New Results on Instruction Cache Attacks

Onur Acıiçmez[1], Billy Bob Brumley[2⋆], and Philipp Grabher[3⋆⋆]

[1] Samsung Electronics, USA
o.aciicmez@samsung.com
[2] Aalto University School of Science and Technology, Finland
billy.brumley@tkk.fi
[3] University of Bristol, UK
grabher@cs.bris.ac.uk

Abstract. We improve instruction cache data analysis techniques with a framework based on vector quantization and hidden Markov models. As a result, we are capable of carrying out efficient automated attacks using live I-cache timing data. Using this analysis technique, we run an I-cache attack on OpenSSL's DSA implementation and recover keys using lattice methods. Previous I-cache attacks were proof-of-concept: we present results of an actual attack in a real-world setting, proving these attacks to be realistic. We also present general software countermeasures, along with their performance impact, that are not algorithm specific and can be employed at the kernel and/or compiler level.

1 Introduction

Cache-timing attacks are emerging attack vectors on security-critical software. They belong to a larger group of cryptanalysis techniques within side-channel analysis called Microarchitectural Attacks (MA). Microarchitectural Cryptanalysis focuses on the effects of common processor components and their functionalities on the security of software cryptosystems. The main characteristic of microarchitectural attacks, which sets them aside from classical side-channel attacks, is the simple fact that they exploit the microarchitectural behavior of modern computer systems. MA techniques have been shown to be effective and practical on real-world systems. For example, Osvik et. al. used cache attacks on dm-crypt application to recover AES keys [11]. Ristenpart et. al. successfully applied cache attacks in Amazon's EC2 cloud infrastructure and showed the information leakage from one virtualized machine to another [14]. Several studies showed the effectiveness of these attacks on various cryptosystems including AES [11, 5], RSA [13, 4, 3], and ECC [6]. Popular cryptographic libraries such as OpenSSL have gone under several revisions to mitigate different MA attacks, c.f. e.g. [1].

⋆ Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).
⋆⋆ Supported in part by EPSRC grant EP/E001556/1.

There are usually two types of caches in today's processors, data cache and instruction cache, which have different characteristics, and hence we have two different types of cache-timing attacks. Our work presented in this paper deals only with instruction caches. I-cache attacks rely on the fact that instruction cache misses increase the execution time of a software. An adversary executes a so-called spy process on the same machine that his target software (e.g. an encryption application) is running on and this spy uses some techniques to keep track of the changes in the state of I-cache during the execution of the target software. Knowing the state changes in I-cache may allow the adversary to extract the instruction flow of the target software. Cipher implementations that have key-dependent instruction flows can be vulnerable to I-cache attacks unless effective countermeasures are in place. I-cache analysis technique was introduced in [2]. We have seen I-cache attack vulnerabilities in widely used RSA implementations [4]. Previous works on I-cache analysis were, in a sense, only proof-of-concept attacks. Spy measurements were either taken *within* the cipher process or in a simplified experimental setup.

In this paper, we present several contributions related to I-cache attacks, their data analysis, and countermeasures. We apply the templating cache-timing data analysis framework [6] to I-cache data. It makes use of Vector Quantization (VQ) and Hidden Markov Models (HMM) to automate the side-channel data analysis step. This allows us to mount a lattice attack on an unmodified OpenSSL-DSA implementation and successfully recover DSA keys. These are the first published results of a real-life I-cache attack on a cryptosystem. In a nut-shell, our contributions in this paper include:

- improving I-cache data analysis techniques,
- mounting a lattice attack on OpenSSL's DSA implementation using this improved analysis,
- presenting results of I-cache Analysis in a real-world attack settings,
- and outlining possible countermeasures to prevent I-cache attacks and measuring their performance impacts.

We give an overview of the original I-cache attack of [2] in Section 2 and present the details of our improved attack and our results on OpenSSL-DSA in Section 3. Our results prove the dangers of I-cache attacks and the necessity of employing appropriate countermeasures. We studied some of the possible countermeasures and analyzed their impacts on cipher performance and also on the performance of the entire system. We discuss these countermeasures and present our results in Sections 4 and 5.

2 I-cache Attack Concept

I-cache analysis relies on the fact that instruction cache misses increase the execution time of software applications. Each I-cache miss mandates an access to a higher level memory, i.e., a higher level cache or main memory, and thus results in additional execution time delays. In I-cache analysis, an adversary

runs a so-called spy process that monitors the changes in I-cache. They spy process continuously executes a set of "dummy" instructions in a loop in a particular order and measures how much time it takes to bring the I-cache to a predetermined state. Sec. 3.1 contains an example of such a spy routine.

