CacheBleed: A Timing Attack on OpenSSL Constant Time RSA

Yuval Yarom[1(✉)], Daniel Genkin[2], and Nadia Heninger[3]

[1] The University of Adelaide and NICTA, Adelaide, Australia
yval@cs.adelaide.edu.au
[2] Technion and Tel Aviv University, Tel Aviv, Israel
danielg3@cs.technion.ac.il
[3] University of Pennsylvania, Philadelphia, USA
nadiah@cis.upenn.edu

Abstract. The scatter-gather technique is a commonly implemented approach to prevent cache-based timing attacks. In this paper we show that scatter-gather is not constant time. We implement a cache timing attack against the scatter-gather implementation used in the modular exponentiation routine in OpenSSL version 1.0.2f. Our attack exploits cache-bank conflicts on the Sandy Bridge microarchitecture. We have tested the attack on an Intel Xeon E5-2430 processor. For 4096-bit RSA our attack can fully recover the private key after observing 16,000 decryptions.

Keywords: Side-channel attacks · Cache attacks · Cryptographic implementations · Constant-time · RSA

1 Introduction

1.1 Overview

Side-channel attacks are a powerful method for breaking theoretically secure cryptographic primitives. Since the first works by Kocher [33], these attacks have been used extensively to break the security of numerous cryptographic implementations. At a high level, it is possible to distinguish between two types of side-channel attacks, based on the methods used by the attacker: hardware-based attacks, which monitor the leakage through measurements (usually using dedicated lab equipment) of physical phenomena such as electromagnetic radiation [43], power consumption [31,32], or acoustic emanation [22], and software-based attacks, which do not require additional equipment but rely instead on the attacker software running on or interacting with the target machine. Examples of the latter include timing attacks which measure timing variations of cryptographic operations [7,16,17] and cache attacks which observe cache access patterns [40,41,49].

Percival [41] published in 2005 a *cache attack*, which targeted the OpenSSL [39] 0.9.7c implementation of RSA. In this attack, the attacker and the victim programs are colocated on the same machine and processor, and thus share the

B. Gierlichs and A.Y. Poschmann (Eds.): CHES 2016, LNCS 9813, pp. 346–367, 2016.
DOI: 10.1007/978-3-662-53140-2_17

same processor cache. The attack exploits the structure of the processor cache by observing minute timing variations due to cache contention. The cache consists of fixed-size *cache lines*. When a program accesses a memory address, the cache-line-sized block of memory that contains this address is stored in the cache and is available for future use. The attack traces the changes that the victim program execution makes in the cache and, from this trace, the attacker is able to recover the private key used for the decryption.

In order to implement the modular exponentiation routine required for performing RSA public and secret key operations, OpenSSL 0.9.7c uses a sliding-window exponentiation algorithm [11]. This algorithm precomputes some values, called *multipliers*, which are used throughout the exponentiation. The access pattern to these precomputed multipliers depends on the exponent, which, in the case of decryption and digital signature operations, should be kept secret. Because each multiplier occupies a different set of cache lines, Percival [41] was able to identify the accessed multipliers and from that recover the private key. To mitigate this attack, Intel implemented a countermeasure that changes the memory layout of the precomputed multipliers. The countermeasure, often called *scatter-gather*, interleaves the multipliers in memory to ensure that the same cache lines are accessed irrespective of the multiplier used [14]. While this countermeasure ensures that the same cache lines are always accessed, the offsets of the accessed addresses within these cache lines depend on the multiplier used and, ultimately, on the private key.

Both Bernstein [7] and Osvik et al. [40] have warned that accesses to different offsets within cache lines may leak information through timing variations due to cache-bank conflicts. To facilitate concurrent access to the cache, the cache is often divided into multiple *cache banks*. Concurrent accesses to different cache banks can always be handled, however each cache bank can only handle a limited number of concurrent requests—often a single request at a time. A *cache-bank conflict* occurs when too many requests are made concurrently to the same cache bank. In the case of a conflict, some of the conflicting requests are delayed. While timing variations due to cache-bank conflicts are documented in the Intel Optimization Manual [28], no attack exploiting these has ever been published. In the absence of a demonstrated risk, Intel continued to contribute code that uses scatter-gather to OpenSSL [23,24] and to recommend the use of the technique for side-channel mitigation [12,13]. Consequently, the technique is in widespread use in the current versions of OpenSSL and its forks, such as LibreSSL [35] and BoringSSL [10]. It is also used in other cryptographic libraries, such as the Mozilla Network Security Services (NSS) [38].

1.2 Our Contribution

In this work we present CacheBleed, the first side-channel attack to systematically exploit cache-bank conflicts. In Sect. 3 we describe how CacheBleed creates contention on a cache bank and measures the timing variations due to conflicts and in Sect. 4 we use CacheBleed in order to attack the scatter-gather implementation of OpenSSL's modular exponentiation routine. After observing 16,000

RSA decryptions or signing operations, we are able to recover 60 % of the secret exponent bits. To find the remaining bits we adapt the Heninger-Shacham algorithm [25] for the information we collect with CacheBleed. In order to achieve full key extraction, our attack requires about two CPU hours. Parallelizing across multiple CPUs, we achieved key extraction in only a few minutes. See Sect. 5 for a more complete discussion.

1.3 Targeted Software and Hardware

Software. In this paper we target the modular exponentiation operation as implemented in OpenSSL version 1.0.2f which was the latest version of OpenSSL prior to our disclosure to OpenSSL. As mentioned above, similar (and thus potentially vulnerable) code can be found in several forks of OpenSSL such as LibreSSL [35] and BoringSSL [10]. Other cryptographic libraries, such as the Mozilla Network Security Services (NSS) [38] use similar techniques and may be vulnerable as well.

Hardware. Our attacks exploit cache-bank conflicts present in Intel Sandy Bridge Processor family. We ran our experiments on an Intel Xeon E5-2430 processor which is a six-core Sandy Bridge machine with a 2.20 GHZ clock. Our target machine is running CentOS 6.7 installed with its default parameters and with huge pages enabled.

Disclosure and Mitigation. We have reported our results to the developers of OpenSSL, LibreSSL, NSS, and BoringSSL. We worked with the OpenSSL developers to evaluate and deploy countermeasures to prevent the attacks described in this paper (CVE-2016-0702). These countermeasures were subsequently incorporated into OpenSSL 1.0.2g and BoringSSL. The LibreSSL development team notified us that they are still working on a patch. The current version (2.4.0) appears to remain vulnerable. For NSS, our attack was documented under Mozilla bug 1252035. The bug documentation indicates that the fix is scheduled to be included in version 3.24.

2 Background

2.1 OpenSSL's RSA Implementation

RSA [44] is a public-key cryptosystem which supports both encryption and digital signatures. To generate an RSA key pair, the user generates two prime numbers p, q and computes $N = pq$. Next, given a public exponent e (OpenSSL uses $e = 65537$), the user computes the secret exponent $d \equiv e^{-1} \bmod \phi(N)$. The public key is the integers e and N and the secret key is d and N. In textbook RSA encryption, a message m is encrypted by computing $m^e \bmod N$ and a ciphertext c is decrypted by computing $c^d \bmod N$.

