Malware Guard Extension: Using SGX to Conceal Cache Attacks

Michael Schwarz(✉), Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard

Graz University of Technology, Graz, Austria
michael.schwarz@iaik.tugraz.at

Abstract. In modern computer systems, user processes are isolated from each other by the operating system and the hardware. Additionally, in a cloud scenario it is crucial that the hypervisor isolates tenants from other tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against the cloud provider and thus the supplied operating system and hardware. Intel SGX provides a mechanism that addresses this scenario. It aims at protecting user-level software from attacks from other processes, the operating system, and even physical attackers.

In this paper, we demonstrate fine-grained software-based side-channel attacks from a malicious SGX enclave targeting co-located enclaves. Our attack is the first malware running on real SGX hardware, abusing SGX protection features to conceal itself. Furthermore, we demonstrate our attack both in a native environment and across multiple Docker containers. We perform a *Prime+Probe* cache side-channel attack on a co-located SGX enclave running an up-to-date RSA implementation that uses a constant-time multiplication primitive. The attack works although in SGX enclaves there are no timers, no large pages, no physical addresses, and no shared memory. In a semi-synchronous attack, we extract 96% of an RSA private key from a single trace. We extract the full RSA private key in an automated attack from 11 traces.

1 Introduction

Modern operating systems isolate user processes from each other to protect secrets in different processes. Such secrets include passwords stored in password managers or private keys to access company networks. Leakage of these secrets can compromise both private and corporate systems. Similar problems arise in the cloud. Therefore, cloud providers use virtualization as an additional protection using a hypervisor. The hypervisor isolates different tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against a possibly malicious cloud provider.

Although hypervisors provide functional isolation, side-channel attacks are often not considered. Consequently, researchers have demonstrated various side-channel attacks, especially those exploiting the cache [15]. Cache side-channel

M. Polychronakis and M. Meier (Eds.): DIMVA 2017, LNCS 10327, pp. 3–24, 2017.
DOI: 10.1007/978-3-319-60876-1_1

attacks can recover cryptographic secrets, such as AES [29] and RSA [33] keys, across virtual machine boundaries.

Intel introduced a new hardware extension SGX (Software Guard Extensions) [27] in their CPUs, starting with the Skylake microarchitecture. SGX is an isolation mechanism, aiming at protecting code and data from modification or disclosure even if all privileged software is malicious [10]. This protection uses special execution environments, so-called enclaves, which work on memory areas that are isolated from the operating system by the hardware. The memory area used by the enclaves is encrypted to protect application secrets from hardware attackers. Typical use cases include password input, password managers, and cryptographic operations. Intel recommends storing cryptographic keys inside enclaves and claims that side-channel attacks "are thwarted since the memory is protected by hardware encryption" [25].

Hardware-supported isolation also led to fear of super malware inside enclaves. Rutkowska [44] outlined a scenario where an enclave fetches encrypted malware from an external server and executes it within the enlave. In this scenario, it is impossible to debug, reverse engineer, or analyze the executed malware in any way. Costan et al. [10] eliminated this fear by arguing that enclaves always run with user space privileges and can neither issue syscalls nor perform any I/O operations. Moreover, SGX is a highly restrictive environment for implementing cache side-channel attacks. Both state-of-the-art malware and side-channel attacks rely on several primitives that are not available in SGX enclaves.

In this paper, we show that it is very well possible for enclave malware to attack its hosting system. We demonstrate a cross-enclave cache attack from within a malicious enclave that is extracting secret keys from co-located enclaves. Our proof-of-concept malware is able to recover RSA keys by monitoring cache access patterns of an RSA signature process in a semi-synchronous attack. The malware code is completely invisible to the operating system and cannot be analyzed due to the isolation provided by SGX. We present novel approaches to recover physical address bits, as well as to recover high-resolution timing in absence of the timestamp counter, which has an even higher resolution than the native one. In an even stronger attack scenario, we show that an additional isolation using Docker containers does not protect against this kind of attack.

We make the following contributions:

1. We demonstrate that, despite the restrictions of SGX, cache attacks can be performed from within an enclave to attack a co-located enclave.
2. By combining DRAM and cache side channels, we present a novel approach to recover physical address bits even if 2 MB pages are unavailable.
3. We obtain high-resolution timestamps in enclaves without access to the native timestamp counter, with an even higher resolution than the native one.
4. In an automated end-to-end attack on the wide-spread *mbedTLS* RSA implementation, we extract 96% of an RSA private key from a single trace.

Section 2 presents the required background. Section 3 outlines the threat model and attack scenario. Section 4 describes the measurement methods and the online phase of the malware. Section 5 explains the offline-phase key recovery. Section 6 evaluates the attack against an up-to-date RSA implementation. Section 7 discusses several countermeasures. Section 8 concludes our work.

2 Background

2.1 Intel SGX in Native and Virtualized Environments

Intel Software Guard Extensions (SGX) are a new set of x86 instructions introduced with the Skylake microarchitecture. SGX allows protecting the execution of user programs in so-called enclaves. Only the enclave can access its own memory region, any other access to it is blocked by the CPU. As SGX enforces this policy in hardware, enclaves do not need to rely on the security of the operating system. In fact, with SGX the operating system is generally not trusted. By doing sensitive computation inside an enclave, one can effectively protect against traditional malware, even if such malware has obtained kernel privileges. Furthermore, it allows running secret code in a cloud environment without trusting hardware and operating system of the cloud provider.

An enclave resides in the virtual memory area of an ordinary application process. This virtual memory region of the enclave can only be backed by physically protected pages from the so-called Enclave Page Cache (EPC). The EPC itself is a contiguous physical block of memory in DRAM that is encrypted transparently to protect against hardware attacks.

Loading of enclaves is done by the operating system. To protect the integrity of enclave code, the loading procedure is measured by the CPU. If the resulting measurement does not match the value specified by the enclave developer, the CPU will refuse to run the enclave.

Since enclave code is known to the (untrusted) operating system, it cannot carry hard-coded secrets. Before giving secrets to an enclave, a provisioning party has to ensure that the enclave has not been tampered with. SGX therefore provides remote attestation, which proves correct enclave loading via the aforementioned enclave measurement.