If another process is running simultaneously with the spy on the same physical core of an SMT processor, the instructions executed by this process will alter the I-cache state and cause evictions of spy's dummy instructions. When the spy measures the time to re-execute its instructions, the latency will be higher for any evicted dummy instructions that must be fetched from a higher memory level. In this manner the spy detects changes in the I-cache state induced by the other (i.e., "spied-on") process and can follow the footprints of this process.

[2] shows an attack on OpenSSL's RSA implementation. They take advantage of the fact that OpenSSL employs sliding window exponentiation which generates a key dependent sequence of modular operations in RSA. Furthermore, OpenSSL uses different functions to compute modular multiplications and square operations that leaves different footprints on I-cache. Thus, a spy can monitor these footprints and can easily determine the operation sequence of RSA. [2] uses a different spy than the one we outline in Sec. 3.1. They try to extract the sequence of multiplication and square operations and thus their spy monitors only the I-cache sets related to these functions. Furthermore, their spy does not take timing measurements for each individual I-cache set, but instead considers a number of sets as a group and takes combined measurements. In our work, the spy takes individual measurements for each I-cache set so that we can monitor each set independently and devise template I-cache attacks.

3 Improved Attack Techniques

In this section, we present our improvements to I-cache timing data analysis and subsequently apply the results to run an I-cache attack on OpenSSL's DSA implementation (0.9.8l) to recover keys. We concentrate on Intel's Atom processor featuring Intel's HyperThreading Technology (HT).

3.1 Spying on the Instruction Cache

The templating framework in [6] used to analyze cache-timing data assumes vectors of timing data where each component is a timing measurement for a distinct cache set. We can realize this with the I-cache as well using a spy process that is essentially the I-cache analogue of Percival's D-cache spy process [13]. It pollutes the I-cache with its own data, then measures the time it takes to re-execute code that maps to a distinct set, then repeats this procedure indefinitely for any desired I-cache sets.

To this end, we outline a generic instruction cache spy process; the example here is for the Atom's 8-way associative 32KB cache, $c = 64$ cache sets, but is straightforwardly adaptable to other cache structures. We lay out contiguous 64-byte regions of code (precisely the size of one cache line) in labels

$\mathcal{L} = \{L_0, L_1, \ldots, L_{511}\}$. Denote subsets $\mathcal{L}_i = \{L_j \in \mathcal{L} : j \bmod c = i\}$ in this case each with cardinality eight, where all regions map to the same cache set yet critically do not share the same address tag. These subsets naturally partition $\mathcal{L} = \bigcup_{i=0}^{c-1} \mathcal{L}_i$. Observe that stepping through a given $\mathcal{L}_i$ pollutes the corresponding cache set i and repeating for all i completely pollutes the entire cache.

The spy steps iteratively through these $\mathcal{L}_i$ and measures their individual execution time. For example, it begins with regions that map to cache set zero: $\mathcal{L}_0 = \{L_0, L_{64}, L_{128}, \ldots, L_{448}\}$, stores the execution time, then continues with cache set one: $\mathcal{L}_1 = \{L_1, L_{65}, \ldots, L_{449}\}$ and so on through all $0 \leq i < c$. For each i we get a single latency measurement, and for all i a vector of measurements: repeating this process gives us the desired side-channel. For completeness, we provide a code snippet in Fig. 1. The majority of the code is `nop` instructions, but they are only used for padding and never executed. Note `rdtsc` is a clock cycle metric.

```
xor %edi, %edi
mov <buffer addr>, %ecx
rdtsc
mov %eax, %esi
jmp L0
.align 4096
L0:
  jmp L64
  .rept 59
  nop
  .endr
L1:
  jmp L65
  .rept 59
  nop
  .endr
...
L64:
  jmp L128
  .rept 59
  nop
  .endr
...
L448:
  rdtsc
  sub %esi, %eax
  movb %al, (%ecx,%edi)
  add %eax, %esi
  inc %edi
  jmp L1
  .rept 49
  nop
  .endr
...
L511:
  rdtsc
  sub %esi, %eax
  movb %al, (%ecx,%edi)
  add %eax, %esi
  inc %edi
  cmp <buffer len>, %edi
  jge END
  jmp L0
```

Fig. 1. Outline of a generic I-cache spy process.

3.2 Realizing the DSA

We use the following notation for the DSA. The parameters include a hash function h and primes p, q such that $g \in \mathbb{F}_p^*$ generates a subgroup of order q. Currently, a standard choice for these would be a 1024-bit p and 160-bit q. Parties select a private key x uniformly from $0 < x < q$ and publish the corresponding public key $y = g^x \bmod p$. To sign a message m, parties select nonce k uniformly from $0 < k < q$ then compute the signature (r, s) by

$$r = g^k \bmod p \bmod q \tag{1}$$

$$s = (h(m) + xr)k^{-1} \bmod q \tag{2}$$

and note OpenSSL pads nonces to thwart traditional timing attacks by adding either q or $2q$ to k.

