Malware Guard Extension: abusing Intel SGX to conceal cache attacks

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, Stefan Mangard

▶ **To cite this version:**

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. Cybersecurity, 2020, 3 (1), 10.1186/s42400-019-0042-y . hal-02866628

HAL Id: hal-02866628
https://inria.hal.science/hal-02866628

Submitted on 12 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Schwarz *et al. Cybersecurity* (2020) 3:2
https://doi.org/10.1186/s42400-019-0042-y

Cybersecurity

RESEARCH Open Access

Malware Guard Extension: abusing Intel SGX to conceal cache attacks

Michael Schwarz[1*], Samuel Weiser[1], Daniel Gruss[1], Clémentine Maurice[2] and Stefan Mangard[1]

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 within 5 min.

Keywords: Intel SGX, Side channel, Side-channel attack, Prime+Probe

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 (Ge et al. 2016). Cache side-channel attacks can recover cryptographic secrets, such as AES (Irazoqui et al. 2014; Gülmezoğlu et al. 2015) and RSA (Inci et al. 2015) keys, across virtual machine boundaries.

Intel introduced a new hardware extension SGX (Software Guard Extensions) (Intel Corporation 2016a) 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 (Costan and Devadas 2016). 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 the application's 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" (Intel Corporation 2016b).

Apart from protecting software, the hardware-supported isolation led to fear of super malware inside enclaves. Rutkowska (2013) outlined a scenario where a benign-looking enclave fetches encrypted malware

*Correspondence: michael.schwarz@iaik.tugraz.at
[1]Graz University of Technology, Graz, Austria
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2020 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

from an external server and decrypts and executes it within the enclave. In this scenario, it is impossible to debug, reverse engineer, or in any other way analyze the executed malware. Aumasson et al. (2016) and Costan et al. (2016) 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. Consequently, no enclave malware has been demonstrated on real hardware so far.

In this paper, we show that it is very well possible for enclave malware to attack its hosting system. We demonstrate a cache attack from within a malicious enclave that is extracting secret keys from co-located enclaves. Our proof-of-concept malware can recover RSA keys by monitoring cache access patterns of an RSA signature process in a semi-synchronous attack. The malware code is entirely invisible to the operating system and cannot be analyzed due to the isolation provided by SGX. In order to build our attack, we present novel approaches to recover physical address bits, as well as to recover highly accurate timing in the absence of the timestamp counter, which is even more accurate 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 2MB pages are unavailable.
3. We show that it is possible to have highly accurate timings within an enclave without access to the native timestamp counter, which is even more accurate than the native one.
4. We demonstrate a fully automated end-to-end attack on the RSA implementation of the wide-spread *mbedTLS* library. We extract 96% of an RSA private key from a single trace and the full key from 11 traces within 5 min.

"Background" section presents the background required for our work. "Threat model and attack setup" section outlines the threat model and our attack scenario. "Extracting private key information" section describes the measurement methods and the online phase of the malware. "Recovering the private key" section explains the key recovery techniques used in the offline phase. "Evaluation" section evaluates the attack against an up-to-date RSA implementation. "Countermeasures" section discusses several countermeasures. "Conclusion" section concludes our work.

Background

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 the cloud provider's hardware and operating system.

An enclave resides in the virtual memory area of an ordinary application process. When creating an enclave, a virtual memory region is reserved for the enclave. This virtual memory region can only be backed by physically protected pages from the so-called Enclave Page Cache (EPC). In SGX, the operating system is in charge of mapping EPC pages correctly. However, any invalid or malicious page mapping is detected by the CPU to maintain enclave protection. The EPC itself is a contiguous physical block of memory in DRAM that is transparently encrypted using a dedicated hardware encryption module. This protects enclaves against hardware attacks trying to read or manipulate enclave content in DRAM.

Creation and loading of enclaves are done by the operating system. To protect the integrity of the 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. During enclave loading, the operating system has full access to the enclave binary. At this point, anti-virus scanners can hook in to analyze the enclave binary before it is executed. Enclave malware will attempt to hide from anti-virus scanners by encrypting the malicious payload.

Since enclave code is known to the (untrusted) operating system, it cannot carry hard-coded secrets. Any secret information might only be provisioned to the enclave during runtime. 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.

SGX comes in two versions. SGX1 specifies basic enclave operation. Moreover, all enclave memory pages have to be allocated at enclave creation. To account for

limited memory resources, enclave pages can be swapped out and in at runtime. SGX2 extends SGX with dynamic memory management, allowing to allocate new enclave pages at runtime. However, we do not use SGX2 features and thus presume that our attack applies to SGX2 as well.

At the time of writing, no hypervisor with SGX support was available to us. While there is an experimental version of KVM with SGX support (Intel Corporation 2016c), this is still under development and not readily usable. Moreover, this project has not been updated since the beginning of 2018. However, Docker (2016a) has support for Intel's SGX. Docker is an operating-system-level virtualization software that allows applications with all their dependencies to be packed into one container. It has emerged as a standard runtime for containers on Linux and can be used on multiple cloud providers. Unlike virtual machines, Docker containers share the kernel and other resources with the host system, requiring fewer resources than a virtual machine. Docker isolates processes from each other but does not give a full isolation guarantee such as virtual machines. Arnautov et al. (2016) proposed to combine Docker containers with SGX to create secure containers.

Microarchitectural attacks

The microarchitecture is the underlying implementation of an instruction-set architecture. 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 attacks able to derive cryptographic secrets (Kocher 1996; Page 2002; Bernstein 2005; Percival 2005).

While early attacks focused on the L1 caches, more 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 64B. The physical address determines to which cache set and byte offset a variable maps. The lowest 6 bits determine the byte offset within a cache way, the following $\log_2 S$ bits starting with bit 6 determine the cache set. Only the cache way is not derived from the physical address but chosen by the CPU using its cache replacement policy.

Prime+Probe is a cache attack technique that has first been used by Osvik et al. (2006). In a *Prime+Probe* attack, the attacker constantly primes (i.e. evicts) a cache set and measures how long this step took. This is accomplished by accessing an eviction set, which is a set of attacker-controlled addresses where each address falls into the same cache set. The amount of time the prime step took is correlated to the number of cache ways in this cache set 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 (Ristenpart et al. 2009; Zhang et al. 2011; Liu et al. 2015; Irazoqui et al. 2015; Maurice et al. 2017).

To prime (i.e. evict) a cache set, the attacker needs n addresses that map to the same cache set (i.e. an *eviction set*), where n depends on the cache replacement policy and the number of ways of the last-level cache. On Intel CPUs before Ivy Bridge, the cache replacement policy was Least-Recently Used (LRU), and thus it was sufficient to access n addresses for an n-way cache. However, on newer microarchitectures, the exact cache replacement policy is unknown. 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. (2016) 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. Figure 1 shows the eviction set access pattern of this eviction strategy.

A more powerful cache attack technique is *Flush+Reload* by Yarom and Falkner (2014). For a *Flush+Reload* attack, attacker and victim need to share memory (i.e. a shared library or page deduplication). The attacker flushes a shared memory line from the cache to then measure the amount of time it takes to reload the cache line. This reveals whether or not another program reloaded this exact cache line. Although *Flush+Reload* attacks have been studied extensively (Irazoqui et al. 2015; Gruss et al. 2015; Irazoqui et al. 2015; Gülmezoğlu et al. 2015; Lipp et al. 2016; Benger et al. 2014; Inci et al. 2016; Irazoqui et al. 2014; Irazoqui et al. 2016)

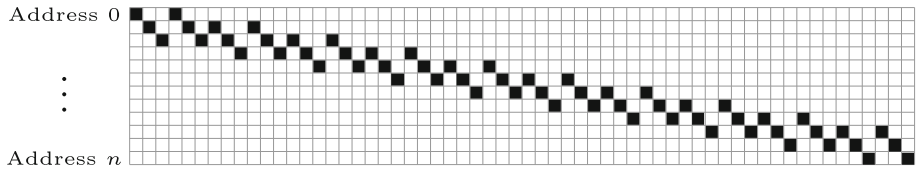

Fig. 1 The access pattern to the eviction set. Addresses 0 to n map to the same cache set

they are now considered impractical in the cloud as most cloud providers disabled page deduplication and thus disabled the only way to obtain shared memory in the cloud.

Pessl et al. (2016) found another attack vector that can yield an accuracy close to a *Flush+Reload* attack without requiring shared memory. They attack the DRAM modules that are shared by all virtual machines running on the same host system. 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.