At the time of writing, no hypervisor with SGX support was available. However, Arnautov et al. [4] proposed to combine Docker containers with SGX to create secure containers. Docker is an operating-system-level virtualization software that allows applications to run in separate containers. It is a standard runtime for containers on Linux which is supported by multiple public cloud providers. Unlike virtual machines, Docker containers share the kernel and other resources with the host system, requiring fewer resources than a virtual machine.

2.2 Microarchitectural Attacks

Microarchitectural attacks exploit hardware properties that allow inferring information on other processes running on the same system. In particular, cache attacks exploit the timing difference between the CPU cache and the main memory. They have been the most studied microarchitectural attacks for the past 20 years, and were found to be powerful to derive cryptographic secrets [15]. Modern attacks target the last-level cache, which is shared among all CPU cores. Last-level caches (LLC) are usually built as n-way set-associative caches. They consist of S cache sets and each cache set consists of n cache ways with a size of 64 B. The lowest 6 physical address bits determine the byte offset within a cache way, the following $\log_2 S$ bits starting with bit 6 determine the cache set.

Prime+Probe is a cache attack technique that has first been used by Osvik et al. [39]. In a *Prime+Probe* attack, the attacker constantly primes (*i.e.*, evicts) a cache set and measures how long this step took. The runtime of the prime step is correlated to the number of cache ways that have been replaced by other programs. This allows deriving whether or not a victim application performed a specific secret-dependent memory access. Recent work has shown that this technique can even be used across virtual machine boundaries [33,35].

To prime (*i.e.*, evict) a cache set, the attacker uses n addresses in same cache set (*i.e.*, an *eviction set*), where n depends on the cache replacement policy and the number of ways. To minimize the amount of time the prime step takes, it is necessary to find a minimal n combined with a fast access pattern (*i.e.*, an *eviction strategy*). Gruss et al. [18] experimentally found efficient eviction strategies with high eviction rates and a small number of addresses. We use their eviction strategy on our Skylake test machine throughout the paper.

Pessl et al. [42] found a similar attack through DRAM modules. Each DRAM module has a row buffer that holds the most recently accessed DRAM row. While accesses to this buffer are fast, accesses to other memory locations in DRAM are much slower. This timing difference can be exploited to obtain fine-grained information across virtual machine boundaries.

2.3 Side-Channel Attacks on SGX

Intel claims that SGX features impair side-channel attacks and recommends using SGX enclaves to protect password managers and cryptographic keys against side channels [25]. However, there have been speculations that SGX could be vulnerable to side-channel attacks [10]. Xu et al. [50] showed that SGX is vulnerable to page fault side-channel attacks from a malicious operating system [1].

SGX enclaves generally do not share memory with other enclaves, the operating system or other processes. Thus, any attack requiring shared memory is not possible, e.g., *Flush+Reload* [51]. Also, DRAM-based attacks cannot be performed from a malicious operating system, as the hardware prevents any operating system accesses to DRAM rows in the EPC. However, enclaves can mount DRAM-based attacks on other enclaves because all enclaves are located in the same physical EPC.

In concurrent work, Brasser et al. [8], Moghimi et al. [37] and Götzfried et al. [17] demonstrated cache attacks on SGX relying on a malicious operating system.

2.4 Side-Channel Attacks on RSA

RSA is widely used to create asymmetric signatures, and is implemented by virtually every TLS library, such as OpenSSL or *mbedTLS*, which is used for instance in cURL and OpenVPN. RSA essentially involves modular exponentiation with a private key, typically using a square-and-multiply algorithm. An unprotected implementation of square-and-multiply is vulnerable to a variety of side-channel attacks, in which an attacker learns the exponent by distinguishing the square step from the multiplication step [15,51]. *mbedTLS* uses a windowed square-and-multiply routine for the exponentiation. Liu et al. [33] showed that if an attack on a window size of 1 is possible, the attack can be extended to arbitrary window sizes.

Earlier versions of *mbedTLS* were vulnerable to a timing side-channel attack on RSA-CRT [3]. Due to this attack, current versions of *mbedTLS* implement a constant-time Montgomery multiplication for RSA. Additionally, instead of using a dedicated square routine, the square operation is carried out using the multiplication routine. Thus, there is no leakage from a different square and multiplication routine as exploited in previous attacks on square-and-multiply algorithms [33,51]. However, Liu et al. [33] showed that the secret-dependent accesses to the buffer b still leak the exponent. Boneh et al. [7] and Blömer et al. [6] recovered the full RSA private key if only parts of the key bits are known.

3 Threat Model and Attack Setup

In this section, we present our threat model. We demonstrate a malware that circumvents SGX and Docker isolation guarantees. We successfully mount a *Prime+Probe* attack on an RSA signature computation running inside a different enclave, on the outside world, and across container boundaries.

3.1 High-Level View of the Attack

In our threat model, both the attacker and the victim are running on the same physical machine. The machine can either be a user's local computer or a host in the cloud. In the cloud scenario, the victim has its enclave running in a Docker container to provide services to other applications running on the host. Docker containers are well supported on many cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. As these containers are more lightweight than virtual machines, a host can run up to several hundred containers simultaneously. Thus, the attacker has good chances to get a co-located container on a cloud provider.

Figure 1 gives an overview of our native setup. The victim runs a cryptographic computation inside the enclave to protect it against any attacks. The attacker tries to stealthily extract secrets from this victim enclave. Both the attacker and the victim use Intel SGX features and thus are subdivided into two parts, the enclave and loader, *i.e.*, the main program instantiating the enclave.

The attack is a multi-step process that can be divided into an online and an offline phase. Section 4 describes the online phase, in which the attacker first

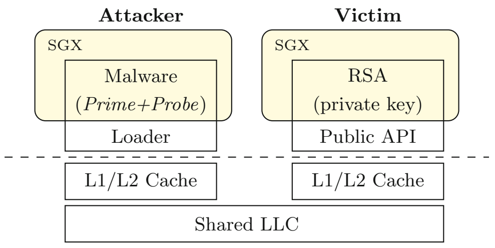


Fig. 1. The threat model: both attacker and victim run on the same physical machine in different SGX enclaves.

locates the victim's cache sets that contain the secret-dependent data of the RSA private key. The attacker then monitors the identified cache sets while triggering a signature computation. Section 5 gives a detailed explanation of the offline phase in which the attacker recovers a private key from collected traces.