The performance bottleneck for the above signatures is the exponentiation in (1); extensive literature exists on speeding up said operation. Arguably the most widely implemented method in software is based on the basic left-to-right square-and-multiply algorithm and employs a standard sliding window (see [8, 14.85]). It is a generalization where multiple bits of the exponent can be processed during a given iteration. This is done to reduce the total number of multiplications using a time-memory trade-off. With the standard 160-bit q, a reasonable choice (and what OpenSSL uses) is a window width $w = 4$.

The OpenSSL library includes an implementation of this algorithm, and uses it for DSA computations. Its speed is highly dependent on how the modular squaring and multiplication functions are implemented. Computations modulo p are carried out in a textbook manner using Montgomery reduction. Outside of the reduction step, the actual squaring and multiplication are implemented in separate functions; this is because we can square numbers noticeably faster than we can multiply them.

3.3 The Attack

We aim to determine partial nonce data during the computation of (1) by observing I-cache timings and use said partial data on multiple nonces to mount a lattice attack on (2) to recover the private key x.

In Sec. 3.2 we mention that squaring and multiplication are implemented as two distinct functions. In light of this, it is reasonable to assume that:

- All portions of these two sections of code are *unlikely* to map to the same I-cache sets;
- The load and consequentially execution time of (1) is dependent on their respective availability in the I-cache;
- An attacker capable of taking I-cache timings by executing their *own* code as outlined in Sec. 3.1 in parallel with the computation of (1) can deduce information about the state of the exponentiation algorithm—thus obtaining critical information about k.

The resulting side-channel is a list of vectors where each vector component is a timing measurement for a distinct cache set. We illustrate in Fig. 2, where we hand picked 16 of 64 possible I-cache sets that seemed to carry pertinent information.

Analyzing Timing Data Next, we analyze this data to determine the sequence of states the exponentiation algorithm passed through. Just the sequence of squarings and multiplications that the sliding window algorithm passes through implies a significant amount of information about the exponent input. We utilize the framework of [6] to analyze the timing data, obtain a good guess at the algorithm state sequence, and infer a number of bits for each nonce. The steps include:

- For each operation we wish to distinguish (for example, squaring and multiplication), take a number of exemplar timing vectors that represent the I-cache behavior during said operation; [6, Sec. 4.2] calls this "templating".

- With these templates, create a Vector Quantization (VQ) codebook for each operation; this is done using a standard supervised learning method called LVQ. This can help eliminate noise and reduce the size of the codebook.
- Create a Hidden Markov Model (HMM) that accurately reflects the control flow of the considered algorithm. The observation input to the HMM is the output from VQ.
- Use the Viterbi algorithm to predict the most likely state sequence given a (noisy) observation sequence (VQ output of I-cache timing data).

Vector Quantization. We categorize timing vectors using VQ, which maps the input vectors to their closest (Euclidean distance-wise) representative vector in a fixed codebook. We obtain codebook vectors during a profiling stage of the attack, where we examine timing data from known input to classify the vectors in the codebook. Essentially, this means we setup an environment similar to the one under attack, obtain side-channel and DSA signatures with our own known key, then partition the obtained vectors into a number of sets with fixed labels. These sets represent the I-cache access behavior of the algorithm in different states, such as multiplication and squaring; these are the labels. When running the attack, we classify the incoming timing vectors using VQ. The algorithm state guess is the label of the closest vector in the codebook. To summarize, we guess at the algorithm state based on previously observed (known) algorithm state.

Hidden Markov Models. We also build and train the HMM during the profiling stage, using the classical Baum-Welch algorithm. The training data is the output from VQ above: the observation domain for the HMM is the range of VQ (the labels). As multiplication and squaring steps in the algorithm span multiple timing vectors in the trace, we consider these steps as meta-states, represented explicitly in the HMM by a number of sub-states corresponding to this span. When running the attack, we feed the trace through VQ and send the output to the HMM. The classical Viterbi algorithm outputs the state sequence that maximizes the probability of the observation sequence. To summarize, we guess the algorithm state sequence that best explains the side-channel observations.

