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
yval@cs.adelaide.edu.au
[2] Technion and Tel Aviv University
danielg3@cs.technion.ac.il
[3] University of Pennsylvania
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 [32], 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 means 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 [42], power consumption [30, 31], or acoustic emanation [21], 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, 15, 16] and cache attacks which observe cache access patterns [37, 38, 48].

Percival [38] published in 2005 the first *cache attack*, which targeted the OpenSSL [41] 0.9.7c implementation of RSA. In this attack, the attacker and the victim are colocated on the same machine and processor, and thus share the 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 executing the victim 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 [10]. 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 [38] 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 [13]. 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 Osvik et al. [37] and Bernstein [7] 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 [27], 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 [22, 23] and to recommend the use of the technique for side-channel mitigation [11, 12]. Consequently, the technique is in widespread use in the current versions of OpenSSL and its forks, such as LibreSSL [40] and BoringSSL [9]. It is also used in other cryptographic libraries, such as the Mozilla Network Security Services (NSS) [36].

1.2 Our Contribution

In this work we present CacheBleed—the first side-channel attack to exploit cache-bank conflicts. In Section 3, we describe how CacheBleed creates contention on a cache bank and measures the timing variations due to conflicts. We use CacheBleed to attack the scatter-gather implementation of RSA in the current version of OpenSSL (1.0.2f). After observing 16,000 decryptions, we are able to recover 60% of the exponent bits. Our attack is described in more detail in Section 4. To find the remaining bits we adapt the Heninger-Shacham algorithm [24] to the information we collect with CacheBleed. Our offline attack requires two CPU hours to completely reconstruct the private key, which

our parallelized implementation was able to compute in minutes. Section 5 describes our implementation in more detail.

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 [40] and BoringSSL [9]. Other cryptographic libraries, such as the Mozilla Network Security Services (NSS) [36] 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 2.20GHZ clock. Our target machine is running CentOS 6.7 installed with its default parameters and with huge pages enabled.

Current Status. We reported our results to the developers of the four cryptographic libraries mentioned above. The OpenSSL team assigned CVE-2016-0702 to the issue and has worked with us on a countermeasure which has been released as part of OpenSSL 1.0.2g. The BoringSSL team reported that they will incorporate the OpenSSL countermeasure. The LibreSSL team is working on a fix, but does not have a timeline yet. We also notified NSS. As of writing this paper, they have not given us a timeline for release of countermeasures.

2 Background

2.1 OpenSSL's RSA implementation

RSA [43] 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 $c^d \bmod N$.

RSA-CRT. RSA decryption is often implemented using the Chinese remainder theorem (CRT), which provides a speedup. 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 [20]:

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

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 $b = \sum_{i=0}^{\lceil n/w \rceil - 1} 2^{wi} b_i$ 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.

Algorithm 1: Fixed-window exponentiation

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

//Precomputation
$a_0 \leftarrow 1$
for $j = 1, \ldots, 2^w - 1$ **do**
  | $a_j \leftarrow a_{j-1} \cdot a \bmod k$
end

//Exponentiation
$r \leftarrow 1$
for $i = \lceil n/w \rceil - 1, \ldots, 0$ **do**
  | **for** $j = 1, \ldots, w$ **do**
  | | $r \leftarrow r^2 \bmod k$
  | **end**
  | $r \leftarrow r \cdot a_{b_i} \bmod k$
end
return r

Sliding-Window Exponentiation.   The *sliding-window algorithm*, represents the exponent b as a sequence of digits b_i such that $b = \sum_{i=0}^{n-1} 2^i b_i$, 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 for 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 that can distinguish between squaring and multiplication operations. Furthermore, the number of squaring operations between consecutive mul-

4

tipliers 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 [38] attack, which exploits these leaks, OpenSSL changed to using 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 the spatial locality of memory accesses to bridge the speed gap between the faster CPU and the 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 [29, 35, 49]. 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 instruction. 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. Acıçmez 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 [37, 38] 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 be longer than accessing data still in the cache. Hence, the attacker can identify the cache sets that the victim has accessed. Persistent-state channels have targeted the L1 data cache [7, 14, 37, 38], the L1 instruction cache [1, 4, 50], the branch prediction buffer [2, 3], the last-level cache [26, 28, 34, 45, 48], and DRAM open rows [39]. The PRIME+PROBE attack was used to recover the accessed multipliers in the sliding-window exponentiation of OpenSSL 0.9.7c [38] and of GnuPG 1.4.18 [26, 34].