3.2 Victim

The victim is an unprivileged program that uses SGX to protect an RSA signing application from both software and hardware attackers. Both the RSA implementation and the private key reside inside the enclave, as suggested by Intel [25]. Thus, they can never be accessed by system software or malware on the same host. Moreover, memory encryption prevents physical information leakage in DRAM. The victim uses the RSA implementation of the widely deployed *mbedTLS* library. The *mbedTLS* library implements a windowed square-and-multiply algorithm, that relies on constant-time Montgomery multiplications. The window size is fixed to 1, as suggested by the official knowledge base [2]. The victim application provides an API to compute a signature for provided data.

3.3 Attacker

The attacker runs an unprivileged program on the same host machine as the victim. The goal of the attacker is to stealthily extract the private key from the victim enclave. Therefore, the attacker uses the API provided by the victim to trigger signature computations.

The attacker targets the exponentiation step of the RSA implementation. The attack works on arbitrary window sizes [33], including window size 1. To prevent information leakage from function calls, *mbedTLS* uses the same function (`mpi_montmul`) for both the square and the multiply operation. The `mpi_montmul` takes two parameters that are multiplied together. For the square operation, the function is called with the current buffer as both arguments. For the multiply operation, the current buffer is multiplied with a buffer holding the multiplier. This buffer is allocated in the calling function `mbedtls_mpi_exp_mod` using `calloc`. Due to the deterministic behavior of the tlibc `calloc` implementation, the used buffers always have the same virtual and physical addresses and thus the same cache sets. The attacker can therefore mount a *Prime+Probe* attack on the cache sets containing the buffer.

In order to remain stealthy, all parts of the malware that contain attack code reside inside an SGX enclave. The enclave can protect the encrypted real attack code by only decrypting it after a successful remote attestation after which the enclave receives the decryption key. As pages in SGX can be mapped as writable and executable, self-modifying code is possible and therefore code can be encrypted. Consequently, the attack is completely stealthy and invisible from anti-virus software and even from monitoring software running in ring 0. Note that our proof-of-concept implementation does not encrypt the attack code as this has no impact on the attack.

The loader does not contain any suspicious code or data, it is only required to start the enclave and send the exfiltrated data to the attacker.

3.4 Operating System and Hardware

Previous work was mostly focused on attacks on enclaves from untrusted cloud operating systems [10,46]. However, in our attack we do not make any assumptions on the underlying operating system, *i.e.*, we do not rely on a malicious operating system. Both the attacker and the victim are unprivileged user space applications. Our attack works on a fully-patched recent operating system with no known software vulnerabilities, *i.e.*, the attacker cannot elevate privileges.

We expect the cloud provider to run state-of-the-art malware detection software. We assume that the malware detection software is able to monitor the behavior of containers and inspect the content of containers. Moreover, the user can run anti-virus software and monitor programs inside the container. We assume that the protection mechanisms are either signature-based, behavioral-based, heuristics-based or use performance counters [12,21].

Our only assumption on the hardware is that attacker and victim run on the same host system. This is the case on both personal computers and on co-located Docker instances in the cloud. As SGX is currently only available on Intel Skylake CPUs, it is valid to assume that the host is a Skylake system. Consequently, we know that the last-level cache is shared between all CPU cores.

4 Extracting Private Key Information

In this section, we describe the online phase of our attack. We first build primitives necessary to mount this attack. Then we show in two steps how to locate and monitor cache sets to extract private key information.

4.1 Attack Primitives in SGX

Successful *Prime+Probe* attacks require two primitives: a high-resolution timer to distinguish cache hits and misses and a method to generate an eviction set for arbitrary cache sets. Due to the restrictions of SGX enclaves, implementing *Prime+Probe* in enclaves is not straight-forward. Therefore, we require new techniques to build a malware from within an enclave.

High-Resolution Timer. The unprivileged `rdtsc` and `rdtscp` instructions, which read the timestamp counter, are usually used for fine-grained timing outside enclaves. In SGX, these instructions are not permitted inside an enclave, as they might cause a VM exit [24]. Thus, we have to rely on a different timing source with a resolution in the order of 10 cycles to reliably distinguish cache hits from misses as well as DRAM row hits from row conflicts.

To achieve the highest number of increments, we handcraft a counter thread [31,49] in inline assembly. The counter variable has to be accessible across threads, thus it is necessary to store the counter variable in memory. Memory addresses as operands incur an additional cost of approximately 4 cycles due to L1 cache access times [23]. On our test machine, a simple counting thread executing `1: incl (%rcx); jmp 1b` achieves one increment every 4.7 cycles, which is an improvement of approximately 2% over the best code generated by `gcc`.

We can improve the performance—and thus the resolution—further, by exploiting the fact that only the counting thread modifies the counter variable. We can omit reading the counter variable from memory. Therefore, we introduce a "shadow counter variable" which is always held in a CPU register. The arithmetic operation (either `add` or `inc`) is performed on this register, unleashing the low latency and throughput of these instructions. As registers cannot be shared across threads, the shadow counter has to be moved to memory using the `mov` instruction after each increment. Similar to the `inc` and `add` instruction, the `mov` instruction has a latency of 1 cycle and a throughput of 0.5 cycles/instruction when copying a register to memory. The improved counting thread, `1: inc %rax; mov %rax, (%rcx), jmp 1b`, is significantly faster and increments the variable by one every 0.87 cycles, which is an improvement of 440% over the simple counting thread. In fact, this version is even 15% faster than the native timestamp counter, thus giving us a reliable timing source with even higher resolution. This new method might open new possibilities of side-channel attacks that leak information through timing on a sub-`rdtsc` level.

Eviction Set Generation. *Prime+Probe* relies on eviction sets, *i.e.*, we need to find virtual addresses that map to the same physical cache set. An unprivileged process cannot translate virtual to physical addresses and therefore cannot simply search for virtual addresses that fall into the same cache set. Liu et al. [33] and Maurice et al. [35] demonstrated algorithms to build eviction sets using large pages by exploiting the fact that the virtual address and the physical address have the same lowest 21 bits. As SGX does not support large pages, this approach is inapplicable. Oren et al. [38] and Gruss et al. [18] demonstrated automated methods to generate eviction sets for a given virtual address. Due to microarchitectural changes their approaches are either not applicable at all to the Skylake architecture or consume several hours on average before even starting the actual *Prime+Probe* attack.