Algorithm 1. Fixed-window exponentiation

input : window size w, base a, modulus k, n-bit exponent $b = \sum_{i=0}^{\lceil n/w \rceil} 2^{wi} \cdot b_i$
output: $a^b \bmod k$

```
//Precomputation
a_0 ← 1
for j = 1, ..., 2^w − 1 do
  | a_j ← a_{j−1} · a mod k
end
//Exponentiation
r ← 1
for i = ⌈n/w⌉ − 1, ..., 0 do
  | for j = 1, ..., w do
  |   | r ← r^2 mod k
  | end
  | r ← r · a_{b_i} mod k
end
return r
```

RSA-CRT. RSA decryption is often implemented using the Chinese remainder theorem (CRT), which provides a speedup over exponentiation mod n. Instead of computing $c^d \bmod n$ directly, RSA-CRT splits the secret key d into two parts $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, and then computes two parts of the message as $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. The message m can then be recovered from m_p and m_q using Garner's formula [21]:

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

The main operation performed during RSA decryption is the modular exponentiation, that is, calculating $a^b \bmod k$ for some secret exponent b. Several algorithms for modular exponentiation have been suggested. In this work we are interested in the two algorithms that OpenSSL has used.

Fixed-Window Exponentiation. In the *fixed-window exponentiation* algorithm, also known as *m-ary exponentiation*, the n-bit exponent b is represented as an $\lceil n/w \rceil$ digit integer in base 2^w for some chosen *window size* w. That is, b is rewritten as $b = \sum_{i=0}^{\lceil n/w \rceil - 1} 2^{wi} \cdot b_i$ where $0 \leq b_i < 2^w$. The pseudocode in Algorithm 1 demonstrates the fixed-window exponentiation algorithm. In the first step, the algorithm precomputes a set of *multipliers* $a_j = a^j \bmod k$ for $0 \leq j < 2^w$. It then scans the base 2^w representation of b from the most significant digit ($b_{\lceil n/w \rceil - 1}$) to the least significant (b_0). For each digit b_i it squares an intermediate result w times and then multiplies the intermediate result by a_{b_i}. Each of the square or multiply operations is followed by a modular reduction.

Sliding-Window Exponentiation. The *sliding-window algorithm* represents the exponent b as a sequence of digits b_i such that $b = \sum_{i=0}^{n-1} 2^i \cdot b_i$, with b_i

being either 0 or an odd number $0 < b_i < 2^w$. The algorithm first precomputes $a_1, a_3, \ldots a_{2^{w-1}}$ as in the fixed-window case. It then scans the exponent from the most significant to the least significant digit. For each digit, the algorithm squares the intermediate result. For non-zero digit b_i, it also multiplies the intermediate result by a_{b_i}.

The main advantages of the sliding-window algorithm over the fixed-window algorithm are that, for the same window size, sliding window needs to precompute half the number of multipliers, and that fewer multiplications are required during the exponentiation. The sliding-window algorithm, however, leaks the position of the non-zero multipliers to adversaries who can distinguish between squaring and multiplication operations. Furthermore, the number of squaring operations between consecutive multipliers may leak the values of some zero bits. Up to version 0.9.7c, OpenSSL used sliding-window exponentiation. As part of the mitigation of the Percival [41] cache attack, which exploits these leaks, OpenSSL changed their implementation to use the fixed-window exponentiation algorithm.

Since both algorithms precompute a set of multipliers and access them throughout the exponentiation, a side-channel attack that can discover which multiplier is used in the multiplication operations can recover the digits b_i and from them obtain the secret exponent b.

2.2 The Intel Cache Hierarchy

We now turn our attention to the cache hierarchy in modern Intel processors. The cache is a small, fast memory that exploits the temporal and spatial locality of memory accesses to bridge the speed gap between the faster CPU and slower memory. In the processors we are interested in, the cache hierarchy consists of three levels of caching. The top level, known as the *L1 cache*, is the closest to the execution core and is the smallest and the fastest cache. Each successive cache level is larger and slower than the preceding one, with the *last-level cache* (LLC) being the largest and slowest.

Cache Structure. The cache stores fixed-sized chunks of memory called *cache lines*. Each cache line holds 64 bytes of data that come from a 64-byte aligned *block* in memory. The cache is organized as multiple *cache sets*, each consisting of a fixed number of *ways*. A block of memory can be stored in any of the ways of a single cache set. For the higher cache levels, the mapping of memory blocks to cache sets is done by selecting a range of address bits. For the LLC, Intel uses an undisclosed hash function to map memory blocks to cache sets [30,37,50]. The L1 cache is divided into two sub caches: the *L1 data cache* (L1-D) which caches the data the program accesses, and the *L1 instruction cache* (L1-I) which caches the code the program executes. In multi-core processors, each of the cores has a dedicated L1 cache. However, multithreaded cores share the L1 cache between the two threads.

Cache Sizes. In the Intel Sandy Bridge microarchitecture, each of the L1-D and L1-I caches has 64 sets and 8 ways to a total capacity of $64 \cdot 8 \cdot 64 = 32,768$ bytes. The L2 cache has 512 sets and 8 ways, with a size of 256 KiB. The L2 cache is *unified*, storing both data and instructions. Like the L1 cache, each core has a dedicated L2 cache. The L3 cache, or the LLC, is shared by all of the cores of the processor. It has 2,048 sets *per core*, i.e. the LLC of a four core processor has 8,192 cache sets. The number of ways varies between processor models and ranges between 12 and 20. Hence the size of the LLC of a small dual core processor is 3 MiB, whereas the LLC of an 8-cores processor can be in the order of 20 MiB. The Intel Xeon E5-2430 processor we used for our experiments is a 6-core processor with a 20-way LLC of size 15 MiB. More recent microarchitectures support more cores and more ways, yielding significantly larger LLCs.

Cache Lookup Policy. When the processor attempts to access data in memory, it first looks for the data in the L1 cache. In a *cache hit*, the data is found in the cache. Otherwise, in a *cache miss*, the processor searches for the data in the next level of the cache hierarchy. By measuring the time to access data, a process can distinguish cache hits from misses and identify whether the data was cached prior to the access.

2.3 Microarchitectural Side-Channel Attacks

In this section we review related works on microarchitectural side-channel timing attacks. These attacks exploit timing variations that are caused by contention on microarchitectural hardware resources in order to leak information on the usage of these resources, and indirectly on the internal operation of the victim. Acıiçmez and Seifert [5] distinguish between two types of channels: those that rely on a persistent state and those that exploit a transient state. Persistent-state channels exploit the limited storage space within the targeted microarchitectural resource. Transient-state channels, in contrast, exploit the limited bandwidth of the targeted element.

