Cache Attacks on Intel SGX

Johannes Götzfried
FAU Erlangen-Nuremberg
johannes.goetzfried@cs.fau.de

Moritz Eckert
FAU Erlangen-Nuremberg
moritz.eckert@fau.de

Sebastian Schinzel
FH Münster
schinzel@fh-muenster.de

Tilo Müller
FAU Erlangen-Nuremberg
tilo.mueller@cs.fau.de

ABSTRACT

For the first time, we practically demonstrate that Intel SGX enclaves are vulnerable against cache-timing attacks. As a case study, we present an access-driven cache-timing attack on AES when running inside an Intel SGX enclave. Using Neve and Seifert's elimination method, as well as a cache probing mechanism relying on Intel PMC, we are able to extract the AES secret key in less than 10 seconds by investigating 480 encrypted blocks on average. The AES implementation we attack is based on a Gladman AES implementation taken from an older version of OpenSSL, which is known to be vulnerable to cache-timing attacks. In contrast to previous works on cache-timing attacks, our attack is executed with root privileges running on the same host as the vulnerable enclave. Intel SGX, however, was designed to precisely protect applications against such root-level attacks. As a consequence, we show that SGX cannot withstand its designated attacker model when it comes to side-channel vulnerabilities. To the contrary, the attack surface for side-channels increases dramatically in the scenario of SGX due to the power of root-level attackers, for example, by exploiting the accuracy of PMC, which is restricted to kernel code.

Keywords

Intel SGX, Cache-timing Attacks, Root-level Attacks

1. INTRODUCTION

We regularly see reports of kernel exploits leading to privilege escalation attacks that enable attackers to get root access to a system. At the same time, not all privileged users can be trusted, e.g., cloud providers cannot be trusted when it comes to the protection of intellectual property or privacy, and end users cannot be trusted when it comes to DRM or the protection against cracking. In consequence of the threat of root-level attacks, there is high demand for a technology that guarantees the confidentiality and integrity of applications running inside untrusted execution environments.

EuroSec'17, April 23 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4935-2/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3065913.3065915

Since 2015, Intel addresses this threat with hardware instructions and provides an architecture extension called *Software Guard Extensions* (SGX) that creates secure containers, so-called enclaves, to protect applications against access from higher privileges, including the OS kernel [1, 7]. Unlike previous solutions, such as the TPM, and similar to Intel TXT, the OS kernel is considered untrusted within the threat model of SGX and no hardware other than the CPU (such as a TPM) is part of the trusted computing base.
Intel's whitepapers about SGX discard cache-timing attacks as unpractical physical attacks [1, 7], not mentioning the case of software-based side-channel attacks. Software-based side-channel attacks, however, are particularly powerful due to Intel's *Performance Monitoring Counters* (PMC), which are restricted to the OS kernel, and according to Intel special care needs to be taken when writing enclave code. In addition to exploiting PMC, root-level attackers have fine-grained control over enclave caches because they are able to enforce a target enclave to run on the same hyperthreading core as the spy process.
The abilities that root-level attacks afford, such as PMC, CPU pinning, and full control over the hyperthreading affinity of a thread, help to overcome difficulties, such as dealing with noise, known from common cache-timing attacks in the literature. Consequently, root-level cache-timing attacks are a promising method of attack against software protected by recent trusted computing architectures such as SGX.

1.1 Contributions

In this paper, we utilize a new method of attack, namely root-level cache-timing attacks, to infer secret information from an Intel SGX enclave. As a case study, we present an access-driven cache-timing attack on AES when running inside an enclave. We used OpenSSL version 0.9.7a as AES implementation, which is already known to be vulnerable to cache-timing attacks. As the method of attacking, we implemented Neve and Seifert's elimination method [8], which is already known to break vulnerable AES implementations. To the best of our knowledge, however, we are the first practically demonstrating that Intel SGX enclaves are vulnerable to cache-timing attacks. In detail our contributions are:

- We describe a root-level cache-timing attack relying on CPU pinning, influencing the hyperthreading affinity, and accessing Intel PMC. CPU pinning as well as accessing PMC are only feasible with superuser privileges and hence, have not yet appeared in the literature for non-SGX scenarios.