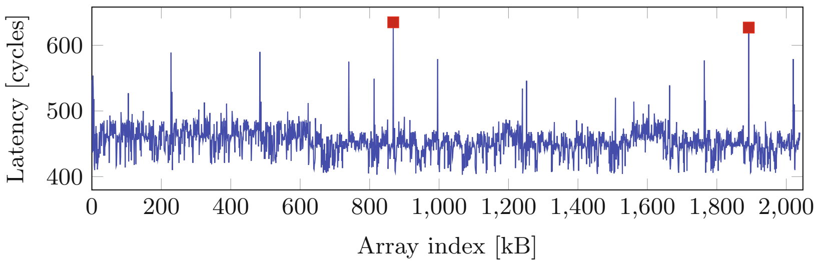


Fig. 2. Access times when alternately accessing two addresses which are 64 B apart. The (marked) high access times indicate row conflicts.

We propose a new method to recover the cache set from a virtual address without relying on large pages. The idea is to exploit contiguous page allocation [28] and DRAM timing differences to recover DRAM row boundaries. The DRAM mapping functions [42] allow to recover physical address bits.

The DRAM organization into banks and rows causes timing differences. Alternately accessing pairs of two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of virtual addresses. Figure 2 shows the average access time for address pairs when iterating over a 2 MB array. The highest two peaks show row conflicts, *i.e.*, the row index changes while the bank, rank, and channel stay the same.

Table 1. Reverse-engineered DRAM mapping functions from Pessl et al. [42].

		Address Bit																
		22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 DIMMs	Channel				⊕	⊕					⊕	⊕	⊕	⊕	⊕	⊕		⊕
	BG0									⊕							⊕	
	BG1	⊕				⊕												
	BA0				⊕				⊕									
	BA1		⊕				⊕											
	Rank			⊕				⊕										

To recover physical address bits we use the reverse-engineered DRAM mapping function as shown in Table 1. Our test machine is an Intel Core i5-6200U with 12 GB main memory. The row index is determined by physical address bits 18 and upwards. Hence, the first address of a DRAM row has the least-significant 18 bits of the physical address set to '0'. To detect row borders, we scan memory sequentially for an address pair in physical proximity that causes a *row conflict*. As SGX enclave memory is allocated contiguously we can perform this scan on virtual addresses.

A virtual address pair that causes row conflicts at the beginning of a row satisfies the following constraints:

1. The least-significant 18 physical address bits of one virtual address are zero. This constitutes a DRAM row border.
2. The bank address (BA), bank group (BG), rank, and channel determine the DRAM bank and must be the same for both virtual addresses.
3. The row index must be different for both addresses to cause a row conflict.
4. The difference of the two virtual addresses has to be at least 64 B (the size of one cache line) but should not exceed 4 kB (the size of one page).

Physical address bits 6 to 17 determine the cache set which we want to recover. Hence, we search for address pairs where physical address bits 6 to 17 have the same known but arbitrary value.

To find address pairs fulfilling the aforementioned constraints, we modeled the mapping function and the constraints as an SMT problem and used the Z3 theorem prover [11] to provide models satisfying the constraints. The model we found yields pairs of physical addresses where the upper address is 64 B apart from the lower one. There are four such address pairs within every 4 MB block of physical memory such that each pair maps to the same bank but a different row. The least-significant bits of the physical address pairs are either (`0x3fffc0`, `0x400000`), (`0x7fffc0`, `0x800000`), (`0xbfffc0`, `0xc00000`) or (`0xffffc0`, `0x1000000`) for the lower and higher address respectively. Thus, at least 22 bits of the higher addresses least-significant bits are 0. As the cache set is determined by the bits 6 to 17, the higher address has the cache set index 0. We observe that satisfying address pairs are always 256 KB apart. Since we have contiguous memory [28], we can generate addresses mapping to the same cache set by adding multiples of 256 KB to the higher address.

In modern CPUs, the last-level cache is split into cache slices. Addresses with the same cache set index map to different cache slices based on the remaining address bits. To generate an eviction set, it is necessary to only use addresses that map to the same cache set in the same cache slice. However, to calculate the cache slice, all bits of the physical address are required [34].

As we are not able to directly calculate the cache slice, we use another approach. We add our calculated addresses from the correct cache set to our eviction set until the eviction rate is sufficiently high. Then, we try to remove single addresses from the eviction set as long as the eviction rate does not drop. Thus, we remove all addresses that do not contribute to the eviction, and the result is a minimal eviction set. Our approach takes on average 2 s per cache set, as we already know that our addresses map to the correct cache set. This is nearly three orders of magnitude faster than the approach of Gruss et al. [18]. Older techniques that have been comparably fast do not work on current hardware anymore due to microarchitectural changes [33,38].

4.2 Identifying and Monitoring Vulnerable Sets

With the reliable high-resolution timer and a method to generate eviction sets, we can mount the first stage of the attack and identify the vulnerable cache sets. As we do not have any information about the physical addresses of the victim, we have to scan the last-level cache for characteristic patterns corresponding to the signature process. We consecutively mount a *Prime+Probe* attack on every cache set while the victim is executing the exponentiation step.

We can then identify multiple cache sets showing the distinctive pattern of the signature operation. The number of cache sets depends on the RSA key size. Cache sets at the buffer boundaries might be used by neighboring buffers and are more likely to be prefetched [20,51] and thus, prone to measurement errors. Consequently, we use cache sets neither at the start nor the end of the buffer.

The measurement method is the same as for detecting the vulnerable cache sets, *i.e.*, we again use *Prime+Probe*. Due to the deterministic behavior of the heap allocation, the address of the attacked buffer does not change on consecutive exponentiations. Thus, we can collect multiple traces of the signature process.

To maintain a high sampling rate, we keep the post-processing during the measurements to a minimum. Moreover, it is important to keep the memory activity at a minimum to not introduce additional noise on the cache. Thus, we only save the timestamps of the cache misses for further post-processing. As a cache miss takes longer than a cache hit, the effective sampling rate varies depending on the number of cache misses. We have to consider this effect in the post-processing as it induces a non-constant sampling interval.