DRAM has a strict hierarchy. DIMMs are connected through one or more *channels* to the CPU. Every DIMM has one or two sides (*ranks*) equipped with DRAM chips. Each rank is composed of multiple *banks*, usually 8 for DDR3 and 16 for DDR4. Finally, the banks are organized in *columns* and *rows* with a row size of typically 8kB. The memory controller maps physical addresses to DRAM cells using an undocumented mapping function reverse-engineered by Pessl et al. (2016). We use the term *same bank* address for addresses that map to the same DIMM, channel, rank, and bank. Same-bank addresses share one row buffer.

Side-channel attacks on SGX

There have been speculations that SGX could be vulnerable to cache side-channel attacks (Costan and Devadas 2016). In fact, Intel does not consider side channels as part of the SGX threat model and thus states that SGX does not provide any specific mechanisms to protect against side-channel attacks (Intel 2016). However, they also explicitly state that SGX features still impair side-channel attacks. Intel recommends using SGX enclaves to protect password managers and cryptographic keys against side channels and advertises this as a feature of SGX (Intel Corporation 2016b). Indeed, SGX does not provide special protection against microarchitectural attacks. Its focus lies on new attack vectors arising from an untrusted operating system. Xu et al. (2015) show that SGX is vulnerable to controlled channel attacks in which a malicious operating system triggers and monitors enclave page faults (Anati et al. 2015). Both attacks rely on a malicious or compromised operating system to break into an enclave.

SGX enclaves generally do not share memory with other enclaves, the operating system, or other processes. Thus, *Flush+Reload* attacks on SGX enclaves are not possible. 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.

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*, formerly known as PolarSSL. *mbedTLS* is used in many well-known open-source projects such as cURL and OpenVPN. The small size of *mbedTLS* is well suitable for the size-constrained enclaves of Intel SGX.

RSA essentially involves modular exponentiation with a private key, where the exponentiation is typically implemented as square-and-multiply, as outlined in Algorithm 1. The algorithm sequentially scans over all exponent bits. Squaring is done in each step, while multiplication is only carried out if the corresponding exponent bit is set. 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 (Yarom and Falkner 2014; Ge et al. 2016).

mbedTLS uses a windowed square-and-multiply routine for the exponentiation. To minimize the memory footprint of the library, the official knowledge base suggests setting the window size to 1 (ARMmbed 2016). With a fixed upper enclave memory limit in current microarchitectures, it is reasonable to follow this recommendation. However, a window size of 1 is equivalent to the basic square-and-multiply exponentiation, as shown in Algorithm 1. Liu et al. (2015) 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 (Arnaud and Fouque 2013). 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

Algorithm 1: Square-and-multiply exponentiation

input : base b, exponent e, modulus n
output: $b^e \mod n$

```
X ← 1;
for i ← bitlen(e) downto 0 do
    X ← multiply(X, X);
    if e_i = 1 then
        X ← multiply(X, b);
    end
end
return X;
```

multiplication routine, as illustrated in Algorithm 1. Thus, there is no leakage from a different square and multiplication routine as exploited in previous attacks on square-and-multiply algorithms (Acıiçmez and Schindler 2008; Zhang et al. 2012; Yarom and Falkner 2014; Liu et al. 2015). However, Liu et al. (2015) showed that the secret-dependent accesses to the buffer b still leak the exponent.

Boneh et al. (1998) and Blömer et al. (2003) showed that it is feasible to recover the full RSA private key if only some of either the most significant or least significant bits are known. Halderman et al. (2009) showed that it is even possible to recover a full RSA key if up to 12% of random bits are corrupted. Heninger et al. (2009) improved these results and recovered a full key for random unidirectional corruptions of up to 46%.

Threat model and attack setup

In this section, we present our threat model. We demonstrate a malware that circumvents SGX's and Docker's 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.

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 by many cloud providers, e.g., Amazon (Docker 2016b) or Microsoft Azure (Microsoft 2016). 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 2 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's SGX feature and are therefore subdivided into two parts, the enclave and loader, i.e. the main program that instantiates the enclave.

The attack is a multi-step process that can be divided into an online and offline phase. "Extracting private key information" section describes the online phase, in which the attacker first 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. "Recovering the private key" section gives a detailed explanation of the offline phase in which the attacker recovers a private key from collected traces.

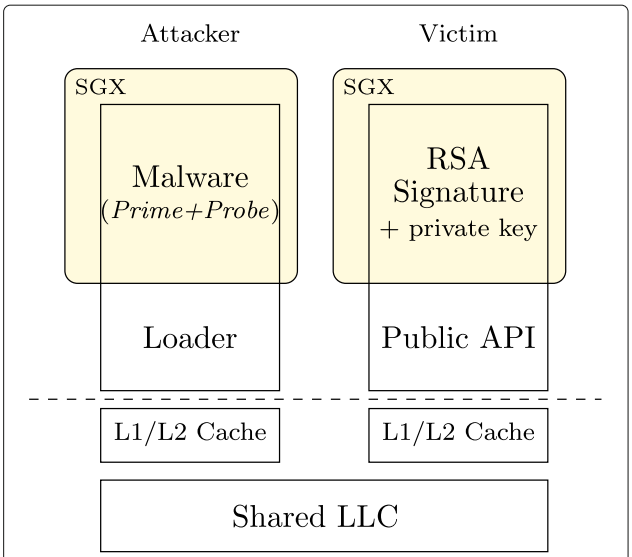

Fig. 2 The threat model: both attacker and victim run on the same physical machine in different SGX enclaves

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 (Intel Corporation 2016b). Thus, they can never be accessed by system software or malware on the same host. Moreover, information leakage from the enclave should not be possible due to hardware isolation and memory encryption. The victim uses the RSA implementation of the widely deployed *mbedTLS* library that relies on constant-time Montgomery multiplications. The victim application provides an API to compute a signature for the provided data.

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. To perform the exponentiation in RSA, *mbedTLS* uses a windowed square-and-multiply algorithm in the Montgomery domain. The window size is fixed to 1, as suggested by the official knowledge base (ARMmbed 2016). If successful, the attack can be extended to arbitrary window sizes (Liu et al. 2015).

To prevent information leakage from function calls, *mbedTLS* uses the same function (`mpi_montmul`) for both the square and the multiply operation (see Algorithm 1). 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's `calloc` implementation, the used buffers always have the same virtual and physical addresses. Thus the buffers are always in 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 entirely 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. The exfiltrated data from inside the malicious enclave will only be handed to the loader in an encrypted form. The loader may also provide a TLS endpoint through which the enclave can send encrypted data to an attacker's server.

Operating system and hardware

Previous work was mostly focused on attacks on enclaves from untrusted cloud operating systems (Li et al. 2014; Baumann et al. 2015; Schuster et al. 2015; Costan and Devadas 2016; Aumasson and Merino 2016). 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 its privileges.

Our only assumption on the hardware is that the attacker and victim run on the same host system. This is the case on both personal computers as well as on co-located Docker instances in the cloud. As SGX is currently only available on Intel's Skylake microarchitecture, 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.

Malware detection

We expect the cloud provider to run state-of-the-art malware detection software. We assume that malware detection software is able to monitor the behavior of containers or even inspect the content of containers. Moreover, the user can run anti-virus software and monitor programs inside the container. This software can either protect the data from infections or the infrastructure from attacks.

Standard malware detection methods are either signature-based, behavioral-based, or heuristics-based (Bazrafshan et al. 2013). Signature-based detection is used by virus scanners to match byte sequence insides executables against a list of such sequences extracted from known malware. This method is fast and rarely causes false-positives, but can only detect known malware (Sukwong et al. 2011). In addition to signature-based detection, modern virus scanners implement behavior-based analysis. Behavior-based analysis has the potential to detect new malware by monitoring system activity, API calls, and user interactions (Sukwong et al. 2011).

We also assume the presence of detection mechanisms using performance counters to detect malware (Demme et al. 2013) and microarchitectural attacks (Herath and Fogh 2015), which are more targeted to our attack.

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.

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 an arbitrary cache set. Due to the restrictions of SGX enclaves, we cannot rely on existing *Prime+Probe* implementations, and 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 SGX1, these instructions are not permitted inside an SGX enclave, as they might cause a VM exit (Intel 2014a). Therefore, we have to rely on a different timing source.

