while the GnuPG team has fixed the vulnerability in their software, their fix does not address the broader threat exposed by this paper.

The FLUSH+RELOAD technique exploits the lack of restrictions on the use of the `clflush` instruction. Not restricting the use of the instruction is a security weakness of the Intel implementation of the X86 architecture. This enables processes to interact using read-only pages.

Addressing this weakness requires a hardware fix, which, unless implemented as a microcode update, will not be applicable to existing hardware.

Preventing page sharing also blocks the FLUSH+RELOAD technique. Given the strength of the attack, we believe that the memory saved by sharing pages in a virtualised environment does not justify the breach in the isolation between guests. We, therefore, recommend that memory de-duplication be switched off.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Thomas Ristenpart, for their valuable comments and support.

This research was performed under contract to the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia.

References

[1] ACIIÇMEZ, O. Yet another microarchitectural attack: exploiting I-Cache. In *Proceedings of the ACM Workshop on Computer Security Architecture* (Fairfax, Virginia, United States, November 2007), P. Ning and V. Atluri, Eds., pp. 11–18.

[2] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems* (Santa Barbara, California, United States, April 2010), S. Mangard and F.-X. Standaert, Eds., pp. 110–124.

[3] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the Second ACM Symposium on Information, Computer and Communication Security* (Singapore, March 2007), pp. 312–320.

[4] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the Cryptographers' Track at the RSA Conference* (San Francisco, California, United States, April 2008), T. Malkin, Ed., pp. 256–273.

[5] ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, Ç. K. Cache based remote timing attacks on the AES. In *Proceedings of the Cryptographers' Track at the RSA Conference* (San Francisco, California, United States, February 2007), M. Abe, Ed., pp. 271–286.

[6] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium* (Montreal, Quebec, Canada, July 2009), pp. 19–28.

[7] *ARM Architecture Reference Manual*, ARMv7-A and ARMv7-R ed., 2012.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art or virtualization. In *Proceedings of the Ninteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, New York, United States, October 2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164–177.

[9] BERNSTEIN, D. J. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.

[10] BERNSTEIN, D. J., LANGE, T., AND SCHWABE, P. The security impact of a new cryptographic library. In *Proceedings of the Second International Conference on Cryptology and Information Security in Latin America* (Santiago, Chile, October 2012), A. Hevia and G. Neven, Eds., pp. 159–176.

[11] BERNSTEIN, D. J., AND SCHWABE, P. A word of warning. CHES 2013 Rump Session, August 2013.

[12] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium* (Washington, DC, United States, August 2003), pp. 105–120.

[13] BOBROW, D. G., BURCHFIEL, J. D., MURPHY, D. L., AND TOMLINSON, R. S. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM 5*, 3 (March 1972), 135–143.

[14] BRUMLEY, B. B., AND HAKALA, R. M. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 667–684.

[15] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (November 1997), 412–447.

[16] CAMPAGNA, M., AND SETHI, A. Key recovery method for CRT implementation of RSA. Report 2004/147, IACR Cryptology ePrint Archive, 2004.

[17] CERON, R., FOLCO, R., LEITAO, B., AND TSUBAMOTO, H. *Power Systems Memory Deduplication*. IBM, September 2012.

[18] CERT vulnerability note vu#976534: L3 cpu shared cache architecture is susceptible to a Flush+Reload side-channel attack. `http://www.kb.cert.org/vuls/id/976534`, October 2013.

[19] CHEN, C., WANG, T., KOU, Y., CHEN, X., AND LI, X. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *The Journal of Systems and Software 86*, 1 (2013), 100–107.

[20] CIPRESSO, T., AND STAMP, M. Software reverse engineering. In *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Springer, 2010, ch. 31, pp. 659–696.

[21] CORON, J.-S. Resistence against differential power analysis for elliptic curve cryptosystems. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems* (Worcester, Massachusetts, United States, August 1999), Ç. K. Koç and C. Paar, Eds., pp. 292–302.

[22] CURTSINGER, C., AND BERGER, E. D. STABILIZER: Statistically sound performance evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, United States, March 2013), pp. 219–228.

[23] ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, 4 (July 1985), 469–472.

[24] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. H. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems* (Cape Code, Massachusetts, United States, May 1997), pp. 67–72.

[25] GARNER, H. L. The residue number system. *IRE Transactions on Electronic Computers EC-8*, 2 (June 1959), 140–147.

[26] GINGELL, R. A., LEE, M., DANG, X. T., AND WEEKS, M. S. Shared libraries in SunOS. In *USENIX Conference Proceedings* (Phoenix, Arizona, United States, Summer 1987), pp. 131–145.