5 Recovering the Private Key

In this section, we describe the offline phase of our attack: recovering the private key from the recorded traces of the victim enclave. This can either be done inside the malware enclave or on the attacker's server.

Ideally, an attacker would combine multiple traces by aligning them and averaging out noise. From the averaged trace, the private key can be extracted more easily. However, most noise sources, such as context switches, system activity and varying CPU clock, alter the timing, thus making trace alignment difficult. We pre-process all traces individually and extract a partial key out of each trace. These partial keys likely suffer from random insertion and deletion errors as well as from bit flips. To eliminate the errors, we combine multiple partial keys in the key recovery phase. This approach has much lower computational overhead than trace alignment since key recovery is performed on partial 4096-bit keys instead of full traces containing several thousand measurements.

Key recovery comes in three steps. First, traces are pre-processed. Second, a partial key is extracted from each trace. Third, the partial keys are merged to recover the private key. In the pre-processing step we filter and resample raw measurement data. Figure 3 shows a trace segment before (top) and after pre-processing (bottom). The pre-processed trace shows high peaks at locations of cache misses, indicating a '1' in the RSA exponent.

To automatically extract a partial key from a pre-processed trace, we first run a peak detection algorithm. We delete duplicate peaks, e.g., peaks where the corresponding RSA multiplications would overlap in time. We also delete peaks that are below a certain adaptive threshold, as they do not correspond to actual

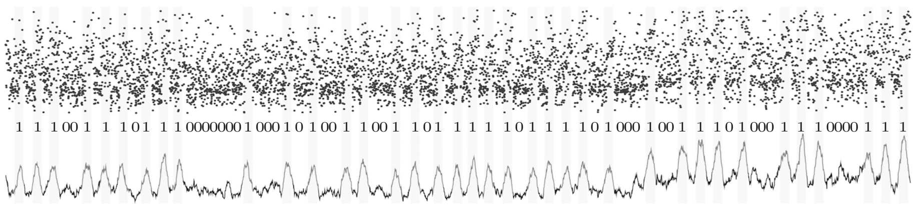


Fig. 3. A raw measurement trace over 4000000 cycles. The peaks in the pre-processed trace on the bottom clearly indicate ‘1’s.

multiplications. Using an adaptive threshold is necessary since neither the CPU frequency nor our timing source (the counting thread) is perfectly stable. The varying peak height is shown in the right third of Fig. 3. The adaptive threshold is the median over the 10 previously detected peaks. If a peak drops below 90% of this threshold, it is discarded. The remaining peaks correspond to the ‘1’s in the RSA exponent and are highlighted in Fig. 3. ‘0’s can only be observed indirectly in our trace as square operations do not trigger cache activity on the monitored sets. ‘0’s appear as time gaps in the sequence of ‘1’ peaks, thus revealing all partial key bits. Note that since ‘0’s correspond to just one multiplication, they are roughly twice as fast as ‘1’s.

When a correct peak is falsely discarded, the corresponding ‘1’ is interpreted as two ‘0’s. Likewise, if noise is falsely interpreted as a ‘1’, this cancels out two ‘0’s. If either the attacker or the victim is not scheduled, we have a gap in the collected trace. However, if both the attacker and the victim are descheduled, this gap does not show up prominently in the trace since the counting thread is also suspended by the interrupt. This is an advantage of a counting thread over the use of the native timestamp counter.

In the final key recovery, we merge multiple partial keys to obtain the full key. We quantify partial key errors using the edit distance. The edit distance between a partial key and the correct key gives the number of bit insertions, deletions and flips necessary to transform the partial key into the correct key.

The full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. To correct the current bit of a wrong partial key, we compute the edit distance to all partial keys that won the majority vote. To reduce the performance overhead, we do not calculate the edit distance over the whole partial keys but only over a lookahead window of a few bits. The output of the edit distance algorithm is a list of actions necessary to transform one key into the other. We apply these actions via majority vote until the key bit of the wrong partial key matches the recovered key bit again.

6 Evaluation

In this section, we evaluate the presented methods by building a malware enclave attacking a co-located enclave that acts as the victim. As discussed in Sect. 3.2, we use *mbedTLS*, in version 2.3.0.

For the evaluation, we attack a 4096-bit RSA key. The runtime of the multiplication function increases exponentially with the size of the key. Hence, larger keys improve the measurement resolution of the attacker. In terms of cache side-channel attacks, large RSA keys do not provide higher security but degrade side-channel resistance [41,48,51].

6.1 Native Environment

We use a Lenovo ThinkPad T460s with an Intel Core i5-6200U (2 cores, 12 cache ways) running Ubuntu 16.10 and the Intel SGX driver. Both the attacker enclave and the victim enclave are running on the same machine. We trigger the signature process using the public API of the victim.

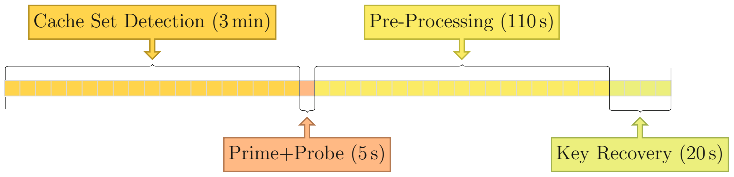


Fig. 4. A high-level overview of the average times for each step of the attack.

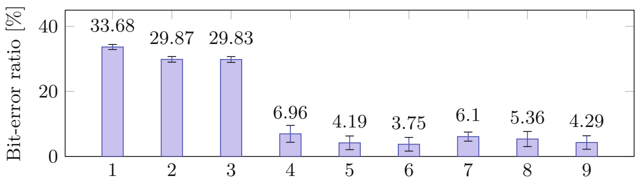
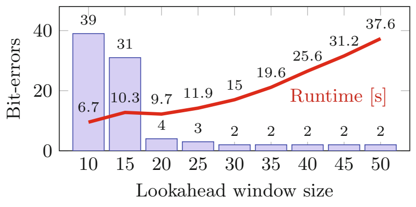


Fig. 5. The 9 cache sets that are used by a 4096-bit key and their error ratio when recovering the key from a single trace.