- For our case study, we implement a cache-based *Prime&Probe* algorithm that is able to identify memory locations accessed by enclave code on cache line granularity. We support our measurements by Intel PMC within the attacker thread and run both the enclave and the attacker thread on the same hyperthreading core. Thus, enclave and attacker thread also share their memory.
- Using an implementation of Neve and Seifert's elimination method, we are able to extract the AES secret key of an encryption application in less than 10 seconds. We are able to deterministically derive the secret key for every run of the vulnerable application when investigating 480 encrypted blocks on average to reduce the number of key candidates to one.

Our implementation is free software published under the GPL v2 available at https://www1.cs.fau.de/sgx-timing.

1.2 Related Work

At the beginning, Intel SGX was used in publications about cloud computing scenarios. The execution system Haven [2] was designed to securely run unmodified legacy applications on an unmodified cloud operating system to provide applications with the protection of SGX without adaptation. Protecting existing applications, however, means protecting a huge code base which often involves crypto routines. As we show, moving existing applications along with their crypto routines into enclaves is not secure per se but requires reviewing the crypto implementations according to the attacker model of SGX. Another publication using SGX for cloud computing is VC3 [10] which offers distributed *Map-Reduce* computations while keeping the processed data hidden from cloud providers. Even though VC3 is not directly affected by cache-timing attacks on AES, similar attacks might be able to harm the privacy of VC3's processed data.
From an attackers point of view, Costan and Devadas [4] recently presented an analysis that points out weak spots of Intel SGX. The authors hypothesize that SGX does not implement protection measures against side-channel attacks. First of all, that are cache-timing attacks, and second, an untrusted operating system might be able to track memory accesses of enclaves on a per page basis. The second claim has been proven by Xu et. al. [13] who demonstrated controlled-channel attacks on protected applications. Furthermore, an attack against SGX called AsyncShock [12] has been published which targets synchronisation bugs such as use-after-free and time-of-check-to-time-of-use within enclave code by manipulating the scheduler.
During the preparation of the camera-ready version of this paper we became aware of independently developed work which is currently unpublished but closely related [3, 11].

2. BACKGROUND

We base our work on the access-driven approach by Neve and Seifert [8] which is considered a solid attack against the last round of AES. For our attack, we assume we can isolate the last round. The attack then proceeds as follows: The general setup is a known ciphertext attack based on the standard Gladman implementation. Thus, the ciphertext byte $c_{i,j}$ depends on the output of the T-table T_4 and the last round key byte $k_{i,j}^{(10)}$:

$$c_{i,j} = k_{i,j}^{(10)} \oplus \left(T_4[s_{i,(i+j) \bmod 4}]\right)_i$$

Where $(\cdot)_i$ denotes the i-th byte within $(\cdot)$. As XOR is self-inverse, the key byte $k_{i,j}^{(10)}$ can be obtained as follows:

$$k_{i,j}^{(10)} = c_{i,j} \oplus \left(T_4[s_{i,(i+j) \bmod 4}]\right)_i$$

Because the AES key schedule is redundant, the bytes $k_{i,j}^{(10)}$ are sufficient to obtain the bytes $k_{i,j}$ and thus the encryption key k. However, we need knowledge about the ciphertext and the result of the table lookups for T_4. For our attack we assume to have access to the ciphertext (known ciphertext attack) and retrieve the result for T_4 with the cache-timing attack.
The *Prime&Probe* method works on the last round as one entity and consequently we have to find out which accessed cache line correspond to which byte of the ciphertext as well as which byte within this line was accessed. Of course it is difficult to deduce what byte has been accessed exactly and that is why Neve and Seifert proposed their elimination method where certain candidates for each key byte are excluded:

$$k_{i,j}^{(10)} \notin c_{i,j} \oplus \neg[T_4 \text{ outputs}]$$

Here $\neg[T_4 \text{ outputs}]$ refers to all T_4 byte values contained within non-accessed cache lines. The basic idea is to discard all candidates for $k_{i,j}^{(10)}$ which would map to not accessed cache lines and thus cannot be candidates given the corresponding ciphertext byte $c_{i,j}$. To be more precise, let Φ be the set of all possible key byte values. When the operation starts Φ contains all possible byte values, i.e., $\Phi = \{i : 0 \leq i \leq 255\}$. The goal is to reduce those values such that only the original key byte $\Phi = \{k_{i,j}^{(10)}\}$ remains. To this end, we issue a couple of encryptions with different inputs while monitoring the cache activity. For each encryption we can eliminate some values $\Phi_e = c_{i,j} \oplus \neg[T_4 \text{ outputs}]$ and thus, reduce the size of the set Φ by assigning $\Phi \leftarrow \Phi \backslash \Phi_e$. For more details and examples on how the elimination method works, we refer to the original paper [8].

3. DESIGN AND IMPLEMENTATION

In this section, we want to give an in-depth description on how we designed and implemented our attack. We first examine the memory cache architecture of our target system (Sect. 3.1) and then give an explanation of the priming and probing implementation (Sect. 3.2). Afterwards, we present the implementation of the elimination method (Sect. 3.3) and finally describe our attack setup in detail (Sect. 3.4).

3.1 Cache Architecture

Our Intel i7-6700HQ CPU has four physical cores, each one has a separate L1 and L2 cache, as well as a shared L3 cache. The L1 cache is split into data and instruction cache, both have a size of 32 KB and are 8-way associative with 64 byte cache lines. This means a 64 bit address is split into $b = log_2(64)$ bit $= 6$ bit offset, $s = log_2(32\text{KB}/(64\text{B} \cdot 8))$ bit $= 6$ bit set and $t = 64$ bit $- (b + s) = 52$ bit tag.
For this specific cache topology it does not matter whether the cache is indexed physically or virtually, because 4096 byte pages require an offset of $log_2(4096)$ bit $= 12$ bit to address bytes within one page. These 12 bit correspond to the cache set s and the cache line offset b and thus, s and b keep the same value regardless of whether the physical or virtual address is used for indexing within the cache.

3.2 Cache Priming and Probing

In order to gain the knowledge about accessed memory addresses on cache line granularity we utilize the *Prime&Probe* method [9]. While the theory behind the *Prime&Probe* algorithm is fairly simple, associative caches introduce another challenge to this problem, because Neve and Seifert describe their attack on a direct mapped cache (cf. Sect. 2).

Unfortunately, direct mapped caches are not used by modern Intel CPUs and we have to take a closer look at the placement of the Gladman T-tables in memory. In the case of our 8-way associative L1-cache we have 8 cache lines per cache-set. This has two effects on our algorithm. First of all two addresses having the same set bit, but belonging to different cache lines, can be stored concurrently in the same cache set. Ultimately, this makes an access to one of them indistinguishable from the other, on the cache set level of our the probing step. Secondly, in the priming phase we have to access each cache set 8 times to be sure we filled the whole cache and force an eviction of our own data by the victim application, as the cache line position cannot be predicted. In fact, it could be, if the eviction algorithm was known and deterministic, but unfortunately this is not the case. If two cache lines of the T_4 table were stored in the same cache set, they would be unusable for the elimination method, because if one of them is accessed, the access would be indistinguishable from an access to the other cache line.

A single Gladman T-table has a size of 1KB and our L1-cache has a cache line size of 64B, so placing the T_4 table in the cache requires 16 cache sets assuming that there are no collisions, i.e., the sets are different for each cache line. A collision, however, is highly unlikely, because the T-tables lie contiguously within memory. Furthermore, in our 8-way associative 32KB L1-cache there are $32\text{KB}/(64\text{B} \cdot 8) = 64$ cache sets, hence the table fits completely into the cache. Consequently, we can use the information about an access to a cache set similarly compared to an access to a cache line for direct mapped caches. Nevertheless, we cannot predict which cache line in each set holds the T-table data. Thus, our priming algorithm has to access every cache set 8 times using particularly aligned addresses to fill every line and eventually detect every single eviction.