Example. In addition to the timing data (rows 0-15) in Fig. 2, we give the VQ output (rows 16-17) and HMM state prediction (rows 18-19). Normally the purpose of any HMM in signal processing is to clean up a noisy signal, but in this case we are able to obtain extremely accurate results from VQ. This leaves little work in the end for the HMM. We chose to template squaring (the dark gray), multiplication (black), and what we can only assume is the Montgomery reduction step (light gray).

Using Partial Nonce Data Having obtained a state sequence guess and thus partial information on nonces k for many signatures, the endgame is a lattice attack.

In such an attack it is difficult to utilize sparse key data, thus an attacker usually concentrates on a fairly long run of consecutive (un)known bits, and

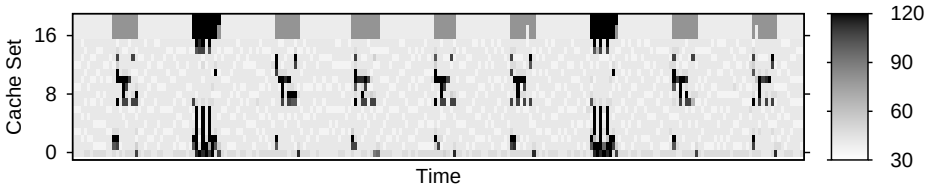

Fig. 2. Live I-cache timing data produced by a spy process running in parallel with an OpenSSL DSA sign operation; roughly 250 timing vectors (in CPU cycles), and time moves left-to-right. The bottom 16 rows are the timing vector components on 16 out of 64 possible cache sets. The top four are meta-data, of which the bottom two are the VQ classification and the top two the HMM state guess given the VQ output. Seven squarings are depicted in dark gray and two multiplications in black.

obtains more equations instead. Furthermore, we experienced that guesses on bits of k get less accurate the farther away they are from the LSB. We sidestep these issues by concentrating on signatures where we believe k has $\{0,1\}\{0\}^6$ in the LSBs—that is, six zeros followed by a zero or one. The top bit is fixed due to the padding, giving us a total of eight known bits separated by a single long run of unknown bits. Experiments suggest we need 37 such signatures to recover the long term key x.

Results We obtained 17K signatures, messages, and corresponding I-cache timing data. Considering we expect the given bit pattern in k with probability 2^{-6}, this number seems unnecessarily high at first glance. Like many practical side-channel attacks, this is due to inherent issues such as noise, context switching, OS scheduling, and lack of synchronization. As our spy process is truly decoupled from the OpenSSL code, running as an independent process, we get absolutely no guarantee that they will execute simultaneously—or when they happen to, for how long.

After obtaining these 17K signatures, our analysis resulted in 75 signatures believed to match the pattern. We ran the lattice attack on five Intel Core2 quad core machines, taking random samples of size 37 until the result yielded a private key that corresponded to the given public key. The first core to succeed did so in 54 minutes, after roughly 3200 lattice attack iterations. Checking afterwards, 59 of these guesses were correct and 16 incorrect.

4 Closing the Instruction Cache Side-Channel

Countermeasures to mitigate the I-cache side-channel can be employed at a hardware and/or a software level. Hardware countermeasures require changes to the micro-architecture and it might take a while until such a new processor generation is available on the market. Previous work in this area proposed using alternative cache hardware, such as Partitioned Caches [12], Partition-locked Caches and Random-permutation Caches [16]. Most current processor designs are driven by performance and power criteria, leaving security as a secondary

consideration; it is questionable whether this view will change in the foreseeable future. In this work, we focus solely on software techniques to address this vulnerability. Such countermeasures can be applied instantly by a software engineer as long as no hardware equivalents are present. In contrast to previously proposed software techniques, which are usually algorithm specific (e.g., Montgomery's powering ladder [9]), our aim is to provide generic methods be employed at the kernel and/or compiler level.

In the following discussion, we have to distinguish between countermeasures applicable to architectures which support SMT and conventional single-threaded processors. While in both architectures multiple threads can exist at the same time, there is a substantial difference in how they are scheduled. Processors with SMT support essentially split a single physical processor into two logical processors by duplicating some sections of the micro-architecture responsible for architectural state. In this way, the OS can schedule two threads/processes to be executed simultaneously on the same processor. These two threads execute literally simultaneously, not in a time-sharing fashion. Memory accesses of both execution threads alter the cache states at the same time.

In contrast, single-threaded processors are only capable to execute a single process/thread at any given point in time. In such architectures, execution time is allocated in time slices to the different processes/threads; by frequently switching between processes/threads, it gives an outward impression that multiple tasks are executed simultaneously. This type of execution is called quasi-parallel execution.