Lipp et al. (2016) demonstrated a counting thread as a high-resolution alternative on ARM, where no unprivileged high-resolution timer is available. The idea is to have a dedicated thread incrementing a global variable in an

endless loop. As the attacks only rely on accurate timing differences and not on absolute timestamps, this global variable serves directly as the timing source.

We require a minimum resolution in the order of 10cycles 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 the counter increment in inline assembly. According to Intel (Intel 2014b), the fastest instructions on the Skylake microarchitecture are `inc` and `add` with both a latency of 1cycle and a throughput of 0.25cycles/instruction when executed with a register as an operand. 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 4cycles due to L1 cache access times (Intel 2014b). To reduce the cost of the `jmp` instruction, we tried to unroll the loop up to the point where we get the most increments per CPU cycle. However, our experiments showed that the unrolling tends to rather have negative effects on the performance. On our test machine, the code from Listing 1 achieves one increment every 4.7cycles, which is an improvement of approximately 2% over the assembly code generated by `gcc` on the highest optimization level (`-O3`).

```
mov &counter, %rcx
1: incl (%rcx)
jmp 1b
```

Listing 1: A counting thread that emulates `rdtsc`.

We can improve the performance—and thus the resolution—further, by exploiting the fact that only the counting thread is writing to the counter variable. Reading the counter variable from memory is, therefore, never necessary as the value will not be changed by any other thread. To gain a higher performance from this observation, we have to eliminate the CPU's read access to the counter variable. Executing arithmetic operations directly on the memory location is thus not an option anymore, and it is necessary to perform any operation with data dependency on a CPU register. Therefore, we introduce a "shadow counter variable" which is always held in a CPU register. The arithmetic operation (either `add` or `inc`) is performed using this register as the operand, 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 1cycle and a throughput of 0.5cycles/instruction when copying a register to a memory location. Listing 2 shows the improved counting thread. This counting thread is capable of incrementing the variable by one every 0.87cycles, which is an improvement of 440% over the code in Listing 1. In fact, this version is even 15% faster than the native timestamp counter, thus giving us a reliable timing source that even has a higher resolution. This new method might open new possibilities of side-channel attacks that leak information through timing on a sub-`rdtsc` level. Figure 3 shows the performance comparison of the C version, the assembly version, the optimized assembly version, and the native timestamp counter as a baseline. Although the method with the shadow counter has the most instructions in the loop body, and an increase of 100% in code size compared to Listing 1, it has the best performance. Due to multiple execution units, pipelining, and the absence of memory dependencies, one increment can be carried out in less than 1cycle on the Skylake microarchitecture even though each instruction has a latency of 1cycle (Fog 2016).

```
mov &counter, %rcx
1: inc %rax
mov %rax, (%rcx)
jmp 1b
```

Listing 2: The improved fast counting thread that acts as the emulation of `rdtsc`.

Eviction set generation

Prime+Probe relies on eviction sets, i.e. we need to find virtual addresses that map to the same 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. This limitation also applies to enclaves, as they are always unprivileged. Liu et al. (2015) and Maurice et al. (2017) 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. At least in the current version, SGX does not support large pages, making this approach inapplicable. Oren et al. (2015) and Gruss et al. (2016) demonstrated fully automated methods to generate eviction sets for a given virtual address. However, the method of Oren et al. (2015) uses a slow pointer-chasing approach and needs to find an eviction set without any assumptions, consuming more time. The method by Gruss et al. (2016) has the overhead of finding an eviction strategy and eviction set without any

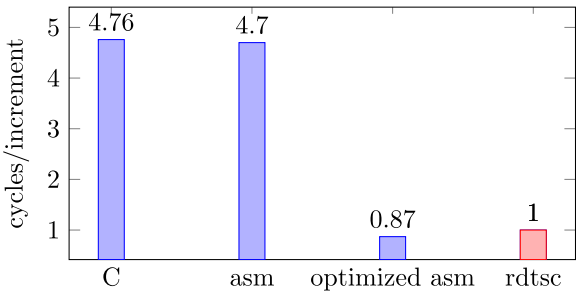

Fig. 3 Comparison of different variants for the counter thread to Intel's native timestamp counter as the baseline

assumptions. Thus, while these approaches work, applying them for our purposes would consume multiple hours on average before even starting the actual *Prime+Probe* attack.

We propose a new method to recover the cache set from a virtual address without relying on large pages. The method requires that an array within an SGX enclave is backed by physically contiguous pages. We verified that we have contiguous pages by inspecting Intel's SGX driver for Linux (Intel Corporation 2016a). When initializing a new enclave, the function `isgx_page_cache_init` creates a list of available physical pages for the enclave. These pages start at a base physical address and are contiguous. If a physical page is mapped, e.g., due to a page fault, the function `isgx_alloc_epc_page_fast` removes and returns the head of the list.

The idea is to exploit the DRAM timing differences that are due to the DRAM organization and to use the DRAM mapping functions (Pessl et al. 2016) to recover physical address bits. Alternately accessing two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of virtual addresses. For the first address of a DRAM row, the least-significant 18bits of the physical address are '0', because the row index only uses physical address bits 18 and upwards. Thus, we scan memory sequentially for an address pair in physical proximity that causes a *row conflict*. As SGX enclave memory is allocated in a contiguous way, 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 bank address (BA), bank group (BG), rank, and channel must be the same for both virtual addresses. Otherwise, a row conflict is not possible.
2. The row index must be different for both addresses.
3. The difference of the two physical addresses (of the virtual addresses) has to be at least 64B (the size of one cache line) but should not exceed 4kB (the size of one page).
4. Physical address bits 6 to 22 have the same known value, all 0 for the higher address and all 1 for the lower address, as only bits in this range are used by the mapping function.

For all virtual addresses satisfying these constraints, bits 6 to 22 have a known value. Thus, we know the exact cache set for these virtual addresses.

Table 1 shows the reverse-engineered DRAM mapping function for our test machine, an Intel Core i5-6200U with 12GB main memory. The row index is determined by the physical address bits starting from bit 18.

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 (De Moura and Bjørner 2008) to provide models satisfying the constraints. The model we found yields pairs of physical addresses where the upper address is 64B apart from the lower one. There are four such address pairs within every 4MB 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 22bits of the higher addresses least-significant bits are 0.

Figure 4 shows the average access time for hammering address pairs when iterating over a 2MB array. For every address, we take a second address which is 64B apart, and alternately access the two addresses while measuring the total access time for both accesses. The highest two peaks correspond to the highest timings (marked in the figure). These peaks show row conflicts, i.e. the row index changes while the bank, rank, and channel stay the same. As the cache set is determined by the bits 6 to 17, the higher address has the cache set index 0 at these peaks. Based on the assumption of contiguous memory, we can generate addresses mapping to the same cache set by adding multiples of 256KB to the higher address.

Table 1 Reverse-engineered DRAM mapping functions using the method from Pessl et al. (2016)

		Address Bit																
		32	31	30	29	18	17	16	15	14	13	12	11	10	9	8	7	6
2 DIMMs	Channel				⊕	⊕					⊕	⊕	⊕	⊕	⊕	⊕		⊕
	BG0									⊕							⊕	
	BG1	⊕				⊕												
	BA0				⊕				⊕									
	BA1		⊕				⊕											
	Rank			⊕				⊕										

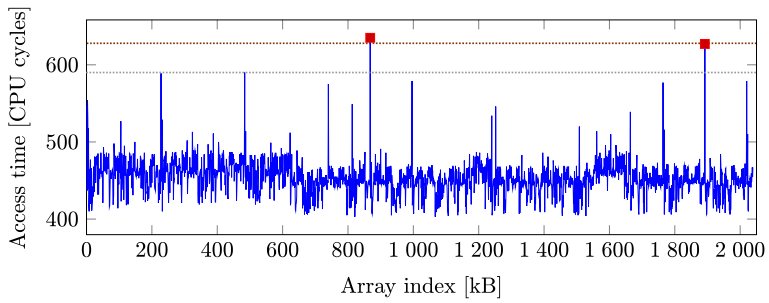

Fig. 4 Access times when alternately accessing two addresses which are 64B apart. The (marked) high access times indicate row conflicts

As the last-level cache is divided into multiple parts called cache slices, there is one cache set per slice for each cache set index. Thus, we will inherently add addresses to our generated eviction set that do not influence the eviction, although they have the correct cache set index. For the eviction set, it is necessary to only use addresses that map to the same cache slice. However, to calculate the cache slice from a physical address, all bits of the physical address are required (Maurice et al. 2015).