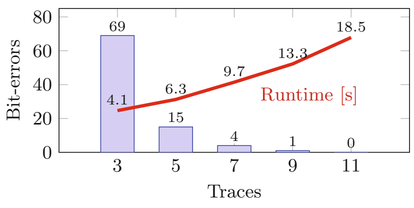
Figure 4 gives an overview of how long the individual steps of an average attack take. The runtime of automatic cache set detection varies depending on which cache sets are used by the victim. The attacked buffer spans 9 cache sets, out of which 6 show a low bit-error ratio, as shown in Fig. 5. For the attack we select one of the 6 sets, as the other 3 suffer from too much noise. The noise is mainly due to the buffer not being aligned to the cache set. Furthermore, as already known from previous attacks, the hardware prefetcher can induce a significant amount of noise [20,51].

Detecting one vulnerable cache set within all 2048 cache sets requires about 340 trials on average. With a monitoring time of 0.21 s per cache set, we require a maximum of 72 s to eventually capture a trace from a vulnerable cache set. Thus, based on our experiments, we estimate that cache set detection—if successful—always takes less than 3 min.

One trace spans 220.47 million CPU cycles on average. Typically, '0' and '1' bits are uniformly distributed in the key. The estimated number of multiplications is therefore half the bit size of the key. Thus, the average multiplication takes 107662 cycles. As the *Prime+Probe* measurement takes on average 734 cycles, we do not have to slow down the victim additionally.



(a) Increasing the lookahead reduces bit errors and increases runtime.



(b) Increasing the number of traces reduces bit errors and increases runtime.

Fig. 6. Relation between number of traces, lookahead window size, number of bit errors, and runtime.

When looking at a single trace, we can already recover about 96% of the RSA private key, as shown in Fig. 5. For a full key recovery we combine multiple traces using our key recovery algorithm, as explained in Sect. 5. We first determine a reasonable lookahead window size. Figure 6a shows the performance of our key recovery algorithm for varying lookahead window sizes on 7 traces. For lookahead windows smaller than 20, bit errors are pretty high. In that case, the lookahead window is too small to account for all insertion and deletion errors, causing relative shifts between the partial keys. The key recovery algorithm is unable to align partial keys correctly and incurs many wrong "correction" steps, increasing the overall runtime as compared to a window size of 20. While a lookahead window size of 20 already shows a good performance, a window size of 30 or more does not significantly reduce the bit errors. Therefore, we fixed the lookahead window size to 20.

To remove the remaining bit errors and get full key recovery, we have to combine more traces. Figure 6b shows how the number of traces affects the key recovery performance. We can recover the full RSA private key without any bit errors by combining only 11 traces within just 18.5 s. This results in a total runtime of less than 130 s for the offline key recovery process.

Generalization. Based on our experiments we deduced that attacks are also possible in a weaker scenario, where only the attacker is inside the enclave. On most computers, applications handling cryptographic keys are not protected by SGX enclaves. From the attacker's perspective, attacking such an unprotected application does not differ from attacking an enclave. We only rely on the last-level cache, which is shared among all applications, whether they run inside an enclave or not. We empirically verified that such attacks on the outside world are possible and were again able to recover RSA private keys.

Table 2 summarizes our results. In contrast to concurrent work on cache attacks on SGX [8,17,37], our attack is the only one that can be mounted from unprivileged user space, and cannot be detected as it runs within an enclave.

Table 2. Our results show that cache attacks can be mounted successfully in the shown scenarios.

Attack from	Attack on		
	Benign userspace	Benign kernel	Benign SGX enclave
Malicious userspace	✓ [33,39]	✓ [22]	✓ **new**
Malicious kernel	—	—	✓ **new** [8,17,37]
Malicious SGX enclave	✓ **new**	✓ **new**	✓ **new**

6.2 Virtualized Environment

We now show that the attack also works in a virtualized environment. As described in Sect. 2.1, no hypervisor with SGX support was available at the time of our experiments. Instead of full virtualization using a virtual machine, we used lightweight Docker containers, as used by large cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. To enable SGX within a container, the host operating system has to provide SGX support. The SGX driver is then simply shared among all containers. Figure 7 shows our setup where the SGX enclaves communicate directly with the SGX driver of the host operating system. Applications running inside the container do not experience any difference to running on a native system.

Considering the performance within Docker, only I/O operations and network access have a measurable overhead [14]. Operations that only depend on memory and CPU do not see any performance penalty, as these operations are not virtualized. Thus, caches are also not affected by the container.

We were successfully able to attack a victim from within a Docker container without any changes in the malware. We can even perform a cross-container attack, *i.e.*, both the malware and the victim are running inside different containers, without any changes. As expected, we require the same number of traces for a full key recovery. Hence, containers do not provide additional protection against our malware at all.

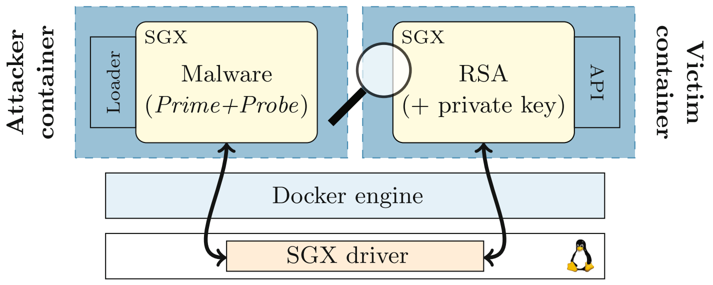


Fig. 7. Running the SGX enclaves inside Docker containers to provide further isolation. The host provides both containers access to the same SGX driver.

7 Countermeasures

Most existing countermeasures cannot be applied to a scenario where a malicious enclave performs a cache attack and no assumptions about the operating system are made. In this section, we discuss 3 categories of countermeasures, based on where they ought to be implemented.

7.1 Source Level

A generic side-channel protection for cryptographic operations (e.g., RSA) is exponent blinding [30]. It will prevent the proposed attack, but other parts of the signature process might still be vulnerable to an attack [45]. More generally bit slicing can be applied to a wider range of algorithms to protect against timing side channels [5,47]

7.2 Operating System Level

Implementing countermeasures against malicious enclave attacks on the operating system level requires trusting the operating system. This would weaken the trust model of SGX enclaves significantly, but in some threat models this can be a viable solution. However, we want to discuss the different possibilities, in order to provide valuable information for the design process of future enclave systems.