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 [33] who suggests that processes can leak information by modifying their CPU usage. Covert channels were also observed with shared bus contention [25, 47], Acıçmez 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. Hence, no secret information was obtained.

Another transient-state channel uses bus contention to leak side-channel information [46]. 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 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* [11, 12]. This
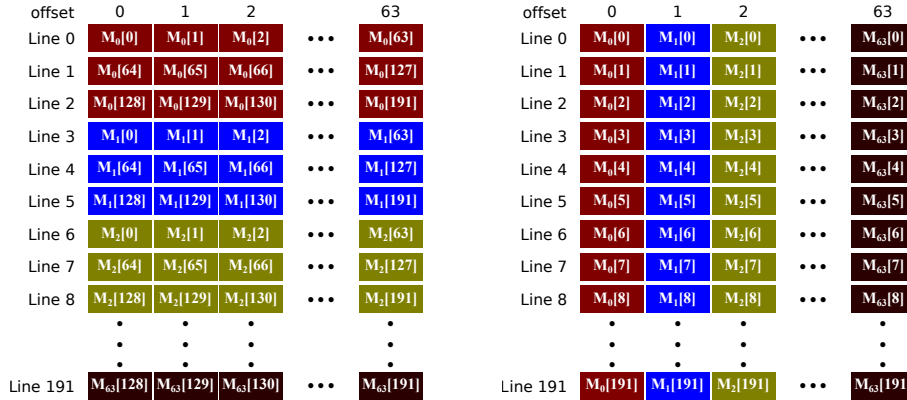
Conventional layout

offset	0	1	2		63
Line 0	$M_0[0]$	$M_0[1]$	$M_0[2]$	$\cdots$	$M_0[63]$
Line 1	$M_0[64]$	$M_0[65]$	$M_0[66]$	$\cdots$	$M_0[127]$
Line 2	$M_0[128]$	$M_0[129]$	$M_0[130]$	$\cdots$	$M_0[191]$
Line 3	$M_1[0]$	$M_1[1]$	$M_1[2]$	$\cdots$	$M_1[63]$
Line 4	$M_1[64]$	$M_1[65]$	$M_1[66]$	$\cdots$	$M_1[127]$
Line 5	$M_1[128]$	$M_1[129]$	$M_1[130]$	$\cdots$	$M_1[191]$
Line 6	$M_2[0]$	$M_2[1]$	$M_2[2]$	$\cdots$	$M_2[63]$
Line 7	$M_2[64]$	$M_2[65]$	$M_2[66]$	$\cdots$	$M_2[127]$
Line 8	$M_2[128]$	$M_2[129]$	$M_2[130]$	$\cdots$	$M_2[191]$
$\vdots$					
Line 191	$M_{63}[128]$	$M_{63}[129]$	$M_{63}[130]$	$\cdots$	$M_{63}[191]$

Scatter-gather layout

offset	0	1	2		63
Line 0	$M_0[0]$	$M_1[0]$	$M_2[0]$	$\cdots$	$M_{63}[0]$
Line 1	$M_0[1]$	$M_1[1]$	$M_2[1]$	$\cdots$	$M_{63}[1]$
Line 2	$M_0[2]$	$M_1[2]$	$M_2[2]$	$\cdots$	$M_{63}[2]$
Line 3	$M_0[3]$	$M_1[3]$	$M_2[3]$	$\cdots$	$M_{63}[3]$
Line 4	$M_0[4]$	$M_1[4]$	$M_2[4]$	$\cdots$	$M_{63}[4]$
Line 5	$M_0[5]$	$M_1[5]$	$M_2[5]$	$\cdots$	$M_{63}[5]$
Line 6	$M_0[6]$	$M_1[6]$	$M_2[6]$	$\cdots$	$M_{63}[6]$
Line 7	$M_0[7]$	$M_1[7]$	$M_2[7]$	$\cdots$	$M_{63}[7]$
Line 8	$M_0[8]$	$M_1[8]$	$M_2[8]$	$\cdots$	$M_{63}[8]$
$\vdots$					
Line 191	$M_0[191]$	$M_1[191]$	$M_2[191]$	$\cdots$	$M_{63}[191]$

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

approach is manifested in the patch Intel contributed to the OpenSSL project to mitigate the Percival [38] 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 [13]. 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. [22] 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 [23] 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.