Cache attacks can work on both SMT processors and single-threaded processors. It is easier to run these attacks on SMT because spy and cipher can run simultaneously on different virtual cores in a single physical processor and spy can monitor cipher execution while cipher is performing its computations. Running cache attacks on single-threaded processors is more difficult. An attacker needs to use some tricks to have a "ping-pong" effect between the spy and cipher processes. [10] showed that it is possible to pause the cipher execution at a determined point and let a spy to examine the cache state. [10] exploited an OS scheduling trick to achieve this functionality and devised an attack on the last round of AES. A similar OS trick was shown in [15] to let a malicious party monopolize CPU cycles. [15] proposes to exploit OS scheduling mechanism to steal CPU cycles unfairly. Their cheating idea and the source code can easily be adapted to cache attacks on single-threaded processors.

Disable Multi-threading In general, cache-based side-channel attacks take advantage of the fact that modern computer systems provide multi-threading capability. This fact allows an attacker to introduce an unprivileged spy process to run simultaneously with a security-critical code, thereby deriving secret key information from the state of the I-cache. A simple solution to eliminate this vulnerability is to turn off multi-threading when a security-critical process is scheduled to be executed: since it is the task of the OS to schedule processes/threads, it can simply decide to ignore all unprivileged processes/threads and not run them. On processors with SMT capability, the OS can adopt a

scheduling policy that does not permit to execute another process in parallel with the crypto process. Alternatively, SMT can be turned off in the BIOS. According to Intel, SMT improves performance of multi-threaded applications by up to 30 %. Therefore it needs to be decided on a case-by-case basis if disabling SMT for a more secure processing platform is acceptable from a performance point of view. Disabling multi-threading alone does not suffice to close I-cache side channel. I-cache attacks can be used on single-threaded processors without SMT capability as we explained above.

Fully Disable Caching Another intuitively simple solution to close the information leakage through the I-cache is to disable the cache entirely. The Intel x86 architecture makes the cache visible to the programmer through the CD flag in the control register *cr0*: if said flag is set, caching is enabled otherwise it is disabled. However, such an approach severely affects the performance of the system as a whole. A more fine-grained control sees the cache only disabled when security-critical code is scheduled to be executed.

Partially Disable Caching The x86 caches allow the OS to use a different cache management policy for each page frame. Of particular interest in this context is the PCD flag in control register *cr0* which determines whether the accessed data included in the page frame is stored in the cache or not. In other words, by setting the PCD flag of the page frames containing security-critical code it is possible to partially disable the caching mechanism. While such an approach successfully eliminates the I-cache side-channel we argue that is has a considerable negative impact on performance (albeit not as severe as with completely turning off the cache). The reason is that most cryptographic primitives spend the vast majority of the execution time in some small time-critical code sections; hence, not caching parts of these sections will be reflected in longer execution times.

Cache Flushing Ideally, the processor would provide an instruction to flush the content of the L1 I-cache only. Unfortunately, such an instruction is not yet available on Intel's x86 range of processors. Instead, the WBINVD instruction [7] can be executed during context switches to flush the L1 I-cache. Note, that this instruction invalidates all internal caches, i.e., the instruction cache as well as the data cache; modified cache lines in the data cache are written back to main memory. After that, the instruction signals the external caches, i.e., the L2 and L3 cache to be invalidated. Invalidation and writing back modified data from the external caches proceeds in the background while normal program execution resumes, which partly mitigates the associated performance overhead. OS can flush the cache when a security-critical process such as a cipher switched out and thus the next process scheduled right after the cipher cannot extract any useful information from the cache state. This countermeasure is not effective on SMT systems because flushing happens during context switch and spy that runs simultaneously with a cipher on SMT can still monitor cipher's execution.

Partial Cache Flushing Flushing the entire L1 I-cache negatively affects performance of both the security application as well as of all the other existing

threads. This performance impact can be reduced when following a more fine-grained approach: instead of flushing the entire I-cache we propose to invalidate only those cache sets that contain security-critical instructions via some kind of OS support.

The x86 processor does not include such a mechanism that allows flushing of specific cache sets. Instead, some architectures provide the CLFLUSH instruction [7] capable of invalidating a cache line from the cache hierarchy. This instruction takes the linear address of the cache line to be invalidated as an argument. Consequently, flushing an entire cache set with this instruction would require the knowledge of both the linear address space of the spy process as well as of the security-critical code sections of the crypto process. While the later can be made easily available to the OS, it is much more difficult to reason about the linear address space of the spy process. This instruction is not suitable for our purposes as a result.

However, flushing of specific cache sets on x86 processors can still be accomplished by beating an attacker at his own game. The simple idea is to divert the spy process from its intended use by employing it as defence mechanism; essentially, the kernel integrates a duplicate spy process into the context switch. This permits the eviction of security-critical code sections from the I-cache each time security-critical code is switched out.