Detecting Malware. One of the core ideas of SGX is to remove the cloud provider from the root of trust. If the enclave is encrypted and only decrypted after successful remote attestation, the cloud provider has no way to access the secret code inside the enclave. Also, heuristic methods, such as behavior-based detection, are not applicable, as the malicious enclave does not rely on malicious API calls or user interaction which could be monitored. However, eliminating this core feature of SGX could mitigate malicious enclaves in practice, as the enclave binary or source code could be read by the cloud provider and scanned for malicious activities.

Herath and Fogh [21] proposed to use hardware performance counters to detect cache attacks. Subsequently, several other approaches instrumenting performance counters to detect cache attacks have been proposed [9,19,40]. However, according to Intel, SGX enclave activity is not visible in the thread-specific performance counters [26]. We verified that even performance counters for last-level cache accesses are disabled for enclaves. The performance counter values are three orders of magnitude below the values as compared to native code. There are no cache hits and misses visible to the operating system or any application (including the host application). This makes it impossible for current anti-virus software and other detection mechanisms to detect malware inside the enclave.

Enclave Coloring. We propose enclave coloring as an effective countermeasure against cross-enclave attacks. Enclave coloring is a software approach to partition the cache into multiple smaller domains. Each domain spans over multiple cache sets, and no cache set is included in more than one domain. An enclave gets one or more cache domains assigned exclusively. The assignment of domains is either done by the hardware or by the operating system. Trusting the operating system contradicts one of the core ideas of SGX [10]. However, if the operating system is trusted, this is an effective countermeasure against cross-enclave cache attacks.

If implemented in software, the operating system can split the last-level cache through memory allocation. The cache set index is determined by physical address bits below bit 12 (the page offset) and bits >12 which are not visible to the enclave application and can thus be controlled by the operating system. We call these upper bits a color. Whenever an enclave requests pages from the operating system (we consider the SGX driver as part of the operating system), it will only get pages with a color that is not present in any other enclave. This coloring ensures that two enclaves cannot have data in the same cache set, and therefore a *Prime+Probe* attack is not possible across enclaves. However, attacks on the operating system or other processes on the same host would still be possible.

To prevent attacks on the operating system or other processes, it would be necessary to partition the rest of the memory as well, *i.e.*, system-wide cache coloring [43]. Godfrey et al. [16] evaluated a coloring method for hypervisors by assigning every virtual machine a partition of the cache. They concluded that this method is only feasible for a small number of partitions. As the number of simultaneous enclaves is relatively limited by the available amount of SGX memory, enclave coloring can be applied to prevent cross-enclave attacks. Protecting enclaves from malicious applications or preventing malware inside enclaves is however not feasible using this method.

Heap Randomization. Our attack relies on the fact, that the used buffers for the multiplication are always at the same memory location. This is the case, as the used memory allocator (`dlmalloc`) has a deterministic best-fit strategy for moderate buffer sizes as used in RSA. Freeing a buffer and allocating it again will result in the same memory location for the re-allocated buffer.

We suggest randomizing the heap allocations for security relevant data such as the used buffers. A randomization of the addresses and thus cache sets bears two advantages. First, automatic cache set detection is not possible anymore, as the identified set will change for every run of the algorithm. Second, if more than one trace is required to reconstruct the key, heap randomization increases the number of required traces by multiple orders of magnitude, as the probability to measure the correct cache set by chance decreases.

Although not obvious at first glance, this method requires a certain amount of trust in the operating system. A malicious operating system could assign only pages mapping to certain cache sets to the enclave, similar to enclave coloring. Thus, the randomization is limited to only a subset of cache sets, increasing the probability for an attacker to measure the correct cache set.

Intel CAT. Recently, Intel introduced an instruction set extension called CAT (cache allocation technology) [24]. With Intel CAT it is possible to restrict CPU cores to one of the slices of the last-level cache and even to pin cache lines. Liu et al. [32] proposed a system that uses CAT to protect general purpose software and cryptographic algorithms. Their approach can be directly applied to protect against a malicious enclave. However, this approach does not allow to protect enclaves from an outside attacker.

7.3 Hardware Level

Combining Intel CAT with SGX. Instead of using Intel CAT on the operating system level it could also be used to protect enclaves on the hardware level. By changing the `eenter` instruction in a way that it implicitly activates CAT for this core, any cache sharing between SGX enclaves and the outside as well as co-located enclaves could be eliminated. Thus, SGX enclaves would be protected from outside attackers. Furthermore, it would protect co-located enclaves as well as the operating system and user programs against malicious enclaves.

Secure RAM. To fully mitigate cache- or DRAM-based side-channel attacks memory must not be shared among processes. We propose an additional fast, non-cachable secure memory element that resides inside the CPU.

The SGX driver can then provide an API to acquire the element for temporarily storing sensitive data. A cryptographic library could use this memory to execute code which depends on secret keys such as the square-and-multiply algorithm. Providing such a secure memory element per CPU core would even allow parallel execution of multiple enclaves.

Data from this element is only accessible by one program, thus cache attacks and DRAM-based attacks are not possible anymore. Moreover, if this secure memory is inside the CPU, it is infeasible for an attacker to mount physical attacks. It is unclear whether the Intel eDRAM implementation can already be instrumented as a secure memory to protect applications against cache attacks.

8 Conclusion

Intel claimed that SGX features impair side-channel attacks and recommends using SGX enclaves to protect cryptographic computations. Intel also claimed that enclaves cannot perform harmful operations.

In this paper, we demonstrated the first malware running in real SGX hardware enclaves. We demonstrated cross-enclave private key theft in an automated semi-synchronous end-to-end attack, despite all restrictions of SGX, e.g., no timers, no large pages, no physical addresses, and no shared memory. We developed a timing measurement technique with the highest resolution currently known for Intel CPUs, perfectly tailored to the hardware. We combined DRAM and cache side channels, to build a novel approach that recovers physical address bits without assumptions on the page size. We attack the RSA implementation of *mbedTLS*, which uses constant-time multiplication primitives. We extract 96% of a 4096-bit RSA key from a single *Prime+Probe* trace and achieve full key recovery from only 11 traces.