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 Figure 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

---

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

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.
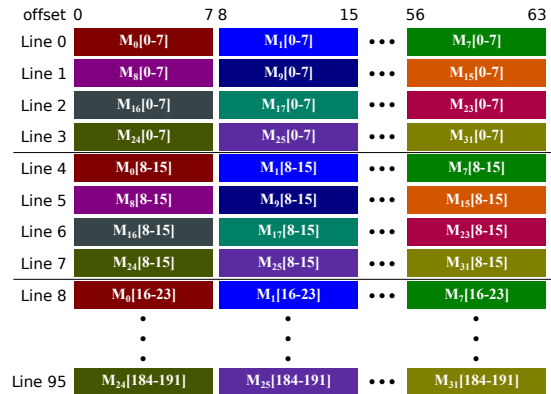

Fig. 2. The memory layout of the multipliers table in OpenSSL

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 [23].

2.5 Intel L1 cache banks

With the introduction of superscalar computing in Intel processors, cache bandwidth became a bottleneck to the 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 [17] 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 [18, 19, 27].

Both Osvik et al. [37] and Bernstein [7] mention that cache-bank conflicts cause timing variations and warn that these may result in a timing channel which may leak low address bits information. Tromer et al. [44] 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 cryptographc 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 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 1.1. 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.

We run the attack code on an Intel Xeon E5-2430 processor—a six-core Sandy Bridge processor, with a clock rate of 2.20GHz. 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