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. Algorithm 2 shows the full algorithm to generate 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. (2016). In cases where the physical memory is not contiguous, e.g., in a virtual machine, an attacker can still fall back to this slower approach.

Algorithm 2: Generating the eviction set

```
input  : memory: char[8 × 1024 × 1024], set: int
output: eviction_set: char*[n]
border ← 0;
border_index ← 0;
for i ← 0xFC0 to 4 × 1024 × 1024 step 4096 do
    time ← hammer(memory[i], memory[i + 64]);
    if time > border then
        border ← time;
        border_index ← i + 64;
    end
end
addr ← (&memory[border_index]) + set ≪ 6;
n ← 0;
repeat
    full_set[n] ← addr + n × 256KB;
    eviction ← evict(full_set, n);
    n ← n + 1;
until eviction > 99%;
for i ← 0 to n do
    removed ← full_set[i];
    full_set[i] ← NULL;
    if evict(full_set, n) < 99% then
    full_set[i] ← removed
end
len ← 0;
for i ← 0 to n do
    if full_set[i] ≠ NULL then
        eviction_set[len] ← full_set[i];
        len ← len + 1;
    end
end
```

Identifying vulnerable sets

Now that we have a 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 on the virtual or physical addresses of the victim, we have to scan the last-level cache for characteristic patterns that correspond to the signature process. We consecutively mount a *Prime+Probe* attack on every cache set while the victim is executing the exponentiation step. This allows us to log cache misses due to a victim's activity inside the monitored cache set.

First, we fill the cache lines of this cache set with the eviction set using the access pattern shown in Algorithm 3. This step is called the *prime* step. We expect our addresses to stay in the cache if the victim has no activity in this specific cache set. Second, we measure the runtime of this algorithm to infer information about the victim. We refer to this step as the *probe* step. Figure 5 shows the timings of a probe step with and without cache activity of the victim. If there was no activity of the victim

Algorithm 3: Attacker accessing a set.

```
input: n: int,
       addrs: int[n]
for i ← 0 to n − 2 do
    *addrs[i];
    *addrs[i+1];
    *addrs[i+2];
    *addrs[i];
    *addrs[i+1];
    *addrs[i+2];
end
```

inside the cache set, the probe step is fast as all addresses of the eviction set are still cached. If we encounter a high timing, we know that there was activity inside the cache set, and at least one of our addresses was evicted from the cache set. For all following measurements, the probe step also acts as the prime step. The measurement ensures that the eviction set is cached again for the next round.

We can identify multiple cache sets showing this distinctive pattern which consists of three parts. The start of an exponentiation is characterized by a high usage of the cache set due to clearing and initialization of the used buffers. It is followed by the actual exponentiation that depends on the secret exponent. The exponentiation ends with another high peak where the buffer is cleared, followed by no cache misses anymore, i.e. it is only influenced by background noise.

To automatically find these sets, we apply a simple peak detection to find the rightmost peak. If we can identify another peak before that within a certain range, we assume that our target buffer uses this cache set. Depending on the size of the RSA exponent, we get multiple cache sets matching this pattern. Our experiments showed that using identified sets, which are neither at the beginning nor the end, yields good results in the actual attack. Neighboring buffers might use the first and last cache set, and they are more likely to be prefetched (Yarom and Falkner 2014; Gruss et al. 2015). Thus, they are more prone to measurement errors.

Monitoring vulnerable sets

Once we have identified a cache set which is used by the exponentiation, we can collect the actual traces. 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.

Figure 6 shows around 700 *Prime+Probe* measurements captured during one run of the signature algorithm. We can see intervals with multiple cache misses and intervals without cache misses, corresponding to high cache usage and no cache usage of the victim, respectively. 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.

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's enclave or on the attacker's server.

Ideally, one would combine multiple traces by aligning them and averaging out the noise. The more traces are combined, the more noise is eliminated. From the resulting averaged trace, one can easily extract the private key. However, the traces obtained in our attack are affected by several noise sources. Most of them alter the timing, making trace alignment difficult. Among them

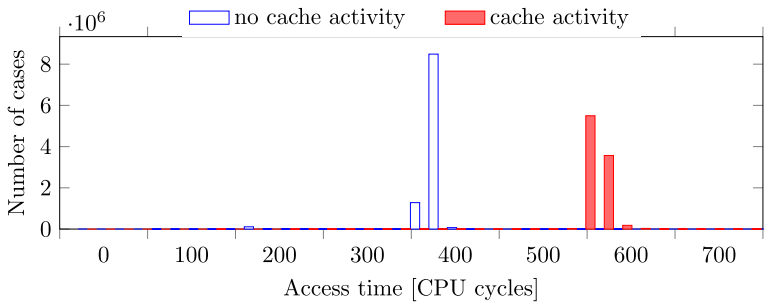

Fig. 5 Histogram showing the runtime of the prime step for cache activity in the same set and no cache activity in the same set

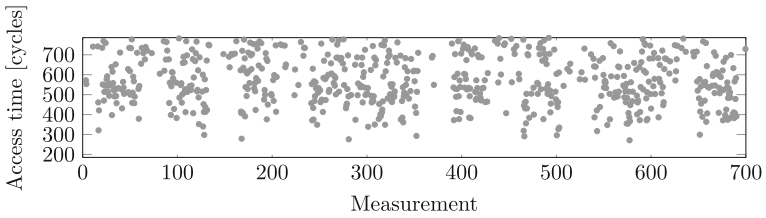

Fig. 6 Dense areas indicate a high cache-hit rate, white areas are intervals with cache misses

are interrupts which lead to context switches and, therefore descheduling of the attacker or the victim. Other sources of noise include unrelated activity on the enclave's cache sets and varying CPU clock frequency due to power management. Although methods exist for aligning such traces (van Woudenberg et al. 2011; Muijrers et al. 2011), we opt for a different strategy. Instead of attempting to align traces beforehand, 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 those errors, multiple partial keys are combined in the key recovery phase. This approach has much lower computational overhead than trace alignment since key recovery is performed on partial keys of length 4KB 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.

Pre-processing

In the pre-processing step, we filter and resample raw measurement data. Figure 7 shows a trace segment before (top) and after pre-processing (bottom). High values in the raw measurement data correspond to cache misses, whereas low values indicate cache hits. Timing measurements have a varying sample rate. This is because a cache miss delays the next measurement while cache hits allow more frequent measurements. To simplify the subsequent steps, we convert the measurements to a constant sampling rate. Therefore, we specify sampling points 1000cycles apart. At each sampling point, we compute the normalized sum of squared measurements within a 10000cycle window. Squaring the measurements is necessary to account for the varying sampling rate. If the measurements exceed a certain threshold, they are considered as noise and are discarded. If too few measurements are available within a window, e.g., due to an interrupt, we apply linear interpolation. The resulting resampled trace shows high peaks at locations of cache misses, indicating a '1' in the RSA exponent, as shown in Figure 7 on the bottom.

Partial key extraction

To automatically extract a partial key from a resampled 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 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 Figure 7. The adaptive threshold is the median over the 10 previously detected peaks. If a peak drops below 90% of this

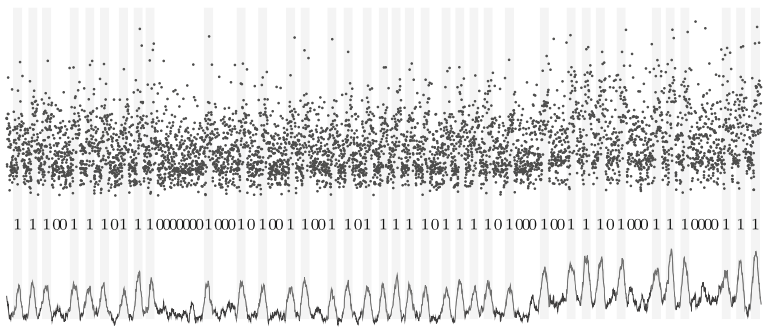

Fig. 7 On top is one trace's raw measurement over 4000000cycles. The peaks in the resampled trace on the bottom clearly indicate '1's

threshold, it is discarded. The remaining peaks correspond to the '1's in the RSA exponent and are highlighted in Figure 7. '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.