Persistent-State Attacks. The Prime+Probe attack [40,41] is an example of a persistent-state attack. The attack exploits the limited storage space in cache sets to identify the sets used for the victim's data. The attacker preloads data to the cache and allows the victim to execute before measuring the time to access the preloaded data. When the victim accesses its data it is loaded into the cache, replacing some of the attacker's preloaded data. Accessing data that has been replaced will take longer than accessing data still in the cache. Thus the attacker can identify the cache sets that the victim has accessed. Persistent-state channels have targeted the L1 data cache [7,15,40,41], the L1 instruction cache [1,4,51], the branch prediction buffer [2,3], the last-level cache [27,29,36,46,49], and DRAM open rows [42]. The Prime+Probe attack was used to recover the accessed multipliers in the sliding-window exponentiation of OpenSSL 0.9.7c [41] and of GnuPG 1.4.18 [27,36].

Transient-State Attacks. Transient-state channels have been investigated mostly within the context of covert channels, where a *Trojan* process tries to covertly exfiltrate information. The idea dates back to Lampson [34] who suggests that processes can leak information by modifying their CPU usage. Covert channels were also observed with shared bus contention [26,48], Acıiçmez and Seifert [5] are the first to publish a side-channel attack based on a transient state. The attack monitors the usage of the multiplication functional unit in a hyperthreaded processor. Monitoring the unit allows an attacker to distinguish between the square and the multiply phases of modular exponentiation. The attack was tested on a victim running fixed-window exponentiation, so no secret information was obtained.

Another transient-state channel uses bus contention to leak side-channel information [47]. By monitoring the capacity of the memory bus allocated to the attacker, the attacker is able to distinguish the square and the multiply steps. Because the attack of [47] was only demonstrated in a simulator, the question of whether actual hardware leaks such high-resolution information is still open.

2.4 Scatter-Gather Implementation

One of the countermeasures Intel recommends against side-channel attacks is to avoid secret-dependent memory access *at coarser than cache line granularity* [12, 13]. This approach is manifested in the patch Intel contributed to the OpenSSL project to mitigate the Percival [41] attack. The patch[1] changes the layout of the multipliers in memory. Instead of storing the data of each of the multipliers in consecutive bytes in memory, the new layout scatters each multiplier across multiple cache lines [14]. Before use, the fragments of the required multiplier are gathered to a single buffer which is used for the multiplication. Figure 1 contrasts the conventional memory layout of the multipliers with the layout used in the scatter-gather approach. This scatter-gather design ensures that the order of accessing cache lines when performing a multiplication is independent of the multiplier used.

Because Intel cache lines are 64 bytes long, the maximum number of multipliers that can be used with scatter-gather is 64. For large exponents, increasing the number of multipliers reduces the number of multiply operations performed during the exponentiations. Gopal et al. [23] suggest dividing the multipliers into 16-bit fragments rather than into bytes. This improves performance by allowing loads of two bytes in a single memory access, at the cost of reducing the maximum number of multipliers to 32. Gueron [24] recommends 32-bit fragments, thus reducing the number of multipliers to 16. He shows that the combined savings from the reduced number of memory accesses and the smaller cache footprint of the multipliers outweighs the performance loss due to the added multiplications required with less multipliers.

[1] https://github.com/openssl/openssl/commit/46a643763de6d8e39ecf6f76fa79b4d04885aa59.

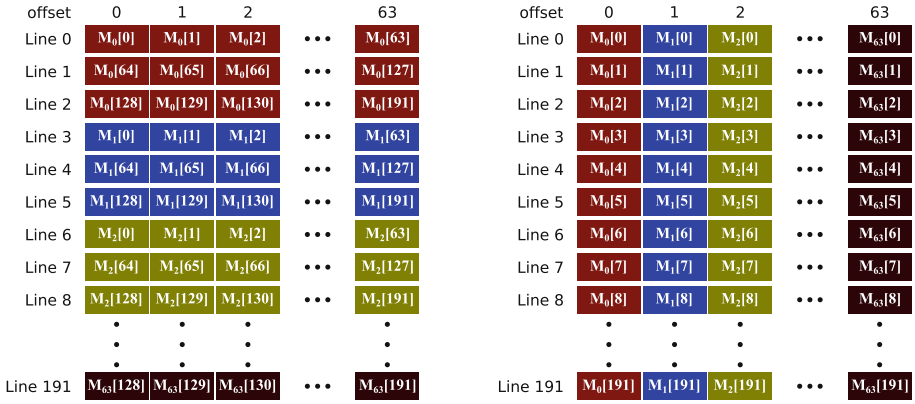

Fig. 1. Conventional (left) vs. scatter-gather (right) memory layout.

The OpenSSL Scatter-Gather Implementation. The implementation of exponentiation in the current version of OpenSSL (1.0.2f) deviates slightly from the layout described above. For 2048-bit and 4096-bit key sizes the implementation uses a fixed-window algorithm with a window size of 5, requiring 32 multipliers. Instead of scattering the multipliers in each cache line, the multipliers are divided into 64-bit fragments, scattered across groups of four consecutive cache lines. (See Fig. 2.) That is, the table that stores the multipliers is divided into groups of four consecutive cache lines. Each group of four consecutive cache lines stores one 64-bit fragment of each multiplier. To avoid leaking information on the particular multiplier used in each multiplication, the gather process accesses all of the cache lines and uses a bit mask pattern to select the ones that contain fragments of the required multiplier. Furthermore, to avoid copying the multiplier data, the implementation combines the gather operation with the multiplication. This spreads the access to the scattered multiplier across the multiplication.

Key-Dependent Memory Accesses. Because the fragments of each multiplier are stored in a fixed offset within the cache lines, all of the scatter-gather implementations described above have memory accesses that depend on the multiplier used and thus on the secret key. For a pure scatter-gather approach, the multiplier is encoded in the low bits of the addresses accessed during the gather operation. For the case of OpenSSL's implementation, only the three least significant bits of the multiplier number are encoded in the address while the other two bits are used as the index of the cache line within the group of four cache lines that contains the fragment.

We note that because these secret-dependent accesses are at a finer than cache line granularity, the scatter-gather approach has been considered secure against side-channel attacks [24].

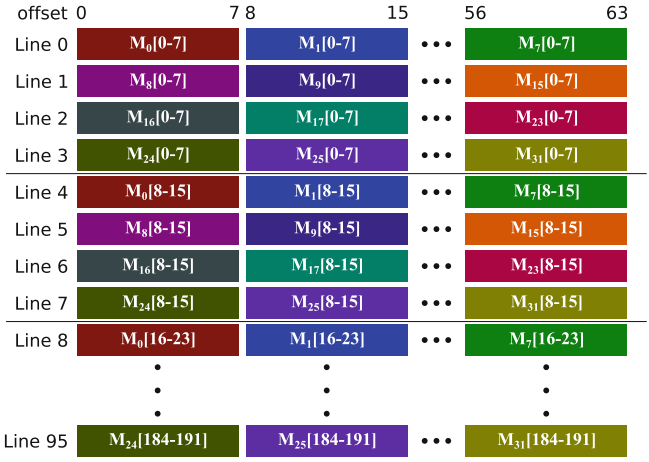

Fig. 2. The memory layout of the multipliers table in OpenSSL

2.5 Intel L1 Cache Banks

With the introduction of superscalar computing in Intel processors, cache bandwidth became a bottleneck for processor performance. To alleviate the issue, Intel introduced a cache design consisting of multiple *banks* [6]. Each of the banks serves part of the cache line specified by the offset in the cache line. The banks can operate independently and serve requests concurrently. However, each bank can only serve one request at a time. When multiple accesses to the same bank are made concurrently, only one access is served, while the rest are delayed until the bank can handle them.

