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

Yuval Yarom Katrina Falkner
School of Computer Science
The University of Adelaide
{yval,katrina}@cs.adelaide.edu.au

18 July 2013

Abstract

FLUSH+RELOAD is a cache side-channel attack that monitors access to data in shared pages. In this paper we demonstrate how to use the attack to extract private encryption keys from GnuPG. The high resolution and low noise of the FLUSH+RELOAD attack enables a spy program to recover over 98% of the bits of the private key in a single decryption or signing round. Unlike previous attacks, the attack targets the last level L3 cache. Consequently, the spy program and the victim do not need to share the execution core of the CPU. The attack is not limited to a traditional OS and can be used in a virtualised environment, where it can attack programs executing in a different VM.

1 Introduction

Side channel attacks are attacks that target an implementation of a cryptographic system rather than targetting theoretical weaknesses of the system. Cache side-channel attacks are a form of side channel attacks that rely on the timing difference between accessing cached and non-cached values.

Cache side-channel attacks have been used in the past against RSA [1, 13], Elgamal [21], DSA [2] and AES [9, 16, 20]. With one notable exception all of these attacks rely on monopolising the cache by accessing enough data to force the victim's data out of the cache.

As most memory activity occurs in the L1 cache, and as monopolising L1 is much simpler than monopolising lower cache levels [13], all of these attacks target the L1 cache and must, therefore, execute on the same CPU core as the victim.

Gullasch et al. [9] describe a cache side-channel attack that relies on shared memory pages between the spy program and the victim. This attack, which we call FLUSH+RELOAD, evicts specific memory lines out of the cache without monopolising the cache. FLUSH+RELOAD evicts the selected memory line from all the cache levels, including L3. This feature, which is not exploited by Gullasch et al., allows the attack to be used between processes executing on different execution cores of the same CPU. Furthermore, as virtual machine hypervisors transparently share memory pages between VMs [5, 19], the attack is applicable across the isolation layer between VMs.

In this paper we describe the FLUSH+RELOAD attack and demonstrate its use against the GnuPG [7] implementation of RSA. We use the attack to probe GnuPG at a resolution of 1.5MHz. This high resolution, combined with the low noise of the attack allows us to recover over 98% of the bits of the secret key by capturing a single signing or decryption operation.

Sections 2 and 3 provide background information about the cache architecture and about RSA. We describe the FLUSH+RELOAD attack in section 4 and its application to GnuPG in section 5.

2 Cache Architecture and Side-Channel Attacks

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 Diagram 1, consists of three cache levels: L1, L2 and L3.

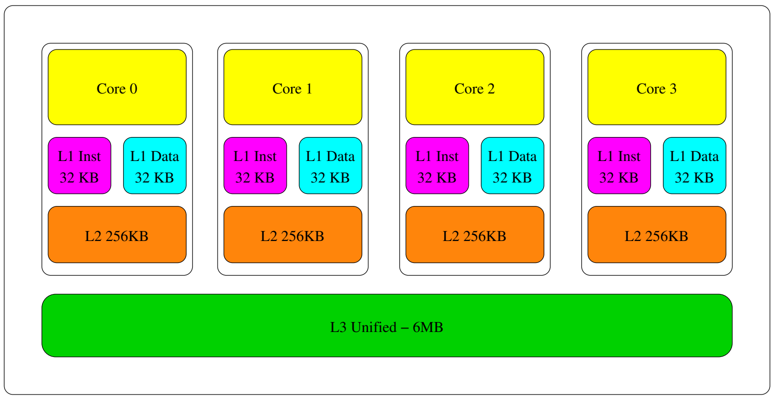

Figure 1: Intel Ivy Bridge Cache Architecture

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.

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.

Retrieving data from memory or from lower cache levels takes longer than retrieving it from higher cache levels. This difference in timing has been exploited for side-channel attacks. Tromer et al. [16] identifies two forms of cache side-channel attacks: the EVICT+TIME attack and the PRIME+PROBE attack.

An EVICT+TIME attack works by setting the cache to a known state, typically by executing a round of computation, evicting selected sets from the cache and measuring the time it takes to perform a round of computation. The time to perform the computation depends on which values are in the cache when the computation starts. Consequently, timing the computation provides information about the memory locations that the computation accesses.