A partial key might suffer from bit flips, random insertions, and deletions, when compared to the correct key. 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. Moreover, if the attacker is not scheduled, we miss certain key bits in the trace. If the victim is not scheduled, we see a region of cache inactivity in the measurement that cannot be distinguished from true '0's. Finally, 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 in fact an advantage of a counting thread over the use of the native timestamp counter. The remaining errors in the partial keys are corrected in the final key recovery.

Final key recovery

In the final key recovery, we merge multiple partial keys to obtain the full key. We quantify partial key errors using the edit distance (Levenshtein 1966). 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.

Algorithm 4 shows the pseudo code for the final key recovery. 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. Before proceeding to the next key bit, we correct all wrong partial keys which did not match the recovered key bit. To correct the current bit of the wrong partial key, we compute the edit distance to all partial keys that won the majority vote. To reduce performance overhead, we calculate the edit distance, not 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. Table 2 gives an example where the topmost 5 bits are already recovered (underlined). The sixth key bit is recovered as '1', since all partial key bits—except for the second one—are '1' (bold). The incorrect '0' of the second partial key is deleted before proceeding to the next bit. This procedure is repeated for all key bits until the majority of partial keys reached the last bit.

Algorithm 4: RSA private key recovery.

```
input  : keys: boolean[], lookahead: int
output: key: boolean[]

key ←[ ];
i ← 0;
while True do
    keybit ← majority(keys, i);
    if keybit =⊥ then
        return key;
    end
    key[ i] ← keybit;
    correct ← {};
    wrong ← {};
    foreach k in keys do
        if k[ i] = keybit then
            correct ← correct ∪ k;
        else
            wrong ← wrong ∪ k;
        end
    end
    foreach kw in wrong do
        actions ← {};
        foreach kc in correct do
            actions ← actions∪
            EditDistance(kw[ i : i + lookahead],
                         kc[ i : i + lookahead]);
        end
        ai ← 0;
        while kw[ i] ≠ keybit do
            action ← majority(actions, ai);
            apply action to kw[ i];
            ai++;
        end
    end
    i++;
end

function majority(set, idx) begin
    counter[ ] ← 0;
    foreach array in set do
        element ← array[ idx];
        increment counter[ element];
    end
    return element with max. counter;
end
```

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 "Victim" section, we use *mbedTLS*, in version 2.3.0. The small code and memory footprint and self-containment of *mbedTLS* makes it easy to use in SGX enclaves.

Table 2 Bit-wise key recovery over five partial keys

No.	Recovered key
1	10111**1**10001100110010111101010000100...
2	10111*0*11000111001100101101101010000...
3	10111**1**10001110011001011110101000010...
4	10111**1**10001110001100101111010100001...
5	10111**1**10001110011001011100010100001...

RSA key sizes and exploitation

For the evaluation, we attack a 4096-bit RSA key as this provides long-term security, based on the recommendation of NIST (Barker and Roginsky 2015). Higher bit sizes are rarely used outside tinfoil-hat environments.

Table 3 shows various RSA key sizes and the corresponding buffer sizes in *mbedTLS*. 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 (Walter 2003; Yarom and Benger 2014; Yarom and Falkner 2014; Pereida García et al. 2016).

Native environment

We use a Lenovo ThinkPad T460s running Ubuntu 16.10. This computer supports SGX1 using Intel's SGX driver. The hardware details for the evaluation are shown in Table 4. 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's enclave.

Figure 8 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 low bit-error rate, as shown in Figure 9. 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 (Yarom and Falkner 2014; Gruss et al. 2015).

Table 3 RSA key sizes and the corresponding CPU cycles to execute one multiplication

Key size	Buffer size	Cache sets	CPU cycles
1024b	136B	3	1764
2048b	264B	5	6624
4096b	520B	9	25462
8192b	1032B	17	100440

Table 4 Experimental setup

Environment	CPU model	Cores	LLC associativity
Native	Core i5-6200U	2	12
Docker	Core i5-6200U	2	12

Detecting one vulnerable cache set within all 2048 cache sets requires about 340 trials on average. With a monitoring time of 0.21s per cache set, we require a maximum of 72s 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 3min.

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 107662cycles. This differs from the values shown in Table 3 because the attacker constantly evicts the victim's buffer, inherently causing a slowdown. In addition, one could artificially slow down a victim through constant eviction to improve the performance of cache attacks. This is known as performance degradation (Allan et al. 2015). However, as the *Prime+Probe* measurement takes on average 734cycles, we do not have to artificially slow down the victim and thus remain stealthy.

When looking at a single trace, we can already recover about 96% of the RSA private key, as shown in Figure 9. For a full key recovery, we combine multiple traces using our key recovery algorithm, as explained in "Final key recovery" section. We first determine a reasonable lookahead window size. Figure 10 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 11 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 18.5sec. This results in a total runtime of less than 130sec for the offline key recovery process.

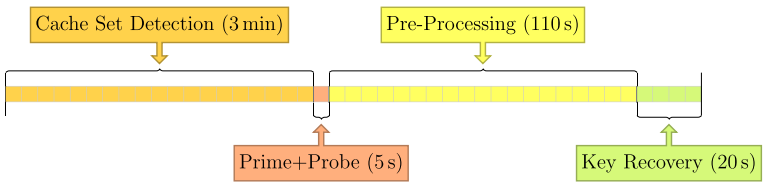

Fig. 8 A high-level overview of the average times for each step of the attack

Generalization

Based on our experiments, we can deduce that the same 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 point of view, 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, independently of 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 5 summarizes our results. In contrast to concurrent work on cache attacks on SGX (Götzfried et al. 2017; Brasser et al. 2017; Moghimi et al. 2017), our attack is the only one that runs from within an enclave and is thus not detectable.

Virtualized environment

We now show that the attack also works in a virtualized environment.

As described in "Intel SGX in native and virtualized environments" section, no hypervisor with SGX support was available at the time of our experiments. Instead of full virtualization using a virtual machine, we used the lightweight Docker containers. Docker containers are also used by large cloud providers, e.g., Amazon (Docker 2016b) or Microsoft Azure (Microsoft 2016). 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 12 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. They can use any functionality provided by the host operating system. Consequently, the unmodified malware also works inside containers.

Considering the performance within Docker, only I/O operations and network access have a measurable overhead (Felter et al. 2015). 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. These results confirm that containers do not provide additional protection against our malware at all.

Furthermore, we can speculate whether our malware would also work within virtual machines based on the experimental KVM support description (Intel Corporation 2016c). Many cross-VM cache attacks have been demonstrated in the past years (Irazoqui et al. 2014; Inci et al. 2015; Liu et al. 2015), as the CPU cache is a shared resource in virtual machines. This does not change with SGX, and thus, enclaves inside virtual machines will also share the last-level cache. The experimental implementation for KVM relies on the host system's SGX driver to provide memory pages to the enclave inside the

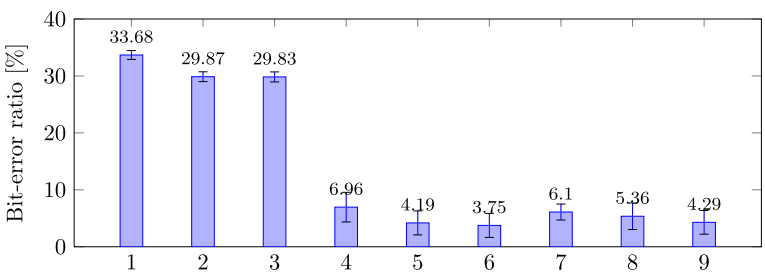

Fig. 9 The 9 cache sets that are used by a 4096b key and their error rate when recovering the key from a single trace

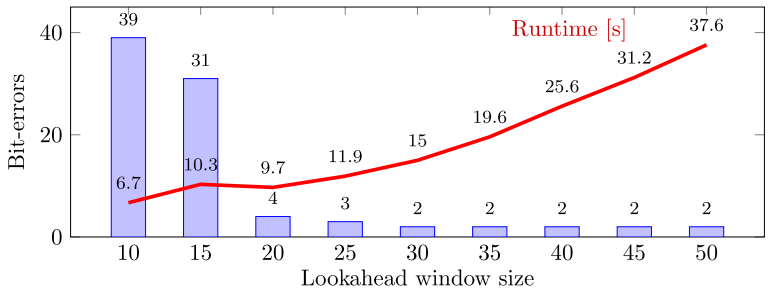

Fig. 10 Up to a lookahead window size of 30, an increased window size reduces the number of bit errors while increasing recovery runtime. The measurement is conducted with 7 traces

virtual machine. We thus expect that our malware will work across virtual machines either with only minor changes, or even without any adaptations.