This leads to the next challenge we need to solve: Identifying an eviction in the probing phase. The original idea of Neve and Seifert is to distinguish an evicted line from a present one by the timing difference between an L1 and L2 access. The Intel *Performance Monitoring Counters* (PMC) [6], however, give us even more accurate information compared to relying on any time-stamp counter. With the PMC, Intel introduced a high-level interface for developers to gain access to several CPU internal performance metrics. This data is obtained from the so-called *Performance Monitoring Units* (PMU) and contains information about L1-, L2-, as well as L3-cache misses. The shortcoming of using the PMC to detect cache misses is the fact that the counters need to be started from privileged kernel space, i.e., ring zero, however they can be read from user space afterwards.

Local attackers with superuser rights are covered by the SGX attacker model and thus, an attacker is allowed to run ring zero code by simply loading a kernel module. We did not write the PMC driver ourselves, but instead we use the *Loadable Kernel Module* (LKM) from Agnar Fog's PMC based performance test suite [5]. After starting the PMC with the desired counters we can read the current count with the `readpmc` assembler instruction from non-enclave user space with the desired PMC identifier as argument. This provides us with a perfect measurement for L1-cache hits or misses. Using the PMC, our probing algorithm works as follows:

1. Read the PMC count using the `readpmc` instruction.
2. Force the CPU to serialize all instructions by dispatching a dummy `cpuid` instruction.
3. Access the desired cache line using a particularly aligned address.
4. Serialize all instructions again (dummy `cpuid` instruction).
5. Read the PMC count (`readpmc` instruction) and return the difference to the last read PMC count.

If the difference is greater than zero, the particular cache line was evicted from the L1-cache, otherwise it was present. This procedure is repeated 8 times again to be able to catch all evictions for a given cache set. If one difference is greater than zero, it can be concluded that the corresponding cache line for T_4 has been accessed. The serializations with the `cpuid` instruction are necessary to prevent out-of-order execution in modern CPUs which would tamper with our results.

3.3 Elimination Method

One advantage of Neve and Seifert's elimination method is that it is resilient against false positives. The probing phase provides us with information about the accessed cache lines during the last round. Theoretically we could perform a probing request successively to each cache set one after the other. This way we would receive the access pattern of a ciphertext with a single encryption. Unfortunately, we encountered that accessing certain cache lines of the T-table has influence on other subsequent cache lines of the T-table, which get preloaded in their particular set. Consequently, we decided to probe each cache set separately for each ciphertext, which results in 16 encryptions per ciphertext until we receive the whole access pattern.

While this would be sufficient for the elimination algorithm, we additionally decided for fault-tolerance. In addition to probing each cache line once, we provide a possibility to repeat the overall process a certain number of times and to count each line evicted more often than a certain threshold as accessed line. The repetition rate and the threshold are configurable and can be adapted by the attacker. If each cache line is only probed once, i.e., no repetition, no threshold is used as well. While smaller repetition rates result in less encryptions needed to receive the secret key, higher ones cause less fluctuation in encryptions needed per ciphertext. We decided to set a repetition rate of 20 and a threshold of five as default values if fault-tolerance is enabled.

Whenever a cache line has been evicted less often than the threshold value and is consequently counted as not accessed, all corresponding key byte candidates are removed from the remaining possibilities for every key byte according to Neve and Seifert's algorithm. This is repeated until only one value remains for each byte which immediately leaks the secret last round key. Afterwards, the fact that Rijndael's key schedule is reversible is used to calculate the secret key based on the last round key.

3.4 Attack Setup

Our attack design consists of one application with two threads pinned to two logical CPUs sharing the same physical core

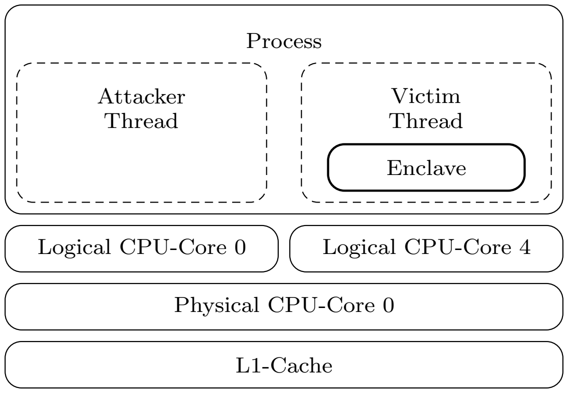

Figure 1: Our attack setup consisting of one application with two threads pinned to two logical CPUs sharing the same physical core and L1-cache.