A PRIME+PROBE attack fills the cache with known memory lines, waits some time and times reading these memory lines to checks which of them were evicted from the cache, if any. Again, knowledge of which cache sets have been accessed provides information on memory locations used by the computation and, consequently, on the data it processes.

These techniques tend to be more applicable for implementing side-channels through the L1 cache, rather than the lower cache levels. One reason for that is that caching is based on physical memory addresses, whereas processes use virtual addresses. With L1 caches, a byte offset in a page always map to the same cache set. Hence, for evicting a set or for priming the cache the attacker needs only as many pages as the cache associativity, and any pages would do.

With larger caches, the virtual memory masks the mapping between memory pages and cache sets. Furthermore, this mapping is not necessarily stable and can change during the execution of the program. Techniques for reconstructing the mapping have been suggested [10]. However, these techniques only provide the mapping for the attacker's memory. The victim's mapping is, still, unknown.

The second limitation on these technique is the size of the L3 cache. The effect of cache size on the PRIME+PROBE attack is to increase the prime and probe time significantly, reducing the potential granularity of the attack. For the EVICT+TIME approach, a large cache presents a big search space, increasing the complexity of attacks.

With practical attacks limited to the L1 cache, the techniques described above only work when the victim and the spy programs share the same execution core.

Gullasch et al. [9] describes a cache side-channel attack that we call FLUSH+RELOAD. The attack is similar to PRIME+PROBE in the sense that it monitors the cache changes during the execution of the computation. However, unlike PRIME+PROBE, FLUSH+RELOAD evicts selected memory lines from the cache and measures the time to reload these lines to detect their use by the computation.

We discuss the FLUSH+RELOAD attack in Section 4

3 RSA

RSA [14] 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 encrypted 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 $D(c) = c^d \bmod n$.

A common optimisation for the implementation of the decryption function is to use:

$$\begin{aligned} d_p &= d \bmod (p-1) \\ d_q &= d \bmod (q-1) \\ M_p &= C^{d_p} \bmod p \\ M_q &= C^{d_q} \bmod q \end{aligned}$$

M is then computed from M_p and M_q using the Garner's formula [6]:

$$\begin{aligned} h &= (M_p - M_q)(q^{-1} \bmod p) \bmod p \\ M &= M_q + hq \end{aligned}$$

To compute the encryption and decryption functions, GnuPG uses the square-and-multiply exponentiation algorithm [8]. 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.

```
function exponent($b$, $e$, $m$)
begin
  $x \leftarrow 1$
  for $i \leftarrow |e| - 1$ downto 0 do
    $x \leftarrow x^2$
    $x \leftarrow x \bmod m$
    if ($e_i = 1$) then
      $x \leftarrow xb$
      $x \leftarrow x \bmod m$
    endif
  done
  return $x$
end
```

Figure 2: Exponentiation by Square-and-Multiply

As can be seen from the implementation, an attacker that can trace the execution of the square-and-multiply exponentiation algorithm can recover the exponent. Obviously, if the exponent is d, an attacker recovering the exponent has broken the encryption. However, as GnuPG uses the optimisation described above, the attacker can only hope to extract d_p and d_q. We will now show that extracting any one of d_p and d_q is enough to factor n and, consequently, to break the encryption.

The first point to note is that because $n = pq$, for every X there exists an integer α such that

$$X \bmod n = X \bmod p + \alpha p$$

Consequently,

$$C^{d_p} \bmod n = C^{d_p} \bmod p + \alpha p$$

As $d_p = d \bmod (p-1)$, by applying Fermat's Little Theorem to the above we get

$$C^{d_p} \bmod n = C^{d} \bmod p + \alpha p$$

Now, as there exists a α' such that $C^d \bmod n = C^d \bmod p + \alpha' p$, we get

$$C^{d_p} \bmod n = C^{d} \bmod n - \alpha' p + \alpha p$$

or

$$C^{d_p} \bmod n = M + (\alpha - \alpha')p$$

Thus p can be calculated using

$$p = \gcd((C^{d_p} - M) \bmod n, n)$$

The next section describes the Flush+Reload attack that we use to extract the exponent from GnuPG.