---

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

```
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 1.1. Cache-Bank Collision Attack Code

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 [19]. 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 Figure 3. In
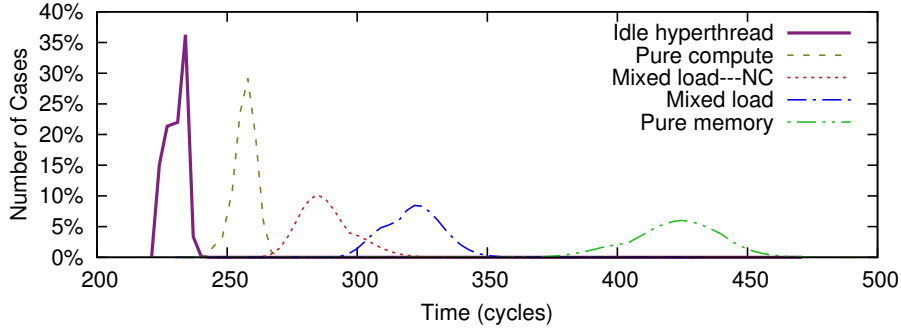
Fig. 3. Distribution of time to read 256 entries from a cache bank.

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.

4 Attacking the OpenSSL Implementation of Modular Exponentiation

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 Section 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 Figure 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 1.1, we create a sequence of measurements of cache-bank conflicts. As mentioned in Section 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.
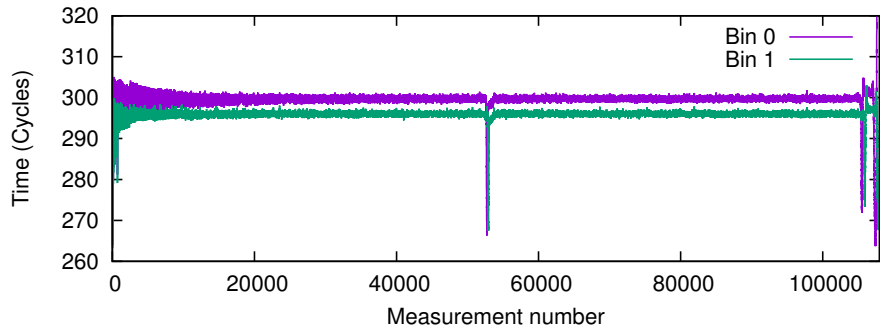
11

Fig. 4. Measurement trace of OpenSSL RSA decryption

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.
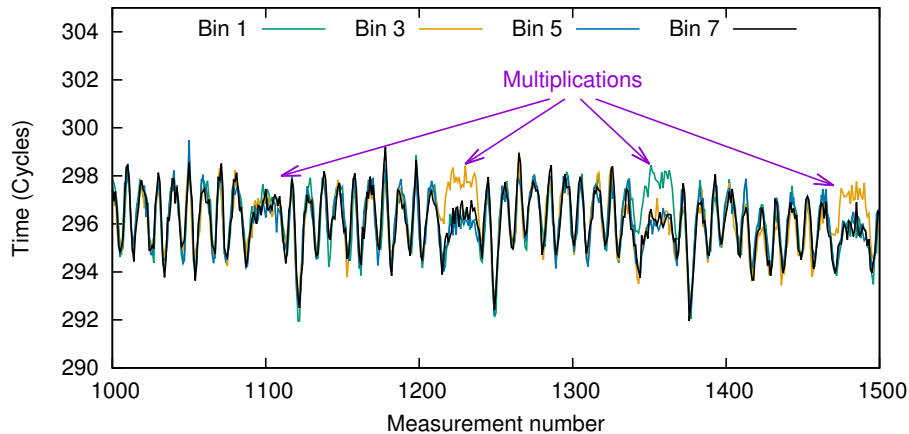

Fig. 5. Measurement trace of OpenSSL RSA decryption—detailed view

Identifying Multiplication Operations. Next, we show that is also possible to identify the individual multiplication operations performed during the modular exponentiation operation. Indeed, Figure 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

12

the OpenSSL exponentiation repeatedly calculate sequences of five modular squaring and reduction operations followed by a modular multiplication.

Identifying Multiplier Values.   We notice 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 section 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 Figure 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 [48] 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 is not enough. 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. On Linux, the timer ticks every millisecond. Each modular exponentiation in the calculation of a 4096-bit RSA-CRT decryption takes 5 ms. Consequently, we experience 5–6 timer interrupts during the exponentiation. The attacker can identify the timer interrupts 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, the attacker can assume that it was interrupted and ignore the measurement.

The attacker, however, does not have exact information on the interrupts that affect the victim. Consequently, the clocks at the victim and at the attacker drift. The result is that 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 the noise, we pass the signal through a low-pass filter. this removes the high frequencies from the signal, and highlights the behavior at a resolution of a multiplication. Figure 7 shows the result of passing the above trace through the filter. In the figure, we can clearly see three multiplications, using bins 7, 5 and 1.

Aligning traces of multiple bins.   As discussed above, measurements in even bins are on average slower 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,
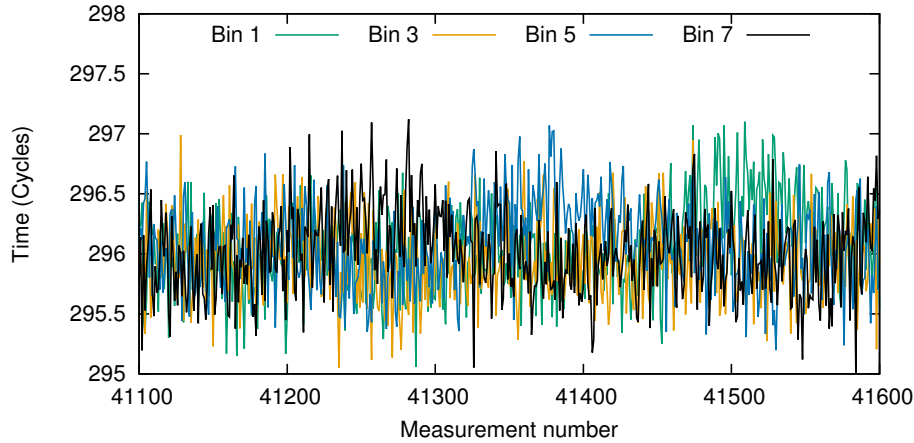
Fig. 6. CacheBleed average trace towards the end of the exponentiation

when we measure over a fixed period of time, traces of even bins will be shorter, i.e. have less measurements, than traces of odd bins. This create a clock shift that becomes larger as the exponentiation progresses.

To normalize the traces we remove element 0 of the frequency domain. This effectively subtracts the trace's average from each trace measurement.

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 may 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 Section 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.