Fog [18] notes that cache-bank conflicts prevent instructions from executing simmultaneously on Pentium processors. Delays due to cache-bank conflicts are also documented for other processor versions [19,20,28].

Both Bernstein [7] and Osvik et al. [40] mention that cache-bank conflicts cause timing variations and warn that these may result in a timing channel which may leak information about low address bits. Tromer et al. [45] note that while scatter-gather has no secret-dependent accesses to cache lines, it does have secret-dependent access to cache banks. Bernstein and Schwabe [8] demonstrate timing variations due to conflicts between read and write instructions on addresses within the same cache bank and suggest these may affect cryptographic software. However, although the risk of side-channel attacks based on cache-bank conflicts has been identified long ago, no attacks exploiting them have ever been published.

3 The CacheBleed Attack

We now proceed to describe CacheBleed, the first side-channel attack to systematically exploit cache-bank conflicts. The attack identifies the times at which a

victim accesses data in a monitored cache bank by measuring the delays caused by contention on the cache bank.

In our attack scenario, we assume that the victim and the attacker run concurrently on two hyperthreads of the same processor core. Thus, the victim and the attacker share the L1 data cache. Recall that the Sandy Bridge L1 data cache is divided into multiple banks and that the banks cannot handle concurrent load accesses. The attacker issues a large number of load accesses to a cache bank and measures the time to fulfill these accesses. If during the attack the victim also accesses the same cache bank, the victim accesses will contend with the attacker for cache bank access, causing delays in the attack. Hence, when the victim accesses the monitored cache bank the attack will take longer than when the victim accesses other cache banks.

To implement CacheBleed we use the code in Listing 1. The bulk of the code (Lines 4–259) consists of 256 addl instructions that read data from addresses that are all in the same cache bank. (The cache bank is selected by the low bits of the memory address in register r9.) We use four different destination registers to avoid contention on the registers themselves. Before starting the accesses, the code takes the value of the current cycle counter (Line 1) and stores it in register r10 (Line 2). After performing 256 accesses, the previously stored value of the cycle counter is subtracted from the current value, resulting in the number of cycles that passed during the attack.

```
  1  rdtscp
  2  movq    %rax, %r10
  3
  4  addl    0x000(%r9), %eax
  5  addl    0x040(%r9), %ecx
  6  addl    0x080(%r9), %edx
  7  addl    0x0c0(%r9), %edi
  8  addl    0x100(%r9), %eax
  9  addl    0x140(%r9), %ecx
 10  addl    0x180(%r9), %edx
 11  addl    0x1c0(%r9), %edi
     .
     .
     .
256  addl    0xf00(%r9), %eax
257  addl    0xf40(%r9), %ecx
258  addl    0xf80(%r9), %edx
259  addl    0xfc0(%r9), %edi
260
261  rdtscp
262  subq    %r10, %rax
```

Listing 1. Cache-Bank Collision Attack Code

We run the attack code on an Intel Xeon E5-2430 processor—a six-core Sandy Bridge processor, with a clock rate of 2.20 GHz. Figure 3 shows the histogram of the running times of the attack code under several scenarios.[2]

Scenario 1: Idle. In the first scenario, *idle hyperthread*, the attacker is the only program executing on the core. That is, one of the two hyperthreads executes the attack code while the other hyperthread is idle. As we can see, the attack takes around 230 cycles, clearly showing that the Intel processor is superscalar and that the cache can handle more than one access in a CPU cycle.

Scenario 2: Pure Compute. The second scenario has a victim running a computation on the registers, without any memory access. As we can see, access in this scenario is slower than when there is no victim. Because the victim does not perform memory accesses, cache-bank conflicts cannot explain this slowdown. Hyperthreads, however, share most of the resources of the core, including the execution units, read and write buffers and the register allocation and renaming resources [20]. Contention on any of these resources can explain the slowdown we see when running a pure-compute victim.

Scenario 3: Pure Memory. At the other extreme is the *pure memory* victim, which continuously accesses the cache bank that the attacker monitors. As we can see, the attack code takes almost twice as long to run in this scenario. The distribution of attack times is completely distinct from any of the other scenarios. Hence identifying the victim in this scenario is trivial. This scenario is, however, not realistic—programs usually perform some calculation.

Scenarios 4 and 5: Mixed Load. The last two scenarios aim to measure a slightly more realistic scenario. In this case, one in four victim operations is a memory access, where all of these memory accesses are to the same cache bank. In this scenario we measure both the case that the victim accesses the monitored cache line (*mixed-load*) and when there is no cache-bank contention between the victim and the attacker (*mixed-load–NC*). We see that the two scenarios are distinguishable, but there is some overlap between the two distributions. Consequently, a single measurement may be insufficient to distinguish between the two scenarios.

In practice, even this mixed-load scenario is not particularly realistic. Typical programs will access memory in multiple cache banks. Hence the differences between measurement distributions may be much smaller than those presented in Fig. 3. In the next section we show how we overcome this limitation and correctly identify a small bias in the cache-bank access patterns of the victim.

[2] For clarity, the presented histograms show the envelope of the measured data.

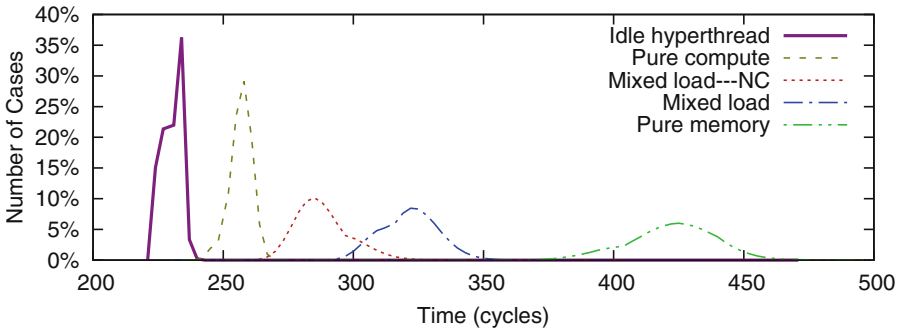

Fig. 3. Distribution of time to read 256 entries from a cache bank.