4 The Flush+Reload Attack

The Flush+Reload attack is a variant of the Prime+Probe attack that relies on sharing pages between the spy and the victim programs. With shared pages, the spy program 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.

A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy program, then, waits to allows the victim time to access the memory line before the third phase. In the third phase, the spy program 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. Diagrams 3 (A) and (B) show the timing of the attack phases without and with victim access.

As shown in Diagram 3 (C), the victim access can overlap the reload phase of the spy program. 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 program will miss the access.

A similar scenario is when the reload operation partially overlaps the victim access. In this case, depicted in Diagram 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 spy program 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 shown in Diagram 3 (E), the likelihood of missing the loop is small.

It should be noted that several processor optimisations may result in false positives due to speculative memory accesses issued by the victim's processor [11]. These optimisations include data prefetching to exploit spatial locality and speculative execution [17]. 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 in the variable `time`. (Line 15.)

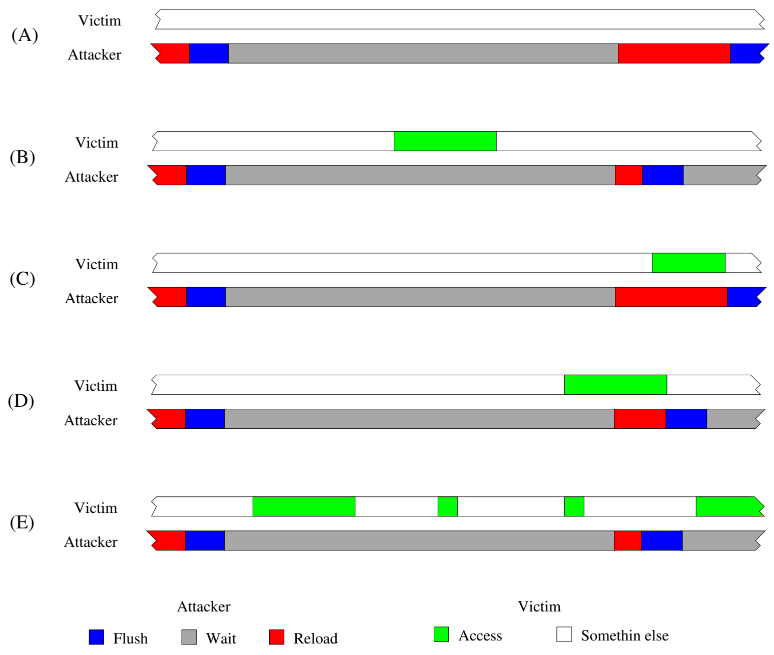

Figure 3: Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses

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, storing 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 Intel architecture may execute instructions in parallel or out of order. Without serialisation, instructions surrounding the measured code segment may be executed within that segment. Intel recommends using the serialising instruction `cpuid` for that purpose [12]. However, in virtualised environments the hypervisor emulates the `cpuid` instruction. This emulation takes significant time (over 1,000 cycles) and this time is not constant, reducing the granularity and the stability of the attack.

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.

Line 18 compares the time difference between the two `rdtsc` instructions against a pre-defined 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 architecture dependent. To find the threshold for the test architecture, 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 are presented in Figure 5.

Virtually all loads from the L1 cache take between 33 and 49 cycles. Of the 100,000 measurements taken we have witnessed 9 outliers, taking over 4,000 cycles. We believe these are the result of OS or

```
int probe(char *adrs) {
  volatile unsigned long time;

  asm __volatile__ (
    "  mfence             \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  lfence             \n"
    "  movl %%eax, %%esi  \n"
    "  movl (%1), %%eax   \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  subl %%esi, %%eax  \n"
    "  clflush 0(%1)      \n"
    : "=a" (time)
    : "c" (adrs)
    :  "%esi", "%edx");
  return time < threshold;
}
```