At first glance, it might seem that invalidating only those cache lines containing security-critical code before giving control to another process (possibly the spy process) can defeat the I-cache attack. However, from the spy's point of view, it makes no difference whether lines with security-critical code have been invalidated or not: in any case, the spy process will measure a longer execution time since the crypto process has evicted a cache line belonging to the spy. Therefore, invalidating only cache lines with security-critical code is not sufficient and the entire sets that hold them need to be invalidated. Similar to flushing the entire cache, partial flushing is not effective on SMT processors as explained above.

Cache-conscious Memory Layout Fundamentally, I-cache attacks rely on the premise that the security-critical code sections or parts of them map to different regions in the I-cache. By mapping these security-critical code sections exactly to the same regions in the cache, the I-cache attacks can no longer recover the operation sequence. However, in some cases this approach might not be sufficient. For example, consider the case of two security-critical code sections that are of equal size and map to the same sets, where the majority of execution time of the two security-critical code sections is spent in disjoint cache sets. In such a scenario, it is still highly likely that the spy observes distinct traces despite the appropriate alignment in memory. Cache-conscious memory layout can be accomplished either by a compiler or via OS support. Given the I-cache parameters and the security-critical code sections, a compiler can generate an executable resistant against I-cache attacks by appropriately aligning said sections. To balance the sizes of these sections, it might be necessary to add some dummy instructions, e.g., NOPs, before and/or after the sections; this padding with dummy operations implies some performance penalty and results in an in-

crease in the size of the executable. Alternatively, the OS can be in charge of placing the security-critical code sections in such a way in memory that they map to the same regions in the cache if the cache is physically addressed. For that, the executable needs to specify the memory sections with security-critical information. Similar to the compiler approach, additional dummy operations might be required to make the security-critical code sections equal in size. None of the above countermeasures provide an effective yet practical mechanism for SMT systems, except cache-conscious memory layout . This countermeasure incurs very low overhead as we will explain in the next section and it is also effective on SMT systems.

5 Performance Evaluation

All our practical experiments were conducted on a Intel Core Duo machine running at 2.2 GHz with a Linux (Ubuntu) Operating System. To minimize the variations in our timing measurements due to process-interference we used the process affinity settings to bind the crypto process to one core and assigned all the other processes to the other core.

Performance impact on the crypto process For the performance evaluation of our proposed software countermeasures we used the RSA decryption function of OpenSSL (version 0.9.8d) as a baseline. Table 1 summarizes the performance impact of our proposed countermeasures upon OpenSSL/RSA in comparison to the baseline implementation; results are given for different key lengths, i.e., 1024-bits, 2048-bits and 4096-bits.

Implementation		1024-bit	2048-bit	4096-bit
Baseline OpenSSL/RSA	Execution time (in ms)	1.735	9.606	57.9
	Decryptions/s	576.3	104	17.3
OpenSSL/RSA with cache disabled	Execution time (in ms)	1273	7204	45060
	Decryptions/s	0.8	0.1	0.02
OpenSSL/RSA with cache flushing	Execution time (in ms)	1.888	11.192	60.6
	Decryptions/s	530	89.3	16.5
OpenSSL/RSA with partial flushing	Execution time (in ms)	1.734	9.535	58.2
	Decryptions/s	576.8	104.9	17.1
OpenSSL/RSA with cache-conscious layout	Execution time (in ms)	1.755	9.727	58.2
	Decryptions/s	570	102.8	17.2

Table 1. Performance evaluation of the proposed countermeasures

The performance evaluation supports our claim that turning the cache off results in an immense performance overhead. For instance, execution of a 1024-bit OpenSSL/RSA with a disabled cache leads to a 3-orders of magnitude degradation in performance. This experiment was conducted with the help of a simple kernel module which turns the cache off when loaded into the kernel and turns

the cache on again when unloaded. Similarly, we expect an unacceptable impact on performance when just partially disabling the cache since this forces the processor to repeatedly fetch instructions from the slow main memory; for that reason we refrained from investigating this approach in more detail. Flushing the cache hierarchy during a context switch incurs a performance overhead of about $5 - 15$ % for the different key sizes. Even more severe than this non-trivial performance penalty is the significant increase in context switch time: using Intel's RDTSC instruction, we measured a 10-fold increase. The performance overhead from both an application as well as OS point of view can be significantly reduced when only invalidating the cache sets containing security-critical code. For that, we aligned the spy process appropriately in the context switch routine to evict the cache sets that hold data of both the OpenSSL/RSA multiplication and squaring routines; in total, it was necessary to evict 29 sets (i.e., 18 sets are occupied by multiplication instructions and 11 sets by squaring instructions) from the instruction cache.