4 Attacking the OpenSSL Modular Exponentiation Implementation

To demonstrate the technique in a real scenario, we use CacheBleed to attack the implementation of the RSA decryption in the current version of OpenSSL (version 1.0.2f). This implementation uses a fixed-window exponentiation with $w = 5$. As discussed in Sect. 2.4 OpenSSL uses a combination of the scatter-gather technique with masking for side-channel attack protection. Recall that the multipliers are divided into 64-bit fragments. These fragments are scattered into 8 *bins* along the cache lines such that the three least significant bits of the multiplier select the bin. The fragments of a multiplier are stored in groups of four consecutive cache lines. The two most significant bits of the multiplier select the cache line out of the four in which the fragments of the multiplier are stored. See Fig. 2. The multiplication code selects the bin to read using the least significant bits of the multiplier. It then reads a fragment from the selected bin in each of the four cache lines and uses masking to select the fragment of the required multiplier. Because the multiplication code needs to access the multiplier throughout the multiplication, the cache banks of the bin containing the multiplier are accessed more often than other cache banks. We use CacheBleed to identify the bin and, consequently, to find the three least significant bits of the multiplier.

Identifying Exponentiations. We begin by demonstrating that it is possible to identify the exponentiation operations using cache-bank conflicts. Indeed, using the code in Listing 1, we create a sequence of measurements of cache-bank conflicts. As mentioned in Sect. 3, the difference between the distributions of measurements in similar scenarios may be very small. Consequently, a single measurement is unlikely to be sufficient for identifying the bin used in each multiplication. To distinguish the distributions, we create multiple sequences and average the measurements at each trace point to get a trace of the average measurement time. Figure 4 shows the traces of measurements of two bins, each averaged over 1,000 decryptions using a 4096-bit key.

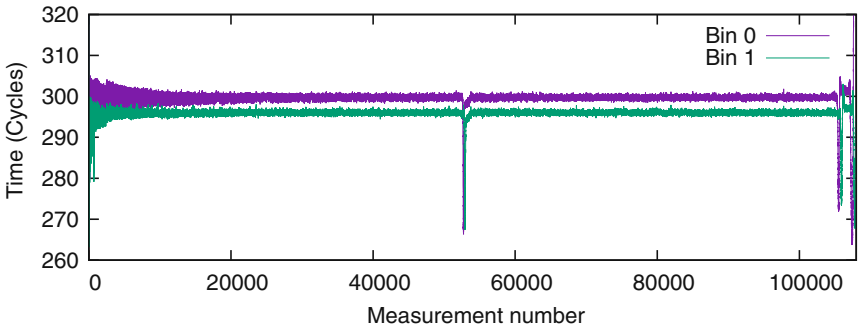

Fig. 4. Measurement trace of OpenSSL RSA decryption

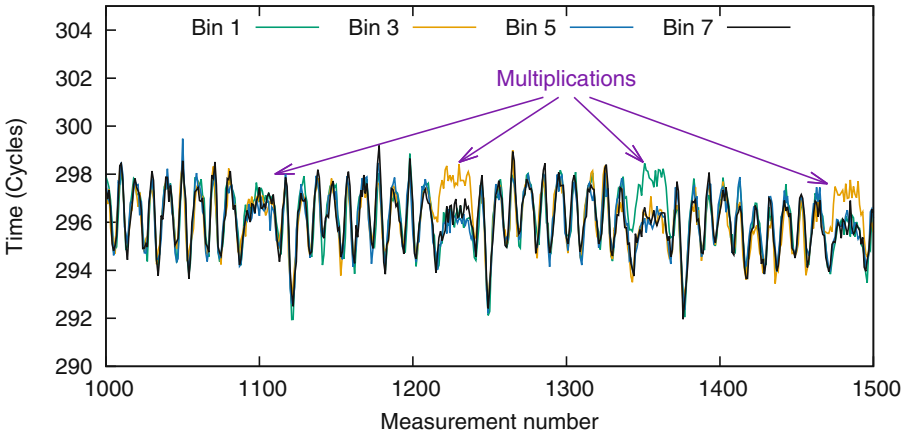

Fig. 5. Measurement trace of OpenSSL RSA decryption—detailed view (Color figure online)

The figure clearly shows the two exponentiations executed as part of the RSA-CRT calculation. Another interesting feature is that the measurements for the two bins differ by about 4 cycles. The difference is the result of the OpenSSL modular reduction algorithm, which accesses even bins more often than odd bins. Consequently, there is more contention on even bins, and measurements on even bins take slightly longer than those on odd bins.

Identifying Multiplication Operations. Next, we show that is also possible to identify the individual multiplication operations performed during the modular exponentiation operation. Indeed, Fig. 5 shows a small section of the traces of the odd bins. In these traces, we can clearly see the multiplication operations (marked with arrows) as well as the spikes for each of the squaring and modular reduction operations. Recall that the OpenSSL exponentiation repeatedly calculate sequences of five modular squaring and reduction operations followed by a modular multiplication.

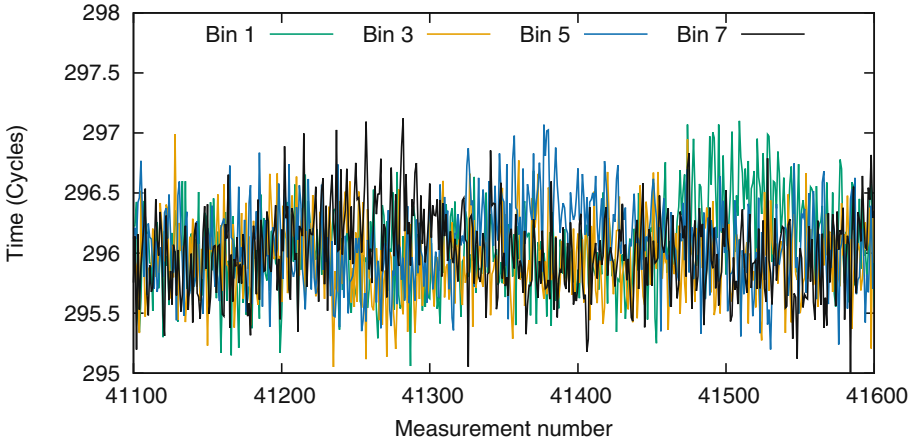

Fig. 6. CacheBleed average trace towards the end of the exponentiation

Identifying Multiplier Values. Note that in the second and fourth multiplications, the measurements in the trace of bin 3 (yellow) take slightly longer than the measurements of the other bins. This indicates that the three least significant digits of the multiplier used in these multiplications are 011. Similarly, the spike in the green trace observed during the third multiplication indicates that the three least significant bits of the multiplier used are 001. This corresponds to the ground truth where the multipliers used in the traced sections are 2, 11, 1, 11.

As we can see, we can extract the multipliers from the trace. However, there are some practical challenges that complicate both the generation of the traces and their analysis. We now discuss these issues.