Figure 4: Code for the Flush+Reload Technique

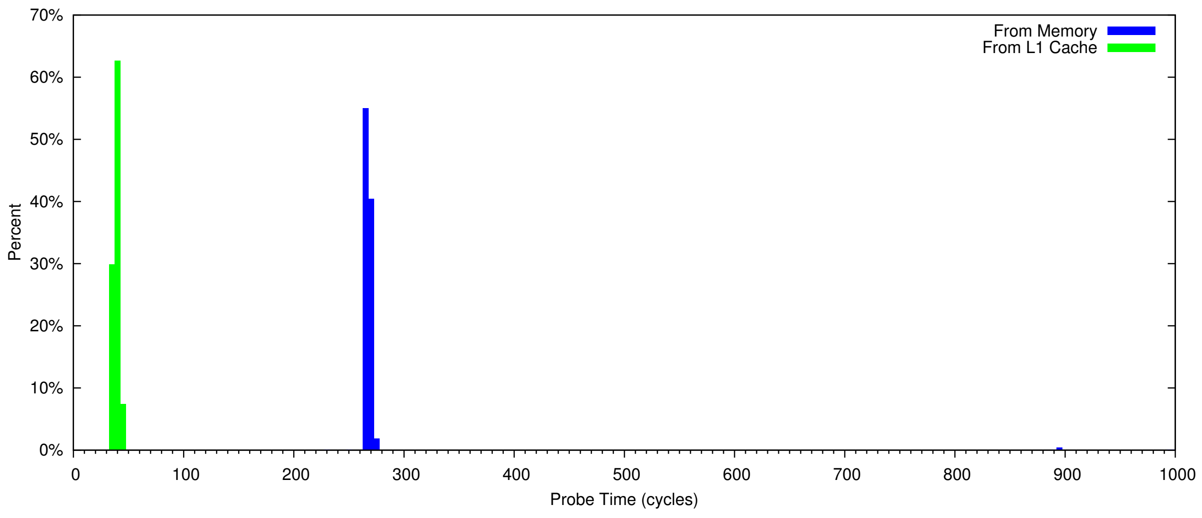

Figure 5: Distribution of Load Times.

hypervisor activity.

Loads from memory show less constant timing. 97% of those take between 200 and 300 cycles. Another 2% are spread between 300 and 400 cycles, but the numbers are too low to be visible in the graph. The last 1% is spread between 850 and 1,200 cycles, with a visible peak at 900 cycles. As in with the L1 measurements, a small number of outliers were seen.

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 [11] 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.

5 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. The attack can work both in a virtualised and in a non-virtualised environment.

We executed the test on an HP Elite 8300, which features an Intel Core i5-3470 processor and 8GB DDR3-1600 memory. The OS is Fedora 18.

The spy program divides time into fixed slots of 2048 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

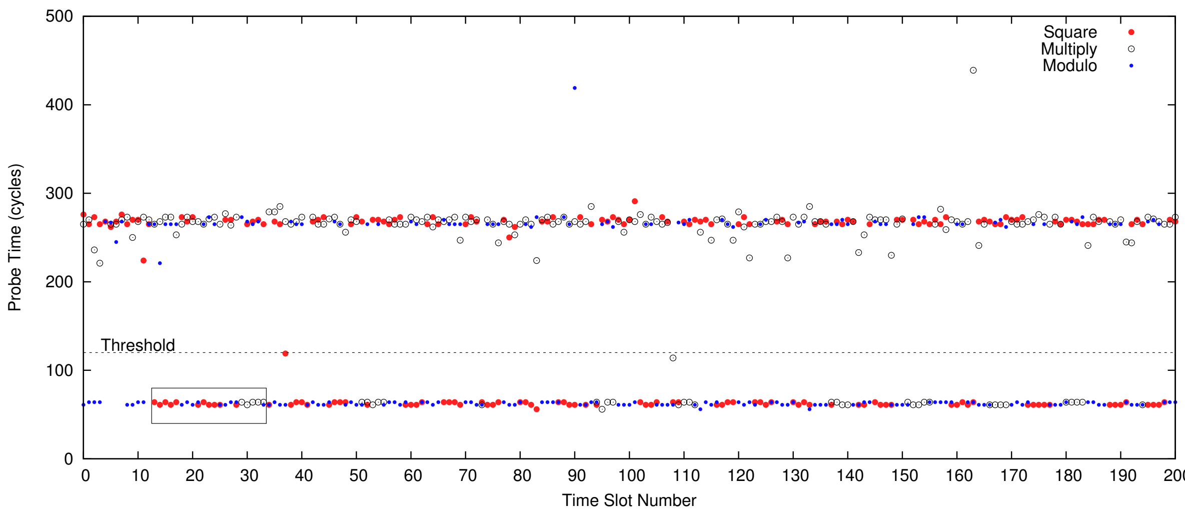

Figure 6: Time measurements of probes

probe capturing the access, we selected memory lines that are executed frequently during the calculation. After probing each memory line, the spy program busy waits to the end of the time slot.