From Table 1 it appears that no noticeable performance overhead is associated with this countermeasure. This result is somewhat expected since the overhead of bringing the evicted instructions from the L2 cache back into the instruction cache is negligible.

Finally, we investigated the cache-conscious memory layout approach. It was necessary to pad the OpenSSL/RSA squaring routine with 406 NOP instructions in total so that it matches the size of the OpenSSL/RSA multiplication. However, having the same code size alone does not prevent information leakage; the security-critical code sections also need to be aligned in memory in such a way that they map into the same cache sets. This can be done by rewriting the linker script to control the memory layout of the output file. In more detail, we first used the gcc compiler option "-ffunction-sections" to place the two security-critical code sections in a separate ELF section each. Then, we redefined the memory model so that each section is placed at a known address such that they will be placed in the same sets in the cache. The performance overhead associated with this countermeasure is so minimal that in practice it can be regarded as negligible.

Performance impact on the system caused by the countermeasures Our proposed software countermeasures may have a negative impact on other processes that are running concurrently with a security-critical application. This impact might be in particular noticeable for the solution where the entire cache content is flushed during a context switch.

To estimate the impact on the system as a whole, we ran the SPEC2000int benchmark simultaneously with a security-critical application; this means the processor's entire cache hierarchy is invalidated at regular intervals, i.e., every time the security-critical process is switched out. Figure 3 illustrates this performance impact on the SPEC benchmark in presence of this countermeasure.

On average, invalidation of the cache during context switches results in a 10 % degradation in performance. This decline is caused by bringing data back into the cache after it has been discarded during the context switch. Note, this

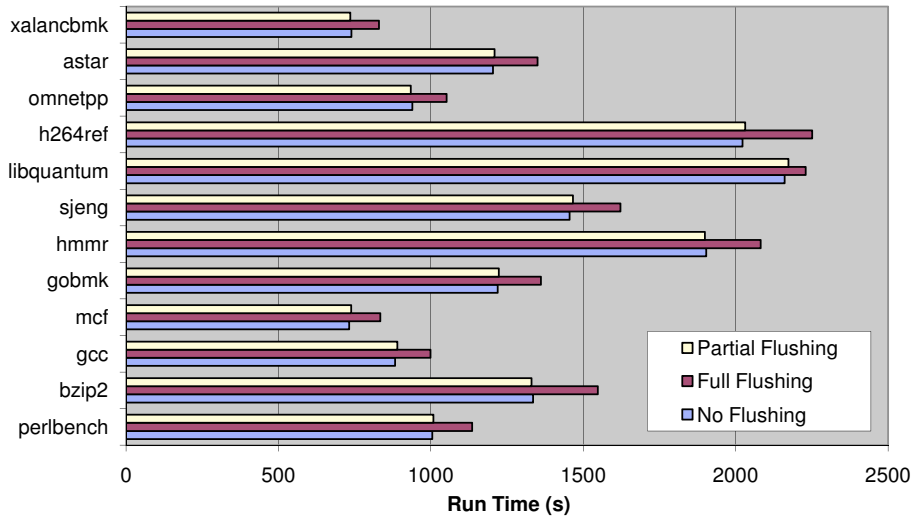

Fig. 3. Performance impact of cache flushing on the SPEC2000int benchmark.

overhead gives an estimation for the worst-case scenario and the impact will typically be less severe on systems where security-critical applications are executed less frequently. Figure 3 also shows the run time of the SPEC benchmark in presence of partial cache eviction as a countermeasure instead. Essentially, the performance impact on the system as a whole is negligible in this case. Further, some of our countermeasures can influence the time it takes to perform a context switch. If there is no process using our countermeasures, which will probably be the case most of the time, the only extra work that needs to be done is to check a flag (i.e., the flag that indicates whether the current process needs to be protected). However, switching out a security-critical process possibly requires some additional work for the scheduler which results in a longer execution time of the context switch. This behaviour is in particular apparent when using the cache flushing approach since the scheduler needs to wait until all dirty data cache lines have been written back to maintain memory coherence. Consequently, depending on the type of process that is switched out, a different amount of time is spent in the context switch routine. This can pose a serious problem to real-time systems, where highly deterministic behaviour is required. Note that partial eviction has a considerably smaller impact on the context switch; in theory, for each cache set that contains security-critical instructions, the OS simply needs to execute a small number of appropriately aligned dummy instructions and this number needs to be equal or larger than the associativity of the I-cache.