Aligning CacheBleed Measurement Sequences for Averaging. Recall that the traces shown in Fig. 5 are generated by averaging the sequences of CacheBleed measurements over 1,000 decryptions. When averaging, we need to ensure that the sequences align with each other. That is, we must ensure that each measurement is taken in the same relative time in each multiplication.

To ensure that the sequences are aligned, we use the Flush+Reload attack [49] to find the start of the exponentiation. Once found, we start collecting enough CacheBleed measurements to cover the whole exponentiation. Flush+Reload has a resolution of about 500 cycles, ensuring that the sequences start within 500 cycles, or up to two measurements, of each other.

Relative Clock Drift. Aligning the CacheBleed sequences at the start of the exponentiation does not result in a clean signal. This is because both the victim and the attacker are user processes, and they may be interrupted by the operating system. The most common interruption is due to timer interrupts, which

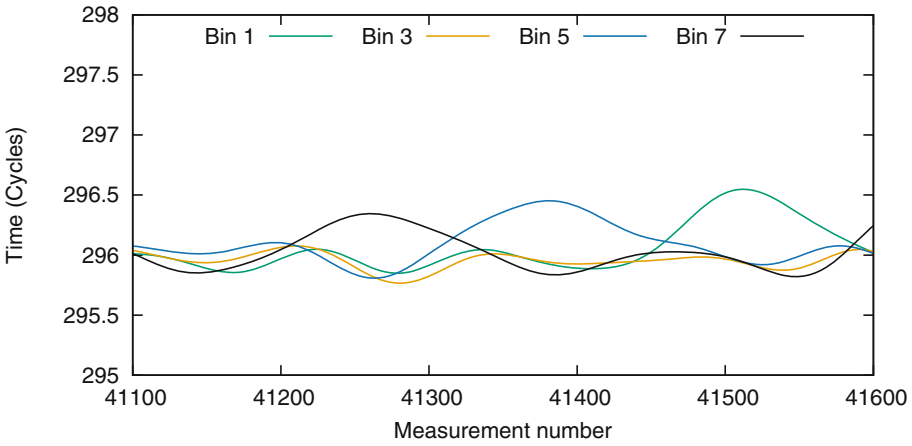

Fig. 7. Measurement trace after a lowpass filter

on Linux-based operating systems happen every millisecond. Since each modular exponentiation in the calculation of a 4096-bit RSA-CRT decryption takes 5 ms, we experience 5 to 6 timer interrupts during the exponentiation. Timer interrupts can be easily identified because serving them takes over 5,000 cycles, whereas non-interrupted measurements take around 300 cycles. Consequently, if a measurement takes more than 1,000 cycles, we assume that it was interrupted and therefore discard it.

The attacker, however, does not have exact information on the interrupts that affect the victim, resulting in clock drift between the attacker and the victim. As we progress through the exponentiation, the signal we capture becomes more noisy. Figure 6 shows the signal towards the end of the exponentiation. As we can see, the multiplications are barely visible.

To reduce noise, we pass the signal through a low-pass filter, which removes high frequencies from the signal and highlights the behavior at the resolution of one multiplication. Figure 7 shows the result of passing the above trace through the filter. It is possible to clearly identify three multiplications, using bins 7, 5 and 1.

Aligning Traces of Multiple Bins. As discussed above, measurements in even bins are slower on average than measurements in odd bins. This creates two problems. The first is that we need to normalize the traces before comparing them to find the multiplier. The second problem is that we use the measurements as a virtual clock. Consequently, when we measure over a fixed period of time, traces of even bins will be shorter, i.e., have fewer measurements, than traces of odd bins. This create a clock shift between traces belonging to even bins and traces belonging to odd bins, which increases as the exponentiation progresses. In order to normalize the trace length, we remove element 0 of the

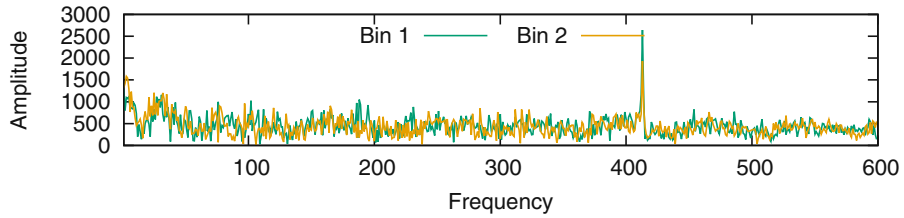

Fig. 8. The frequency spectrum of a trace

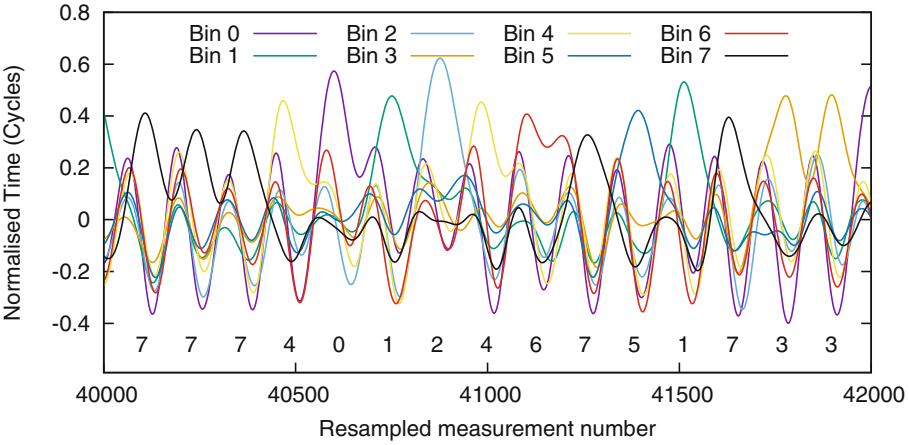

Fig. 9. Normalized resampled traces

frequency domain. This effectively subtracts the trace's average from each trace measurement, thereby making all the traces be at the same length.

We then find the frequency of multiplications in the trace by looking at the frequency domain of the trace. Figure 8 shows the frequency spectrum of two of the traces. For a 4096-bit key, OpenSSL performs two exponentiations with 2048-bit exponents. With a window size of 5, there are $2048/5 \approx 410$ multiplications. As we can see, there is a spike around the frequency 410 matching the number of multiplications. Using the frequency extracted from the trace, rather than the expected number of multiplications, allows us to better adjust to the effects of noise at the start and end of the exponentiation which might otherwise result in a loss of some multiplications.

Partial Key Extraction. We use CacheBleed to collect 16 traces, one for each of the 8 bins in each of the two exponentiations. Each trace is the average of 1,000 sequences of measurements, totalling 16,000 decryption operations. Figure 9 shows a sample of the analyzed traces, i.e. after averaging, passing through a low-pass filter, normalizing the signal and resampling. As we can see, the used bins are clearly visible in the figure.

We manage to recover the three least significant bits of almost all of the multipliers. Due to noise at the start and the end of the exponentiations, we miss one or two of the leading and trailing multiplications of each exponentiation. Next, in Sect. 5, we show that the information we obtain about the three least significant bits of almost all of the multipliers is enough for key extraction.