To facilitate locating the memory lines, we used a version of the gpg program that we compiled with debugging symbols. In a real attack settings, the attacker will need to reverse engineer the attacked program to locate the memory lines. Debugging information allows us to find the memory addresses corresponding to each source line without the tedious work of reverse engineering.

Measurement times for the first 200 time slots of the GnuPG signing are displayed in Figure 6. Measurements under the threshold indicate victim access to the respective memory line. The exponentiations for signing takes a total of 22,402 slots or about 15ms. The exponents used have a total of 2046 bits. Of these 9 were missed and 25 were inverted, giving an error rate of 1.6%.

Diagram 7 is an enlarged view of the boxed section in Diagram 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 13 and 17 the victim was calculating a square, Time slots 18–21 are for modulo reduce calculation and 29–33 for multiplication.

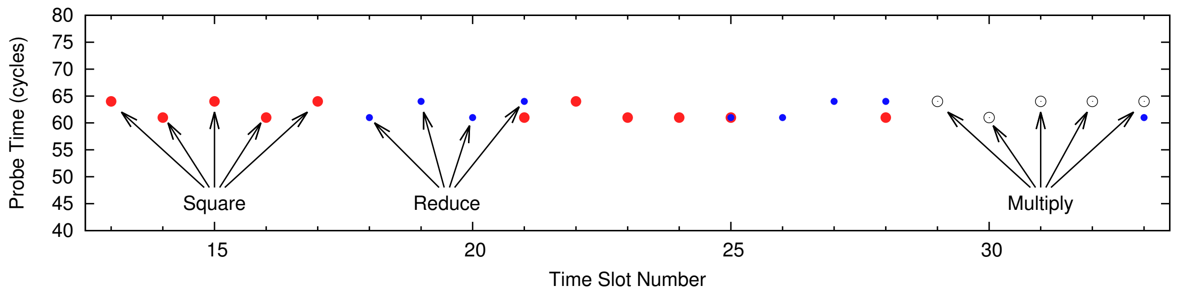

Figure 7: Time measurements of probes

Sequences of Square-Reduce-Multiply-Reduce indicate a 1 bit. Sequences of Square-Reduce which are not followed by Multiply indicate a 0 bit. The bit sequence shown in Diagram 6 is, therefore, 01010000110011110. These bits match the start of d_p. Table 1 shows the time slots corresponding to each bit. The modulo reduce calculation between time slots 0 and 3 is for calculating $d_p = d \bmod p$. The modulo reduce between slots 8 and 11 is for reducing the message hash to modulus p.

Speculative execution is apparent in the graph. For example, in slot 94 the memory lines corresponding to all three operations are loaded from the cache. Obviously, under normal execution of the exponentiation, Multiply and Square cannot occur in the same time slot. The reason all three memory locations are accessed is that the processor tries to predict the future behaviour of the program. In the case of a conditional branch instruction, the processor may start following one of the branches or even

Seq.	Time Slots	Value
1	13–21	0
2	21–37	1
3	38–45	0
4	45–59	1
5	59–65	0
6	66–73	0
7	73–80	0
8	80–87	0
9	87–101	1
10	102–116	1
11	116–123	0
12	123–130	0
13	130–144	1
14	145–159	1
15	159–173	1
16	163–187	1
17	188–194	0

Table 1: Time Slots for Bit Sequence

both before evaluating the condition. Only when the condition is evaluated the processor will commit to the execution.

In time slot 94, the processor speculated that the value of the current bit may be 0. It, therefore, skipped the if body and fetched the memory line of the Square operation. When the value of the bit was evaluated, the processor terminated the speculated branch and proceeded with the Multiply step.