[27] GNU Privacy Guard. `http://www.gnupg.org`, 2013.

[28] GORDON, D. M. A survey of fast exponentiation methods. *Journal of Algorithms 27*, 1 (April 1998), 129–146.

[29] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games — bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, United States, may 2011), pp. 490–595.

[30] HENINGER, N., AND SHACHAM, H. Reconstructing RSA private keys from random key bits. In *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO 2009)* (Santa Barbara, California, United States, August 2009), S. Halevi, Ed., pp. 1–17.

[31] HERMANN, M., AND MAY, A. Solving linear equations modulo divisors: On factoring given any bits. In *Advances in Cryptology - ASIACRYPT 2008* (Melbourne, Australia, December 2008), vol. 5350 of *Lecture Notes in Computer Science*, pp. 406–424.

[32] HU, W.-M. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, United States, May 1991), pp. 8–20.

[33] HUFFMAN, C. Memory combining in Windows 8 and Windows Server 2012. `http://blogs.technet.com/b/clinth/archive/2012/11/29/memory-combining-in-windows-8-and-windows-server-2012.aspx`, November 2012.

[34] INTEL CORPORATION. *Intel 64 and IA-32 Architecture Optimization Reference Manual*, April 2012.

[35] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, March 2013.

[36] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the Annual Computer Security Applications Conference* (Miami Beach, Florida, United States, December 2006), pp. 339–348.

[37] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. **kvm**: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, June 2007), vol. one, pp. 225–230.

[38] KOCH, W. GnuPG 1.4.14 released. `http://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000330.html`, July 2013.

[39] MIŁOŚ, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, California, United States, June 2009).

[40] MURPHY, D. L. Storage organization and mamagement in TENEX. In *Proceedings of the Fall Joint Computer Conference, AFIPS'72, Part I* (Anaheim, California, United States, December 1972), pp. 23–32.

[41] OPENSSL. `http://www.openssl.org`.

[42] ORGANICK, E. I. *The Multics System: An Examination of Its Structure*. The MIT Press, 1972.

[43] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. `http://www.cs.tau.ac.il/~tromer/papers/cache.pdf`, November 2005.

[44] OWENS, R., AND WANG, W. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communicatons Conference* (Orlando, Florida, United States, November 2011), S. Zhong, D. Dou, and Y. Wang, Eds., IEEE, pp. 1–8.

[45] PAOLONI, G. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, September 2010.

[46] PERCIVAL, C. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, 2005.

[47] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communication Security* (Chicago, Illinois, United States, November 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., pp. 199–212.

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

[49] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory deduplication as a threat to the guest. In *Proceedings of the 2011 European Workshop on System Security* (Salzburg, Austria, 2011).

[50] The PaX project. `http://pax.grsecurity.net/`.

[51] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks in AES, and countermeasures. *Journal of Cryptology 23*, 2 (January 2010), 37–71.

[52] UHT, A. K., AND SINDAGI, V. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture* (Ann Arbor, Michigan, United States, November 1995), pp. 313–325.

[53] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating fine grained timers in Xen. In *Proceedings of the ACM Workshop on Cloud Computing Security* (Chicago, Illinois, United States, October 2011), C. Cachin and T. Ristenpart, Eds., pp. 41–46.

[54] VMWARE INC. *Understanding Memory Resource Management in VMware ESX Server*. Palo Alto, California, United States, 2009.

[55] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, Massachusetts, United States, December 2002), D. E. Culler and P. Druschel, Eds., pp. 181–194.

[56] WALTER, C. D. Longer keys may facilitate side channel attacks. In *Selected Areas in Cryptography* (2004), M. Matsui and R. J. Zuccherato, Eds., vol. 3006 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 42–57.

[57] WEISS, M., HEINZ, B., AND STUMPF, F. A cache timing attack on AES in virtualization environments. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security* (Bonaire, February 2012), A. D. Keromytis, Ed.

[58] XIAO, J., XU, Z., HUANG, H., AND WANG, H. A covert channel construction in virtualized environments. In *Proceedings of the 19th ACM Conference on Computer and Communication Security* (Raleigh, North Carolina, United States, October 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., pp. 1040–1042.

[59] XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems* (Florence, Italy, October 2003), pp. 260–269.

[60] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the ACM Workshop on Cloud Computing Security* (Chicago, Illinois, United States, October 2011), C. Cachin and T. Ristenpart, Eds., pp. 29–40.

[61] ZHANG, Y., JULES, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communication Security* (Raleigh, North Carolina, United States, October 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., pp. 305–316.