5 Recovering the RSA Private Key

Successfully carrying out the attack in the previous sections for a 4096-bit modulus allowed us to learn the three least significant bits of every window of five bits for the Chinese remainder theorem coefficients $d_p = d \bmod p - 1$ and $d_q = d \bmod q - 1$. In this section, we describe how to use this knowledge to recover the full private RSA key. We use the techniques of Heninger and Shacham [25] and İnci et al. [27].

Solving for the Modular Multipliers. We have partial knowledge of the bits of d_p and d_q, where each satisfies the relation $ed_p = 1 + k_p(p-1)$ and $ed_q = 1 + k_q(q-1)$ for positive integers $k_p, k_q < e$. In the common case of $e = 65537$, this leaves us with at most 2^{32} possibilities for pairs of k_p, k_q to test. Following [27], the k_p and k_q are related, so we only need to search 65,537 possible values of k_p.

We start by rearranging the relations on d_p and d_q to obtain $ed_p - 1 - k_p = k_p p$ and $ed_q - 1 - k_q = k_q q$. Multiplying these together, we obtain the relation

$$(ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N. \tag{1}$$

Reducing modulo e yields $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$.

Thus, given a value for k_p we can solve for the unique value of $k_q \bmod e$. We do not have enough information about d_p and d_q to deduce further information, so we must test all e values of k_p.

Branch and Prune Algorithm. For each candidate k_p and k_q, we will use Eq. 1 to iteratively solve for d_p and d_q starting from the least or most significant bits, branching to generate multiple potential solutions when bits are unknown and pruning potential solutions when known bits contradict a given solution. In contrast to [25], the bits we know are not randomly distributed. Instead, they are synchronized to the three least significant bits of every five, with one or two full windows of five missing at the least and most significant positions of each exponent. This makes our analysis much simpler: when a bit of d_p and d_q is unknown at a location i, we branch to generate two new solutions. When a bit of d_p and d_q is known at a particular location, using the same heuristic assumption as in [25], an incorrect solution will fail to match the known bit of d_p and d_q with probability 0.5. When k_p and k_q are correct, we expect our algorithm to generate four new solutions for every pair of unknown bits, and prune these to a single correct solution at every string of three known bits. When k_p and k_q are incorrect, we expect no solutions to remain after a few steps.

Empirical Results. We tested key recovery on the output of our attack run on a 4096-bit key, which correctly recovered the three least significant bits of every window of five, but missed the two least significant windows and one most significant window for both d_p and d_q. We implemented this algorithm in Sage and ran it on a Cisco UCS Server with two 2.30 GHz Intel E5-2699 processors and 128 GiB of RAM. For the correct values of k_p and k_q, our branch-and-prune implementation recovered the full key in 1 second on a single core after examining 6,093 candidate partial solutions, and took about 160 ms to eliminate an incorrect candidate pair of k_p and k_q after examining an average of 1,500 candidate partial solutions. A full search of all 65,537 candidate pairs of k_p and k_q parallelized across 36 hyperthreaded cores took 3.5 min. We assumed the positions of the missing windows at the most and least significant bits were known. If the relative positions are unknown, searching over more possible offsets would increase the total search time by a factor of 9.

6 Mitigation

Countermeasures for the CacheBleed attack can operate at the hardware, the system or the software level. Hardware-based mitigations include increasing the bandwidth of the cache banks. Our attack does not work on Haswell processors, which do not seem to suffer from cache-bank conflicts [20,28]. But, as Haswell does show timing variations that depend on low address bits [20], it may be vulnerable to similar attacks. Furthermore, this solution does not apply to the Sandy Bridge processors currently in the market.

Disabling Hyperthreading. The simplest countermeasure at the system level is to disable hyperthreading. Disabling hyperthreading, or only allowing hyperthreading between processes within the same protection domain, prevents any concurrent access to the cache banks and eliminates any conflicts. Unlike attacks on persistent state, which may be applicable when a core is time-shared, the transient state that CacheBleed exploits is not preserved during a context switch. Hence the core can be time-shared between non-trusting processes. The limited security of hyperthreading has already been identified [5]. We recommend that hyperthreading be disabled even on processors that are not vulnerable to CacheBleed for security-critical scenarios where untrusted users share processors.

Constant-Time Implementations. At the software level, the best countermeasure is to use a *constant-time* implementation, i.e. one that does not have secret-dependent branches or memory accesses. A common technique for implementing constant-time table lookup is to use a combination of arithmetic and bitwise operations to generate a mask that depends on the secret value. The whole table is then accessed and the mask is used to select the required table entry. Mozilla's fix for CacheBleed uses this approach.

Modifying Memory Accesses. Rather than using to a constant-time implementation, OpenSSL mitigates CacheBleed through a combination of two changes. The first change is to use 128-bit memory accesses, effectively halving the number of bins used. The second change is to modify the memory access pattern during the gathering process so that the software accesses a different offset in each of the four cache lines.

Combining the four different offsets with the 128-bit accesses means that when gathering a multiplier fragment, OpenSSL accesses all 16 of the cache banks. The order of accessing the cache banks depends on the value of the multiplier, so the design leaks secret key information to adversaries that can recover the order of the accesses. We note, however, that our attack does not have the resolution required to determine the order of successive memory accesses and that we are not currently aware of any technique for exploiting this leak.

Furthermore, using 128-bit memory accesses means that the potential leakage created by the order of accessing the cache banks is only two bits for each multiplier, or 40 % of the bits of the exponents for the 5-bit windows used by OpenSSL for both 2048 and 4096-bit exponents. Our key recovery technique will produce exponentially many solutions in this case: heuristically, we expect it to branch to produce two solutions for each multiplier. In this case, the attacker could use the branch-and-prune method to produce exponentially many candidates up to half the length of each Chinese remainder theorem exponent d_p or d_q, and then use the method of Blömer and May [9] to recover the remaining half in polynomial time. Thus even if an adversary is able to exploit the leak, full key recovery may only be feasible for very small keys without further algorithmic improvements.

While we are not aware of a practical exploit of the leak in the OpenSSL code, we believe that leaving a known timing channel is an undue risk. We have conveyed information about the leak and our concerns to the OpenSSL development team.

7 Conclusions

In this work, we presented CacheBleed, the first timing attack to recover low address bits from secret-dependent memory accesses. We demonstrate that the attack is effective against state-of-the-art cryptographic software, widely thought to be immune to timing attacks.

The timing variations that underlie this attack and the risk associated with them have been known for over a decade. Osvik et al. [40] warn that "*Cache bank collisions (e.g., in Athlon 64 processors) likewise cause timing to be affected by low address bits.*" Bernstein [7] mentions that " *For example, the Pentium 1 has similar cache-bank conflicts.*" A specific warning about the cache-bank conflicts and the scatter-gather technique appears in Footnote 38 of Tromer et al. [45].