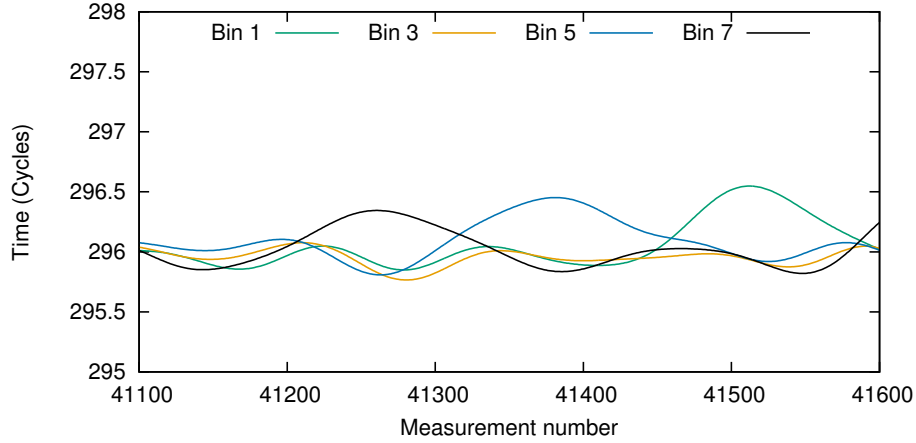
14

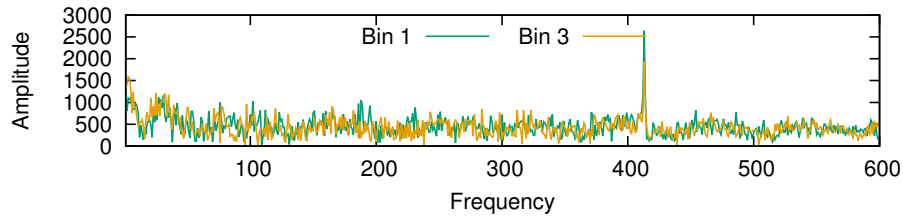**Fig. 7.** Measurement trace after a lowpass filter

Fig. 8. The frequency spectrum of a trace

5 Recovering the RSA Public 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 [24] and İnci et al. [26].

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 [26], 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

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

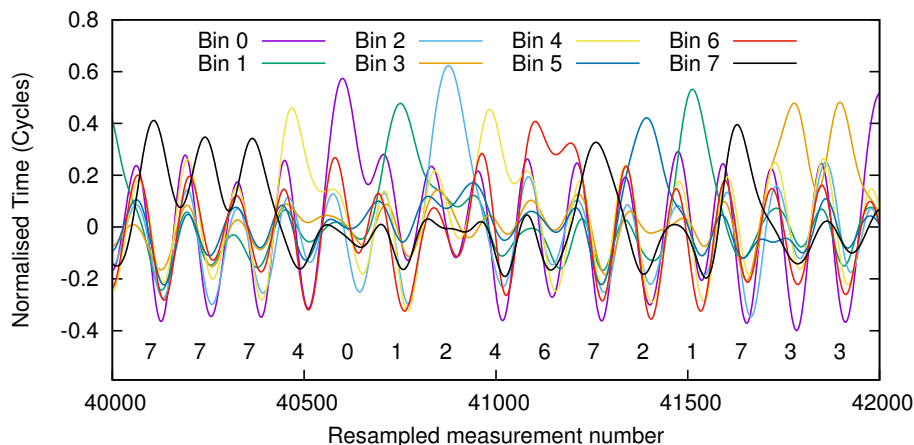Reducing modulo e yields $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$.

15

Fig. 9. Normalized resampled traces

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 Equation 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 [24], 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 [24], 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 2.30GHz 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 minutes. 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 [19, 27]. But, as Haswell does show timing variations that depend on low address bits [19], it may be vulnerable to similar attacks. Furthermore, this solution does not apply to the Sandy Bridge processors currently in the market.

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.

CacheBleed demonstrates that secret-dependent memory access at a finer than cache line granularity is vulnerable to timing attacks. In particular, the scatter-gather technique is not safe. Sensitive software should be written to eliminate all secret-dependent memory access.

7 Conclusions

In this work we present 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. [37] 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. [44].

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.

Acknowledgements

Bibliography

[1] Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *CSAW*, Fairfax, VA, US, 2007.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openSSL and necessary software countermeasures. In *11th IMA International Conference on Cryptography and Coding*, pages 185–203, Cirencester, UK, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *2007 CT-RSA*, pages 225–242, 2007.

[4] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, Santa Barbara, CA, US, 2010.

[5] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, Vienna, AT, 2007.

[6] Donald B. Alpert, Mustafiz R. Choudhury, and Jack D. Mills. Interleaved cache for multiple accesses per clock cycle in a microprocessor. US Patent 5559986, Sep 1996.

[7] Daniel J Bernstein. Cache-timing attacks on AES, 2005. Preprint available at http://cr.yp.to/papers.html#cachetiming.

[8] Daniel J. Bernstein and Peter Schwabe. A word of warning. CHES'13 Rump Session, Aug 2013.

[9] BoringSSL. Boringssl. https://boringssl.googlesource.com/boringssl/.

[10] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *CRYPTO'89*, pages 400–407, Santa Barbara, CA, US, Aug 1989.

[11] Ernie Brickell. Technologies to improve platform security. CHES'11 Invited Talk, Sep 2011. URL http://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf.

[12] Ernie Brickell. The impact of cryptography on platform security. CT-RSA'12 Invited Talk, Feb 2012. URL http://www.rsaconference.com/writable/presentations/file_upload/cryp-106.pdf.

[13] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, Feb 2006.

[14] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *15th ASIACRYPT*, pages 667–684, Tokyo, JP, Dec 2009.

[15] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *16th ESORICS*, Leuven, BE, 2011.