Countermeasures

In this section, we discuss advantages and disadvantages of different countermeasures. Previously presented countermeasures mostly cannot be applied to a scenario where a malicious enclave performs a cache attack and no assumptions about the operating system are made. We group countermeasures into 3 categories, based on whether they require:

1. a modification of the enclave (source level),
2. a modification of the operating system (OS level) assuming the operating system is benign,
3. a change in hardware (hardware level).

Source level

Exponent blinding

A generic side-channel protection for RSA is exponent blinding (Kocher 1996). To sign a message m, the signer generates a random blinding value k for each signature. The signer then calculates the signature as $m^{d+k\cdot\phi(N)}$ mod N where d is the private key, and N is the RSA modulus.

An attacker will only be able to measure the blinded exponent on every execution. When a single-trace key recovery is not possible, the attacker has to wait for collisions, i.e. signatures where the same blinding was used. For a sufficiently large blinding factor k, e.g., 64bit, this becomes infeasible in practice. As the exponent grows with the blinding factor, this solution is a trade-off between performance and side-channel resistance. This has no effect if key recovery from a single trace is possible, only if more than one trace is required. Furthermore, this countermeasure relies on the presence of a random number source.

Exponent blinding is specific to certain cryptographic operations, such as RSA signature computations. It will prevent the proposed attack, but other parts of the signature process might still be vulnerable to an attack (Schindler 2015).

Bit slicing

Bit slicing is a technique originally proposed by Biham (1997) to improve the performance of DES. Matsui (2006) was the first to show a bit-sliced implementation of AES. Sudhakar et al. (2007) presented a bit-sliced Montgomery multiplication for RSA and ECC. The main idea of bit slicing is to use only bit operations for computations throughout the algorithm. No lookup tables or branches are used in these algorithms, and thus, they are not vulnerable to cache attacks.

Again, this countermeasure is specific to certain cryptographic algorithms. It requires the support of the used

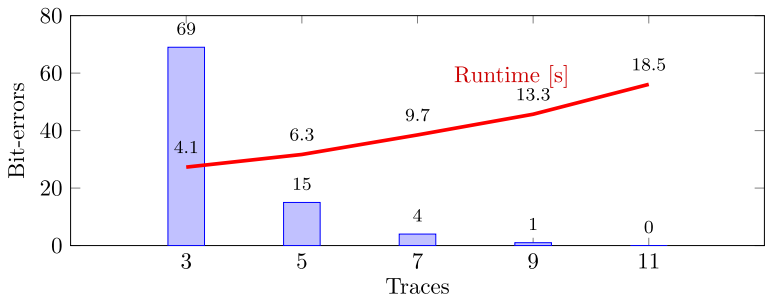

Fig. 11 With the number of captured traces, the number of bit errors decreases while the runtime to recover the key increases

Table 5 Our results show that cache attacks can be mounted successfully in the shown scenarios

Attack to	Benign	Benign	Benign
Attack from	Userspace	Kernel	SGX Enclave
Malicious Userspace	✓(Osvik et al. 2006; Liu et al. 2015)	✓(Hund et al. 2013)	✓**new**
Malicious Kernel	—	—	✓**new** (Götzfried et al. 2017; Brasser et al. 2017; Moghimi et al. 2017)
Malicious SGX Enclave	✓**new**	✓**new**	✓**new**

cryptography library and hardware support for streaming SIMD (SSE) instructions is necessary to achieve a reasonable performance (Käsper and Schwabe 2009). Bit slicing can be a good software solution while there is no hardware countermeasure. Other countermeasures for cryptographic implementations have been discussed by Ge et al. (2016).

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 and is thus unrealistic. However, we want to discuss the different possibilities in order to provide valuable information for the design process of future enclave systems.

Eliminating timers

Removing access to high-resolution timers (Percival 2005; Gullasch et al. 2011) or decreasing the accuracy (Hu 1992; Vattikonda et al. 2011; Martin et al. 2012) is often discussed as a countermeasure against cache attacks. However, our results using the timing counter show that removing precise timers is not a viable countermeasure, as we are still able to mount a high-resolution *Prime+Probe* attack. Moreover, on recent microarchitectures, we can even get a higher resolution using our timing thread than with the native high-resolution timestamp counter.

However, it is possible to remove access to high-resolution timers and all forms of simultaneous multithreading to prevent this alternative approach. This would effectively eliminate access to sufficiently accurate timers and mitigate many attacks.

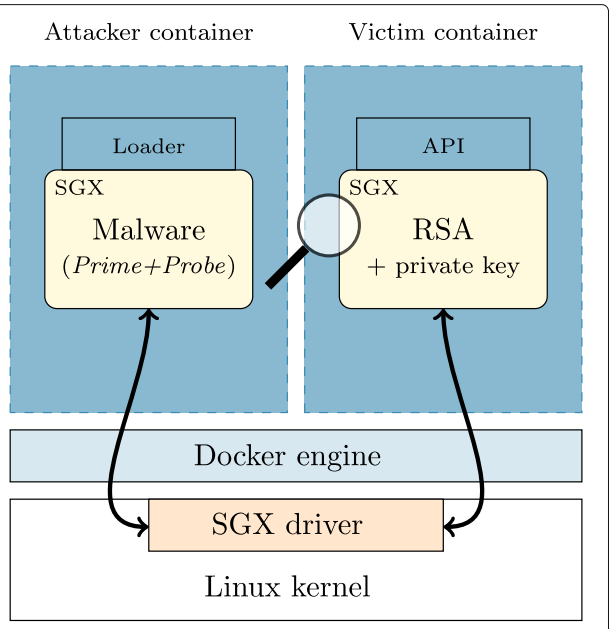

Fig. 12 Running the SGX enclaves inside Docker containers to provide further isolation. The host provides both containers access to the same SGX driver

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. 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.

Heuristic methods, such as behavior-based detection, are not applicable, as the malicious enclave does not rely on API calls or user interaction. Furthermore, for encrypted enclave code, a signature-based virus scanner has no access to the code, and the malware can easily change its signature by either re-encryption or modification of the plaintext. Thus, only the host binary—which contains no malicious code—can be inspected by a virus scanner.

Herath and Fogh (2015) proposed to use hardware performance counters to detect cache attacks. Subsequently, several other approaches instrumenting performance counters to detect cache attacks have been proposed (Chiappetta et al. 2015; Gruss et al. 2016; Payer 2016). However, according to Intel, the SGX enclave activity is not visible in the thread-specific performance counters (Intel Corporation 2016b). We verified that even performance counters for last-level cache accesses are disabled for enclaves. Figure 13 shows the results of a simple test program running inside a debug and pre-release enclave, and without an enclave. The visible cache hits and misses are caused by the host application only. This makes it impossible for current antivirus software and other detection mechanisms to detect the malware.

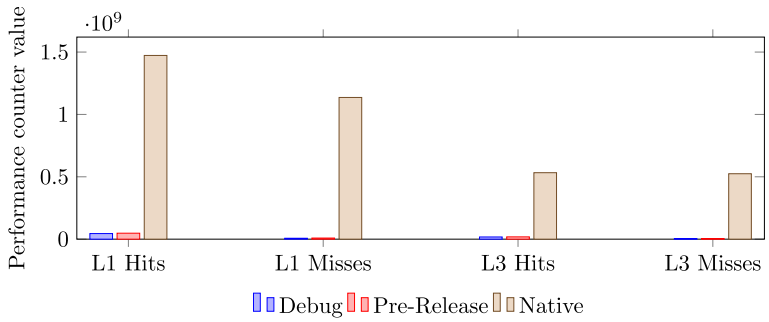

Fig. 13 Performance counters for caches are disabled in an enclave. *Flush+Reload* of one variable in a loop results in a high cache activity, which can be seen in native environment, but not on SGX debug or pre-release mode

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 parts. Each of the parts spans over multiple cache sets, and no cache set is included in more than one part. An enclave gets one or more such cache parts. This assignment of cache parts is either done by the hardware or by a trusted operating system.

If implemented in software, the operating system can split the last-level cache through memory allocation. The cache set is determined by bits of the physical address. The lower bits of the cache set index are below bit 12 and therefore determined by the page offset, i.e. the data's position within a 4KB page. The upper bits of the cache set are not visible to the enclave application and can thus be controlled by the operating system when allocating pages. 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 thus an eviction of the data—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.