Besides not fully preventing malicious enclaves, SGX provides protection features to conceal attack code. Even the most advanced detection mechanisms using performance counters cannot detect our malware. This unavoidably provides attackers with the ability to hide attacks as it eliminates the only known technique to detect cache side-channel attacks. We discussed multiple design issues in SGX and proposed countermeasures for future SGX versions.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

1. Anati, I., McKeen, F., Gueron, S., Huang, H., Johnson, S., Leslie-Hurd, R., Patil, H., Rozas, C.V., Shafi, H.: Intel Software Guard Extensions (Intel SGX) (2015). Tutorial Slides presented at ICSA 2015
2. ARMmbed: Reduce mbed TLS memory and storage footprint, February 2016. https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint. Accessed 24 Oct 2016
3. Arnaud, C., Fouque, P.-A.: Timing attack against protected RSA-CRT implementation used in PolarSSL. In: Dawson, E. (ed.) CT-RSA 2013. LNCS, vol. 7779, pp. 18–33. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36095-4_2
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., et al.: SCONE: secure Linux containers with Intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016) (2016)

5. Biham, E.: A fast new DES implementation in software. In: International Workshop on Fast Software Encryption, pp. 260–272 (1997)
6. Blömer, J., May, A.: New partial key exposure attacks on RSA. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 27–43. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45146-4_2
7. Boneh, D., Durfee, G., Frankel, Y.: An attack on RSA given a small fraction of the private key bits. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 25–34. Springer, Heidelberg (1998). doi:10.1007/3-540-49649-1_3
8. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.: Software grand exposure: SGX cache attacks are practical (2017). http://arxiv.org/abs/1702.07521
9. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034 (2015)
10. Costan, V., Devadas, S.: Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086 (2016)
11. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
12. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. ACM SIGARCH Comput. Archit. News **41**(3), 559–570 (2013)
13. Docker: Amazon web services - docker (2016). https://docs.docker.com/machine/drivers/aws/
14. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS) (2015)
15. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Technical report, Cryptology ePrint Archive, Report 2016/613 (2016)
16. Godfrey, M.M., Zulkernine, M.: Preventing cache-based side-channel attacks in a cloud environment. IEEE Trans. Cloud Comput. **2**(4), 395–408 (2014)
17. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on Intel SGX. In: Proceedings of the 10th European Workshop on Systems Security (EuroSec 2017) (2017)
18. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in JavaScript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 300–321. Springer, Cham (2016). doi:10.1007/978-3-319-40667-1_15
19. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: a fast and stealthy cache attack. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 279–299. Springer, Cham (2016). doi:10.1007/978-3-319-40667-1_14
20. Gruss, D., Spreitzer, R., Mangard, S.: Cache template attacks: automating attacks on inclusive last-level caches. In: USENIX Security Symposium (2015)
21. Herath, N., Fogh, A.: These are not your grand Daddys CPU performance counters - CPU hardware performance counters for security. In: Black Hat USA (2015)
22. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: S&P 2013 (2013)
23. Intel: Intel® 64 and IA-32 Architectures Optimization Reference Manual (2014)

24. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide 253665 (2014)
25. Intel Corporation: Hardening Password Managers with Intel Software Guard Extensions: White Paper (2016)
26. Intel Corporation: Intel SGX: Debug, Production, Pre-release what's the difference? https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference. Accessed 24 Oct 2016
27. Intel Corporation: Intel Software Guard Extensions (Intel SGX) (2016). https://software.intel.com/en-us/sgx. Accessed 7 Nov 2016
28. Intel Corporation: Intel(R) Software Guard Extensions for Linux* OS (2016). https://github.com/01org/linux-sgx-driver. Accessed 11 Nov 2016
29. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! a fast, cross-VM attack on AES. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 299–319. Springer, Cham (2014). doi:10.1007/978-3-319-11379-1_15
30. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). doi:10.1007/3-540-68697-5_9
31. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: cache attacks on mobile devices. In: USENIX Security Symposium (2016)
32. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: defeating last-level cache side channel attacks in cloud computing. In: IEEE International Symposium on High Performance Computer Architecture (HPCA 2016) (2016)
33. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: S&P 2015 (2015)
34. Maurice, C., Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse engineering intel last-level cache complex addressing using performance counters. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 48–65. Springer, Cham (2015). doi:10.1007/978-3-319-26362-5_3
35. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the other side: SSH over robust cache covert channels in the cloud. In: NDSS 2017 (2017)
36. Microsoft: Create a Docker environment in azure using the docker VM extension, October 2016. https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-dockerextension/
37. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: how SGX amplifies the power of cache attacks. arXiv preprint arXiv:1703.06986 (2017)
38. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: practical cache attacks in JavaScript and their implications. In: CCS 2015 (2015)
39. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: CT-RSA 2006 (2006)
40. Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS 2016 (2016)
41. Pereida García, C., Brumley, B.B., Yarom, Y.: Make sure DSA signing exponentiations really are constant-time. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016)
42. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: exploiting DRAM addressing for Cross-CPU attacks. In: USENIX Security Symposium (2016)
43. Raj, H., Nathuji, R., Singh, A., England, P.: Resource management for isolation enhanced cloud services. In: Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW 2009), pp. 77–84 (2009)

44. Rutkowska, J.: Thoughts on Intel's upcoming Software Guard Extensions (Part 2) (2013). http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-on-intels-upcoming-software.html. Accessed 20 Oct 2016
45. Schindler, W.: Exclusive exponent blinding may not suffice to prevent timing attacks on RSA. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 229–247. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48324-4_12
46. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: trustworthy data analytics in the cloud using SGX (2015)
47. Sudhakar, M., Kamala, R.V., Srinivas, M.: A bit-sliced, scalable and unified montgomery multiplier architecture for RSA and ECC. In: 2007 IFIP International Conference on Very Large Scale Integration, pp. 252–257 (2007)
48. Matsui, M., Zuccherato, R.J. (eds.): SAC 2003. LNCS, vol. 3006. Springer, Heidelberg (2004)
49. Wray, J.C.: An analysis of covert timing channels. J. Comput. Secur. **1**(3–4), 219–232 (1992)
50. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: deterministic side channels for untrusted operating systems. In: S&P 2015, May 2015
51. Yarom, Y., Falkner, K.: Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium (2014)