6 Conclusions

This paper describes the Flush+Reload attack and its use for extracting the RSA private key from GnuPG. The attack is able to recover over 98% of the bits of the private key, virtually breaking the key, by capturing a single decryption or signing round. The attack requires the spy program and the victim to share pages, and can work over the isolation layer of virtualised systems.

It is hard to overstate the severity of the weakness in GnuPG. GnuPG is a very popular cryptography package. It is used as the cryptography module of many open-source projects and is used, for example, for email, file and communication encryption. With our attack, any process running on the system can extract private keys. Hence, GnuPG in its current form is not safe for a multi-user system or for any system that may run untrusted code.

Past research has indicated that the square-and-multiply exponentiation may be vulnerable to side channel attacks. The most important lesson that software developers can take from this paper is that sometimes the reuse of an existing technique is all it takes to change a theoretical weakness to an exploitable vulnerability.

It may be possible that other known weaknesses can become exploitable with this attack. Examples that come to mind include identifying the extra reduction in Montgomery Multiplication [3, 15] and the Vaudenay padding oracle attack [4, 18]. Further research is required to determine the extent to which Flush+Reload can be used as part of such attacks and whether existing countermeasures provide sufficient protection against Flush+Reload.

It should be noted that the attack is not limited to cryptography software. Other software may leak sensitive information through the cache. This is particularly worrying in a virtualised environment, where users expect complete isolation between VMs. Memory de-duplication may save resources, but it compromises security. We recommend that memory de-duplication in virtualised systems is disabled.

Acknowledgments

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

References

[1] O. Acıiçmez, “Yet another microarchitectural attack: exploiting I-Cache,” in *Proceedings of the ACM Workshop on Computer Security Architecture*, Fairfax, Virginia, United States, November 2007, pp. 11–18.

[2] O. Acıiçmez, B. B. Brumley, and P. Grabher, “New results on instruction cache attacks,” in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, Santa Barbara, California, United States, August 2010, pp. 110–124.

[3] O. Acıiçmez and W. Schindler, “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, pp. 256–273.

[4] N. J. AlFardan and K. G. Paterson, “Plaintext-recovery attacks against datagram TLS,” in *Proceedings of the 19th Annual Network & Distributed Systems Security Symposium*, San Diego, California, United States, February 2012.

[5] A. Arcangeli, I. Eidus, and C. Wright, “Increasing memory density by using KSM,” in *Proceedings of the Linux Symposium*, Montreal, Quebec, Canada, July 2009, pp. 19–28.

[6] H. L. Garner, “The residue number system,” *IRE Transactions on Electronic Computers*, vol. EC-8, no. 2, pp. 140–147, June 1959.

[7] “GNU Privacy Guard,” http://www.gnupg.org, 2013.

[8] D. M. Gordon, “A survey of fast exponentiation methods,” *Journal of Algorithms*, vol. 27, no. 1, pp. 129–146, April 1998.

[9] D. Gullasch, E. Bangerter, and S. Krenn, “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.

[10] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space ASLR,” in *Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, California, United States, May 2013, pp. 191–205.

[11] *Intel 64 and IA-32 Architecture Optimization Reference Manual*, Intel Corporation, April 2012.

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

[13] C. Percival, “Cache missing for fun and profit,” http://www.daemonology.net/papers/htt.pdf, 2005.

[14] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[15] W. Schindler, “A timing attack against RSA with the Chinese remainder theorem,” in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, no. 109–124, Worcester, Massachusetts, United States, August 2000.

[16] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks in AES, and countermeasures,” *Journal of Cryptology*, vol. 23, no. 2, pp. 37–71, January 2010.

[17] A. K. Uht and V. Sindagi, “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.

[18] S. Vaudenay, “Security flaws induced by CBC padding. applications to SSL, IPSEC, WTLS...,” in *EUROCRYPT 2002, Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, Amsterdam, The Netherlands, April 2002, pp. 534–546.

[19] C. A. Waldspurger, “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, pp. 181–194.

[20] M. Weiß, B. Heinz, and F. Stumpf, “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.

[21] Y. Zhang, A. Jules, M. K. Reiter, and T. Ristenpart, “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, pp. 305–316.