Enclave coloring requires a trusted operating system and is therefore not always applicable as it contradicts SGX's idea of having an untrusted operating system (Costan and Devadas 2016). If the operating is trusted, this is an effective countermeasure against cross-enclave cache attacks.

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 (Raj et al. 2009). Godfrey et al. (2014) 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 indeed the case, as the memory allocator (`dlmalloc`) has a deterministic behavior and uses a best-fit approach for moderate buffer sizes as used in the RSA implementation. Freeing a buffer and allocating it again will always result in the same memory location for the 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, an automatic cache set detection is not possible anymore, as the identified set will change for the next run of the algorithm. Second, if more than one trace is required to reconstruct the key, this countermeasure increases the number of required traces by multiple orders of magnitude as the probability to measure the correct cache set decreases.

Although not obvious at first glance, this method requires a certain amount of trust in the underlying 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 from 0.1% to 7%.

Intel CAT

Recently, Intel introduced an instruction set extension called CAT (cache allocation technology) (Intel 2014a). 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. (2016) 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 also does not allow to protect enclaves from an outside attacker.

Hardware level

Combining intel CAT with SGX

Instead of using Intel CAT on the operating 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 secure memory element that resides inside the CPU. Data stored within this memory is not cachable. Thus the memory has to be fast to not incur performance penalties.

The SGX driver can then provide a special API to acquire this 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.

As data from this element is only accessed by one program and is never cached, 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 or to probe the memory bus. It is unclear whether Intel's eDRAM implementation can already be abused as a secure memory to protect applications against cache attacks.

Conclusion

There have been speculations that SGX could be vulnerable to cache side-channel attacks and might allow the implementation of super malware. However, Intel claimed that SGX features impair side-channel attacks and recommends using SGX enclaves to protect cryptographic computations. Furthermore, it was presumed that they cannot perform harmful operations.

In this paper, we demonstrated the first malware running in real SGX hardware enclaves. We demonstrated private key theft in a fully automated end-to-end attack from a co-located SGX enclave, despite all restrictions of SGX, e.g., no timers, no large pages, no physical addresses, and no shared memory.

We developed the most accurate timing measurement technique 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* that is used, for instance, in OpenVPN. The attack succeeds despite protection against side-channel attacks using a constant-time multiplication primitive. We extract 96% of a 4096-bit RSA private key from a single *Prime+Probe* trace and achieve full key recovery from only 11 traces within 5 min.

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. Intel intentionally does not include SGX activity in the performance counters for security reasons. However, 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 that should be considered for future versions.

Acknowledgements
Not applicable.

Authors' contributions
MS had the initial idea and wrote most parts of the manuscript. SW did most of the attack implementation. DG provided ideas and text for "Virtualized environment" and "Countermeasures" sections. CM gave technical advise on the implementation and improved the text. SM provided valuable feedback on many concepts. All authors reviewed the final manuscript. All authors read and approved the final manuscript.

Funding
This project has received funding from the European Research Council (ERC) under the European Union's 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".

Availability of data and materials
All relevant code parts which were developed in the paper are contained in the paper.

Ethics approval and consent to participate
Not applicable.

Consent for publication
All authors consent to the publication.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Graz University of Technology, Graz, Austria. [2]CNRS, IRISA, Rennes, France.

Received: 2 August 2019 Accepted: 18 December 2019
Published online: 19 January 2020

References

Acıiçmez O, Schindler W (2008) A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In: CT-RSA 2008. https://doi.org/10.1007/978-3-540-79263-5_16

Allan T, Brumley BB, Falkner K, Pol JVD, Yarom Y (2015) Amplifying Side Channels Through Performance Degradation. Cryptology ePrint Archive: Report 2015/1141. https://doi.org/10.1145/2991079.2991084

Anati I, McKeen F, Gueron S, Huang H, Johnson S, Leslie-Hurd R, Patil H, Rozas CV, Shafi H (2015) Intel Software Guard Extensions (Intel SGX). Tutorial Slides presented at ICSA 2015. https://sgxisca.weebly.com/

ARMmbed (2016) Reduce mbed TLS memory and storage footprint. https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint. Accessed 24 Oct 2016

Arnaud C, Fouque P-A (2013) Timing attack against protected rsa-crt implementation used in polarssl. In: CT-RSA 2013. https://doi.org/10.1007/978-3-642-36095-4_2

Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C, Lind J, Muthukumaran D, O'Keeffe D, Stillwell ML, et al. (2016) Scone: Secure linux containers with intel sgx. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

Aumasson J-P, Merino L (2016) SGX Secure Enclaves in Practice: Security and Crypto Review. In: Black Hat 2016 Briefings. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review-wp.pdf

Barker EB, Roginsky AL (2015) Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Nist sp 800-131a revision 1. https://doi.org/10.6028/nist.sp.800-131a

Baumann A, Peinado M, Hunt G (2015) Shielding applications from an untrusted cloud with haven. ACM Trans Comput Syst (TOCS). https://doi.org/10.1145/2799647

Bazrafshan Z, Hashemi H, Fard SMH, Hamzeh A (2013) A survey on heuristic malware detection techniques. In: The 5th Conference on Information and Knowledge Technology. Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/ikt.2013.6620049

Benger N, van de Pol J, Smart NP, Yarom Y (2014) "ooh aah... just a little bit": A small amount of side channel can go a long way. In: CHES'14. https://doi.org/10.1007/978-3-662-44709-3_5

Bernstein DJ (2005) Cache-timing attacks on AES. Technical report. Department of Mathematics, Statistics, and Computer Science, University of Illinois, Chicago

Biham E (1997) A fast new des implementation in software. In: International Workshop on Fast Software Encryption. pp 260–272. https://doi.org/10.1007/bfb0052352

Blömer J, May A (2003) New partial key exposure attacks on rsa. In: Crypto'03. https://link.springer.com/chapter/10.1007/978-3-540-45146-4_2

Boneh D, Durfee G, Frankel Y (1998) An attack on rsa given a small fraction of the private key bits. In: International Conference on the Theory and Application of Cryptology and Information Security. https://doi.org/10.1007/3-540-49649-1_3

Brasser F, Müller U, Dmitrienko A, Kostiainen K, Capkun S, Sadeghi A-R (2017) Software grand exposure: SGX cache attacks are practical. In: WOOT. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

Chiappetta M, Savas E, Yilmaz C (2015) Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptol ePrint Archive, Report 2015/1034. https://doi.org/10.1016/j.asoc.2016.09.014

Costan V, Devadas S (2016) Intel sgx explained. Technical report. Cryptology ePrint Archive, Report 2016/086

De Moura L, Bjørner N (2008) Z3: An efficient smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. pp 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S (2013) On the feasibility of online malware detection with performance counters. ACM SIGARCH Comput Archit News 41(3):559–570

Docker (2016) What is Docker? https://www.docker.com/what-docker

Docker (2016) Amazon Web Services - Docker. https://docs.docker.com/machine/drivers/aws/

Felter W, Ferreira A, Rajamony R, Rubio J (2015) An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS). https://doi.org/10.1109/ispass.2015.7095802

Fog A (2016) The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler makers. http://www.agner.org/optimize/microarchitecture.pdf. Accessed 16 Jan 2016

Ge Q, Yarom Y, Cock D, Heiser G (2016) A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Technical report. Cryptology ePrint Archive, Report 2016/613, 2016. https://doi.org/10.1007/s13389-016-0141-6

Godfrey MM, Zulkernine M (2014) Preventing cache-based side-channel attacks in a cloud environment. IEEE Trans Cloud Comput. https://doi.org/10.1109/tcc.2014.2358236

Götzfried J, Eckert M, Schinzel S, Müller T (2017) Cache attacks on intel sgx. In: EuroSec. https://doi.org/10.1145/3065913.3065915

Gruss D, Maurice C, Mangard S (2016) Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16. https://doi.org/10.1007/978-3-319-40667-1_15

Gruss D, Maurice C, Wagner K, Mangard S (2016) Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA'16. https://doi.org/10.1007/978-3-319-40667-1_14

Gruss D, Spreitzer R, Mangard S (2015) Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. https://www.usenix.org/node/191011

Gullasch D, Bangerter E, Krenn S (2011) Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P'11. https://doi.org/10.1109/sp.2011.22