Our research illustrates the risk to users when cryptographic software developers dismiss a widely hypothesized potential attack merely because no proof-of-concept has yet been demonstrated. This is the prevailing approach for security

vulnerabilities, but we believe that for cryptographic vulnerabilities, this approach is risky, and developers should be proactive in closing potential vulnerabilities even in the absence of a fully practical attack. To that end we observe that OpenSSL's decision to use an ad-hoc mitigation techniques, instead of deploying a constant-time implementation, continues to follow such a risky approach.

Acknowledgements. We would like to thank Daniel J. Bernstein for suggesting the name CacheBleed and for helpful comments.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This material is based upon work supported by the U.S. National Science Foundation under Grants No. CNS-1408734, CNS-1505799, and CNS-1513671, a gift from Cisco, the Blavatnik Interdisciplinary Cyber Research Center, the Check Point Institute for Information Security, a Google Faculty Research Award, the Israeli Centers of Research Excellence I-CORE program (center 4/11), the Leona M. & Harry B. Helmsley Charitable Trust, and by NATO's Public Diplomacy Division in the Framework of "Science for Peace".

References

1. Acıiçmez, O.: Yet another microarchitectural attack: exploiting I-cache. In: CSAW, Fairfax, VA, US (2007)
2. Acıiçmez, O., Gueron, S., Seifert, J.-P.: New branch prediction vulnerabilities in openSSL and necessary software countermeasures. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 185–203. Springer, Heidelberg (2007)
3. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 225–242. Springer, Heidelberg (2006)
4. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010)
5. Acıiçmez, O., Seifert, J.-P.: Cheap hardware parallelism implies cheap security. In: 4th International Workshop on Fault Diagnosis and Tolerance in Cryptography, Vienna, AT, pp. 80–91 (2007)
6. Alpert, D.B., Choudhury, M.R., Mills, J.D.: Interleaved cache for multiple accesses per clock cycle in a microprocessor. US Patent 5559986, September 1996
7. Bernstein, D.J.: Cache-timing attacks on AES (2005). Preprint http://cr.yp.to/papers.html#cachetiming
8. Bernstein, D.J., Schwabe, P.: A word of warning. In: CHES 2013 Rump Session, August 2013
9. Blömer, J., May, A.: New partial key exposure attacks on RSA. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 27–43. Springer, Heidelberg (2003)
10. BoringSSL. https://boringssl.googlesource.com/boringssl/
11. Bos, J.N.E., Coster, M.J.: Addition chain heuristics. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 400–407. Springer, Heidelberg (1990)
12. Brickell, E.: Technologies to improve platform security. In: CHES 2011 Invited Talk, September 2011. http://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf

13. Brickell, E.: The impact of cryptography on platform security. In: CT-RSA 2012 Invited Talk, February 2012. http://www.rsaconference.com/writable/presentations/file_upload/cryp-106.pdf
14. Brickell, E., Graunke, G., Seifert, J.-P.: Mitigating cache/timing based side-channels in AES and RSA software implementations. In: RSA Conference 2006 Session DEV-203, February 2006
15. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 667–684. Springer, Heidelberg (2009)
16. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 355–371. Springer, Heidelberg (2011)
17. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: 12th USENIX Security, Washington, DC, US, pp. 1–14 (2003)
18. Fog, A.: How to optimize for the Pentium processor, August 1996. https://notendur.hi.is/hh/kennsla/sti/h96/pentopt.txt
19. Fog, A.: How to optimize for the Pentium family of microprocessors, April 2004. https://cr.yp.to/2005-590/fog.pdf
20. Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs: an optimization guide for assembly programmers and compiler makers, January 2016. http://www.agner.org/optimize/microarchitecture.pdf
21. Garner, H.L.: The residue number system. IRE Trans. Electron. Comput. **EC–8**(2), 140–147 (1959)
22. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 444–461. Springer, Heidelberg (2014)
23. Gopal, V., Guilford, J., Ozturk, E., Feghali, W., Wolrich, G., Dixon, M.: Fast and constant-time implementation of modular exponentiation. In: Embedded Systems and Communications Security, Niagara Falls, NY, US (2009)
24. Gueron, S.: Efficient software implementations of modular exponentiation. J. Crypt. Eng. **2**(1), 31–43 (2012)
25. Heninger, N., Shacham, H.: Reconstructing RSA private keys from random key bits. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 1–17. Springer, Heidelberg (2009)
26. Wei-Ming, H.: Reducing timing channels with fuzzy time. In: 1991 Computer Society Symposium on Research Security and Privacy, Oakland, CA, US, pp. 8–20 (1991)
27. İnci, M.S., Gülmezoğlu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. IACR Cryptology ePrint Archive, Report 2015/898, September 2015
28. Intel 64 & IA-32 AORM: Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation, April 2012
29. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In: S&P, San Jose, CA, US (2015)
30. Irazoqui, G., Eisenbarth, T., Sunar, B.: Systematic reverse engineering of cache slice selection in Intel processors. IACR Cryptology ePrint Archive, Report 2015/690, July 2015
31. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
32. Kocher, P., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. J. Cryptogr. Eng. **1**, 5–27 (2011)

33. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
34. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**, 613–615 (1973)
35. LibreSSL Project. https://www.libressl.org
36. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: S&P, San Jose, CA, US, pp. 605–622, May 2015
37. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse engineering intel last-level cache complex addressing using performance counters. In: Bos, H., et al. (eds.) RAID 2015. LNCS, vol. 9404, pp. 48–65. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26362-5_3
38. Mozilla: Network security services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
39. OpenSSL Project. https://openssl.org
40. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: 2006 CT-RSA (2006)
41. Percival, C.: Cache missing for fun and profit. In: BSDCan 2005, Ottawa, CA (2005)
42. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: Reverse engineering Intel DRAM addressing and exploitation (2015). arXiv Preprint arXiv:1511.08756
43. Quisquater, J.-J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In: E-Smart 2001, Cannes, FR, pp. 200–210, September 2001
44. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. CACM **21**, 120–126 (1978)
45. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. J. Cryptol. **23**(1), 37–71 (2010)
46. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Nyberg, K. (ed.) CT-RSA 2015. LNCS, vol. 9048, pp. 3–21. Springer, Heidelberg (2015)
47. Wang, Y., Suh, G.E.: Efficient timing channel protection for on-chip networks. In: 6th NoCS, Lyngby, Denmark, pp. 142–151 (2012)
48. Zhenyu, W., Zhang, X., Wang, H.: Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In: 21st USENIX Security, Bellevue, WA, US (2012)
49. Yarom, Y., Falkner, K.: Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In: 23rd USENIX Security, San Diego, CA, US, pp. 719–732 (2014)
50. Yarom, Y., Ge, Q., Liu, F., Lee, R.B., Heiser, G.: Mapping the Intel last-level cache, September 2015. http://eprint.iacr.org/
51. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: 19th CCS, Raleigh, NC, US, pp. 305–316, October 2012