6 Conclusions

We presented improved I-cache analysis techniques based on vector quantization and hidden Markov models. The analysis is automated and fast, capable of

analyzing large volumes of concrete I-cache timing data. This can be used to perform automated I-cache attacks.

We demonstrated its effectiveness by carrying out an I-cache attack on an unmodified version of OpenSSL's DSA implementation (0.9.8l). We used the framework to process the timing data from thousands of signatures and subsequently recovered keys using lattice methods. This attack is automated, recovering a DSA private key within an hour.

Our study clearly proves that I-cache cryptanalysis is realistic, practical, and a serious security threat for software systems. We believe it is necessary to conduct a thorough analysis on current software cryptosystems to detect I-cache analysis (more generally Microarchitectural Analysis) vulnerabilities. We already saw several MA vulnerabilities in cryptographic software like OpenSSL and they were fixed by specific algorithm-level solutions such as removing extra reduction step from Montgomery multiplication. However, it is crucial to design generic algorithm-agnostic mitigation mechanisms.

Mitigation mechanisms can be employed at a hardware and/or a software level. Hardware countermeasures require changes to the micro-architecture and much longer time to hit the market compared to software countermeasures. Thus, we focused solely on generic software-level mitigations in our work and presented some countermeasures to close I-cache side channel. We studied their impacts on cipher performance and also on the performance of the overall system. Naive approaches such as disabling cache or flushing the entire cache before or after the execution of security critical software have high performance overheads associated with them. Thus, such approaches are far from gaining wide usage due to their low practicality even though they can eliminate I-cache side channel leakage. However, we presented two practical approaches, "partial flushing" and "cache conscious memory layout", that have very low performance overheads.

We realized that even very primitive support from the hardware can be very helpful towards designing and developing low-cost mitigations. For instance, if a processor's ISA includes an instruction permitting to flush cache sets, mitigations like partial flushing become much easier to implement and have lower performance overheads.

As our final remark, we want to emphasize that our results stress the significance of considering security as a dimension in processor design space and paying it the same level of attention as cost, performance, and power.

Acknowledgments The authors would like to thank Dan Page for his input throughout the duration of this work.

References

1. Http://cvs.openssl.org/chngview?cn=16275
2. Acıiçmez, O.: Yet another microarchitectural attack: Exploiting I-cache. In: Proceedings of the 1st ACM Workshop on Computer Security Architecture (CSAW 2007). pp. 11–18. ACM Press (2007)

3. Acıiçmez, O., Koç, Ç.K., Seifert, J.P.: On the power of simple branch prediction analysis. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS 2007). pp. 312–320. ACM Press (2007)
4. Acıiçmez, O., Schindler, W.: A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In: Topics in Cryptology — CT-RSA 2008. Lecture Notes in Computer Science, vol. 4964, pp. 256–273. Springer Verlag (2008)
5. Acıiçmez, O., Schindler, W., Ç. K. Koç: Cache based remote timing attacks on the AES. In: Topics in Cryptology — CT-RSA 2007. Lecture Notes in Computer Science, vol. 4377, pp. 271–286. Springer Verlag (2007)
6. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Advances in Cryptology — ASIACRYPT 2009. Lecture Notes in Computer Science, vol. 5912, pp. 667–684. Springer Verlag (2009)
7. Intel Corporation: Intel(R) 64 and IA-32 Architectures Software Developer's Manual. Available for download at http://developer.intel.com/Assets/PDF/manual/253667.pdf
8. Menezes, A., Vanstone, S., van Oorschot, P.: Handbook of Applied Cryptography. CRC Press, Inc. (1996)
9. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
10. Neve, M., Seifert, J.P.: Advances on access-driven cache attacks on AES. In: Selected Areas in Cryptography — SAC 2007. Lecture Notes in Computer Science, vol. 4356, pp. 147–162. Springer Verlag (2007)
11. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Topics in Cryptology — CT-RSA 2006. Lecture Notes in Computer Science, vol. 3860, pp. 1–20. Springer Verlag (2006)
12. Page, D.: Partitioned cache architecture as a side-channel defense mechanism. Cryptology ePrint Archive, Report 2005/280 (2005), available for download at http://eprint.iacr.org
13. Percival, C.: Cache missing for fun and profit. In: Proceedings of BSDCan 2005 (2005), available for download at http://www.daemonology.net/papers/htt.pdf
14. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009). pp. 199–212. ACM Press (2009)
15. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Secretly monopolizing the CPU without superuser privileges. In: Proceedings of the 16th USENIX Security Symposium (SECURITY 2007). pp. 239–256. USENIX Association (2007)
16. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side-channel attacks. In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA 2007). pp. 494–505. ACM Press (2007)