Gülmezoğlu B, Inci MS, Eisenbarth T, Sunar B (2015) A Faster and More Realistic Flush+Reload Attack on AES. In: Constructive Side-Channel Analysis and Secure Design (COSADE). https://doi.org/10.1007/978-3-319-21476-4_8

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, Feldman AJ, Appelbaum J, Felten EW (2009) Lest we remember: cold-boot attacks on encryption keys. Commun ACM. https://doi.org/10.1145/1506409.1506429

Heninger N, Shacham H (2009) Reconstructing RSA Private Keys from Random Key Bits. https://doi.org/10.1007/978-3-642-03356-8_1

Herath N, Fogh A (2015) These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat 2015 Briefings. https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf

Hu W-M (1992) Reducing timing channels with fuzzy time. J Comput Secur. https://doi.org/10.1109/risp.1991.130768

Hund R, Willems C, Holz T (2013) Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13. https://doi.org/10.1109/sp.2013.23

Inci MS, Gulmezoglu B, Irazoqui G, Eisenbarth T, Sunar B (2015) Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Technical report. Cryptology ePrint Archive, Report 2015/898, 2015

Inci MS, Gulmezoglu B, Irazoqui G, Eisenbarth T, Sunar B (2016) Cache attacks enable bulk key recovery on the cloud. In: CHES'16. https://doi.org/10.1007/978-3-662-53140-2_18

Intel (2014a) Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide 253665. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architecturesoptimization-manual.pdf

Intel (2016) Software Guard Extensions SDK for Linux OS Developer Reference. Intel Corporation. Rev 1.5. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf

Intel Corporation (2016a) Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. Accessed 7 Nov 2016

Intel Corporation (2016b) Hardening Password Managers with Intel Software Guard Extensions: White Paper. https://pdfs.semanticscholar.org/ec40/833215b3d415c9525940690d0a94d2a178ca.pdf

Intel Corporation (2016c) kvm-sgx wiki. https://github.com/01org/kvm-sgx/wiki. Accessed 11 Nov 2016

Intel Corporation (2016) Intel(R) Software Guard Extensions for Linux* OS. https://github.com/01org/linux-sgx-driver. Accessed 11 Nov 2016

Intel Corporation (2016) 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

Intel (2014b) Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/architecture-andtechnology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html
Irazoqui G, Eisenbarth T, Sunar B (2015) S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P'15. https://doi.org/10.1109/sp.2015.42
Irazoqui G, Eisenbarth T, Sunar B (2016) Cross processor cache attacks. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'16). https://doi.org/10.1145/2897845.2897867
Irazoqui G, Inci MS, Eisenbarth T, Sunar B (2014) Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. https://doi.org/10.1007/978-3-319-11379-1_15
Irazoqui G, Inci MS, Eisenbarth T, Sunar B (2015) Know thy neighbor: Crypto library detection in cloud. Proc Priv Enhancing Technol 1(1):25–40
Käsper E, Schwabe P (2009) Faster and timing-attack resistant AES-GCM. In: Cryptographic Hardware and Embedded Systems (CHES). pp 1–17. https://doi.org/10.1007/978-3-642-04138-9_1
Kocher PC (1996) Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: Crypto'96. https://doi.org/10.1007/3-540-68697-5_9
Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions and reversals. In: Soviet Physics Doklady, vol. 10. p 707
Li Y, McCune J, Newsome J, Perrig A, Baker B, Drewry W (2014) Minibox: A two-way sandbox for x86 native code. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). https://www.usenix.org/node/183976
Lipp M, Gruss D, Spreitzer R, Maurice C, Mangard S (2016) ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp
Liu F, Ge Q, Yarom Y, Mckeen F, Rozas C, Heiser G, Lee RB (2016) Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: IEEE International Symposium on High Performance Computer Architecture (HPCA'16). https://doi.org/10.1109/hpca.2016.7446082
Liu F, Yarom Y, Ge Q, Heiser G, Lee RB (2015) Last-Level Cache Side-Channel Attacks are Practical. In: S&P'15. https://doi.org/10.1109/sp.2015.43
Martin R, Demme J, Sethumadhavan S (2012) Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. ACM SIGARCH Comput Archit News. https://doi.org/10.1109/isca.2012.6237011
Matsui M (2006) How far can we go on the x64 processors? In: International Workshop on Fast Software Encryption. Springer. pp 341–358. https://doi.org/10.1007/11799313_22
Maurice C, Le Scouarnec N, Neumann C, Heen O, Francillon A (2015) Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID'15. https://doi.org/10.1007/978-3-319-26362-5_3
Maurice C, Weber M, Schwarz M, Giner L, Gruss D, Boano CA, Mangard S, Römer K (2017) Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17. to appear. https://doi.org/10.14722/ndss.2017.23294
Microsoft (2016) Create a Docker environment in Azure using the Docker VM extension. https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-dockerextension/
Moghimi A, Irazoqui G, Eisenbarth T (2017) Cachezoom: How sgx amplifies the power of cache attacks. In: CHES. https://doi.org/10.1007/978-3-319-66787-4_4
Muijrers RA, van Woudenberg JG, Batina L (2011) Ram: Rapid alignment method. In: International Conference on Smart Card Research and Advanced Applications. Springer. https://doi.org/10.1007/978-3-642-27257-8_17
Oren Y, Kemerlis VP, Sethumadhavan S, Keromytis AD (2015) The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS'15. https://doi.org/10.1145/2810103.2813708
Osvik DA, Shamir A, Tromer E (2006) Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA 2006. https://doi.org/10.1007/11605805_1
Page D (2002) Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169 2002:169. https://eprint.iacr.org/2002/169
Payer M (2016) HexPADS: a platform to detect "stealth" attacks. In: ESSoS'16. https://doi.org/10.1007/978-3-319-30806-7_9
Percival C (2005) Cache missing for fun and profit. In: Proceedings of BSDCan. https://papers.freebsd.org/2005/cperciva-cache_missing/
Pereida García C, Brumley BB, Yarom Y (2016) Make sure dsa signing exponentiations really are constant-time. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/2976749.2978420
Pessl P, Gruss D, Maurice C, Schwarz M, Mangard S (2016) DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl
Raj H, Nathuji R, Singh A, England P (2009) Resource Management for Isolation Enhanced Cloud Services. In: Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09). pp 77–84. https://doi.org/10.1145/1655008.1655019
Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS'09. https://doi.org/10.1145/1653662.1653687
Rutkowska J (2013) Thoughts on Intel's upcoming Software Guard Extensions (Part 2). http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-on-intels-upcom%ing-software.html. Accessed 20 Oct 2016
Schindler W (2015) Exclusive exponent blinding may not suffice to prevent timing attacks on rsa. In: International Workshop on Cryptographic Hardware and Embedded Systems. https://doi.org/10.1007/978-3-662-48324-4_12
Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, Russinovich M (2015) Vc3: trustworthy data analytics in the cloud using sgx. https://doi.org/10.1109/sp.2015.10
Sudhakar M, Kamala RV, Srinivas M (2007) 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. https://doi.org/10.1109/vlsisoc.2007.4402507
Sukwong O, Kim H, Hoe J (2011) Commercial antivirus software effectiveness: An empirical study. Computer. https://doi.org/10.1109/mc.2010.187
van Woudenberg JG, Witteman MF, Bakker B (2011) Improving differential power analysis by elastic alignment. In: CT-RSA 2011. https://doi.org/10.1007/978-3-642-19074-2_8
Vattikonda BC, Das S, Shacham H (2011) Eliminating fine grained timers in xen. In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW'11). https://doi.org/10.1145/2046660.2046671
Walter CD (2003) Longer keys may facilitate side channel attacks. In: International Workshop on Selected Areas in Cryptography. https://doi.org/10.1007/978-3-540-24654-1_4
Xu Y, Cui W, Peinado M (2015) Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P'15. https://doi.org/10.1109/sp.2015.45
Yarom Y, Benger N (2014) Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. IACR Cryptol ePrint Arch 2014:140. https://www.usenix.org/node/184416. https://eprint.iacr.org/2014/140,
Yarom Y, Falkner K (2014) Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium
Zhang Y, Juels A, Oprea A, Reiter MK (2011) HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P'11. https://doi.org/10.1109/sp.2011.31
Zhang Y, Juels A, Reiter MK, Ristenpart T (2012) Cross-VM side channels and their use to extract private keys. In: CCS'12. https://doi.org/10.1145/2382196.2382230

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.