[16] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security*, pages 1–14, Washington, DC, US, 2003.

[17] Agner Fog. How to optimize for the Pentium processor. https://notendur.hi.is/hh/kennsla/sti/h96/pentopt.txt, Aug 1996.

[18] Agner Fog. How to optimize for the Pentium family of microprocessors. `https://cr.yp.to/2005-590/fog.pdf`, Apr 2004.

[19] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `http://www.agner.org/optimize/microarchitecture.pdf`, Jan 2016.

[20] Harvey L. Garner. The residue number system. *IRE Trans. Electron. Computers*, EC-8(2):140–147, Jun 1959.

[21] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pages 444–461, Santa Barbara, CA, US, Aug 2014.

[22] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and constant-time implementation of modular exponentiation. In *Embedded Systems and Communications Security*, Niagara Falls, NY, US, Sep 2009.

[23] Shay Gueron. Efficient software implementations of modular exponentiation. *J. Cryptographic Engin.*, 2(1):31–43, May 2012.

[24] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In *CRYPTO 2009*, pages 1–17, Santa Barbara, CA, US, Aug 2009.

[25] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *1991 Comp. Soc. Symp. Research Security & Privacy*, pages 8–20, Oakland, CA, US, 1991.

[26] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. IACR Cryptology ePrint Archive, Report 2015/898, Sep 2015.

[27] Intel 64 & IA-32 AORM. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Apr 2012.

[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 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, May 2015.

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. IACR Cryptology ePrint Archive, Report 2015/690, Jul 2015.

[30] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397, 1999.

[31] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engin.*, 1:5–27, Apr 2011.

[32] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113. Springer, 1996.

[33] Butler W. Lampson. A note on the confinement problem. *CACM*, 16:613–615, 1973.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, pages 605–622, San Jose, CA, US, May 2015.

[35] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, Kyoto, Japan, Nov 2015.

[36] Mozilla. Network security services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *2006 CT-RSA*, 2006.

[38] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[39] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Reverse engineering Intel DRAM addressing and exploitation. *arXiv preprint arXiv:1511.08756*, 2015.

[40] LibreSSL Project. Libressl. https://www.libressl.org, .

[41] OpenSSL Project. Openssl. https://openssl.org, .

[42] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *E-Smart'01*, pages 200–210, Cannes, FR, Sep 2001.

[43] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 21:120–126, 1978.

[44] Eran Tromer, DagArne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[45] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *2015 CT-RSA*, pages 3–21, San Francisco, CA, USA, Apr 2015.

[46] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *6th NoCS*, pages 142–151, Lyngby, Denmark, 2012.

[47] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *21st USENIX Security*, Bellevue, WA, US, 2012.

[48] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security*, pages 719–732, San Diego, CA, US, 2014.

[49] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. http://eprint.iacr.org/, Sep 2015.

[50] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th CCS*, pages 305–316, Raleigh, NC, US, Oct 2012.