as shown in Figure 1. The victim thread runs an OpenSSL AES implementation which is known to be vulnerable to cache-timing attacks within an SGX enclave. The encryption key is generated within this enclave and never leaves it. The attacker thread executes regular ring 3 code in non-enclave mode. It takes care of priming, probing, and the elimination by reading and comparing PMC values.
Furthermore, communication is exclusively performed using shared memory, no process context switches occur, and the enclave is never exited. This setup combines specific possibilities of root-level attackers within one single root-level cache-timing attack. It demonstrates that long-term secrets within enclaves that would usually never be exposed to the non-enclave world can be obtained using cache-timing attacks. Our implementation aims to serve as a case study and shows that software developers writing enclave code need to take particular care when it comes to crypto implementations.

Communication with Shared Memory.

To demonstrate cache-timing attacks, often a classic client-server application is deployed in such a way that the two involved processes are pinned to the same CPU core. When porting this setup to SGX enclaves we would need to handle the communication outside of the enclave, because certain instructions such as system calls are not allowed within enclaves. System calls, however, are needed to communicate between the client and the server such that the enclave has to be exited first. Unfortunately, this setup implies using *ECALLs* and *OCALLs* which introduce too much noise, at least when used with the official Intel SGX SDK. In fact, they evict the whole T_4 table when triggering the encryption function after the priming step which makes the detection of accessed cache lines in the probing phase impossible.
When designing our victim application, we came to the conclusion that it is not possible to perform a cache-timing attack solely focusing on the L1-cache, whenever an ECALL or OCALL is included between the measurements. This left us with two options. First, we could focus on lower cache levels. Second, we can build our victim application in a way that it does not have to leave enclave mode during the measured section. While the first option is an interesting path to pursue in future work, we decided to continue with our attack focused on the L1-cache. We solved the communication problem inside the enclave mode by allocating and using shared memory and utilizing control flags. Our attacker process copies the particular plaintext to the shared memory and triggers the encryption process by setting a particular flag. The enclave is actively waiting for this flag to be set and immediately starts the encryption process. Afterwards, it sets a flag itself to signal the end of the encryption. The attacker already waits for the encryption to finish and reads the ciphertext from the shared memory. Finally, it continues with the next plaintext. This setup allows the attacker to perform the priming right before and the probing right after the encryption. Shared memory is a well-known and fast communication technique which can be used between SGX enclaves and unprotected code as well. The control flags are used to reduce the synchronisation effort. Finding the right timing for a regular encryption loop, would lead to the same results, but complicate our setup.

Attacker and Victim Thread.

Using two separate processes for attacker and victim still does not allow us to observe the cache activity. The reason for that results in the fact that the L1-cache operates on virtual addresses. Whenever a process switch occurs, the CPU has to replace all the memory abstractions of the current process with the ones of the next process. This includes the page tables which are necessary for translating virtual addresses to physical ones. In order to set the correct page tables, the CPU has to update the `CR3` register which contains the base address of the current page directory. Unfortunately, changing the page tables invalidates all information connected to virtual addresses. Consequently, a write to `CR3` will trigger a *Translation Lookaside Buffer* (TLB) and L1-cache flush, as well as flushing all other virtually tagged caches. In our scenario, the L1 cache would be flushed every time we switch from the attacker to the victim or vice versa occurring after the priming step and thus, destroying our probing results.
Consequently, we changed our scenario to have the attacker in the same process as the victim, but within two different threads. To this end, we pinned two kernel-level threads on the same core using the *PThread* threading library. In our implementation the attacker enters the victim enclave within a separate thread in the process and again uses shared memory for communication. Additionally, we eliminated every system-call in between the encryption and probing phase, forcing both parties to perform active waiting. Note, however, that in our SGX scenario the enclave contents are the only parts being protected, and hence, manipulating the surrounding application or the operating system is covered by the strong attacker model of SGX.

HyperThreading.

Having two threads pinned on the same CPU did not provide us with the measurements we expected either. The reason is that whenever the kernel switches between the execution of the two threads, the CPU has to enter or leave the victim enclave which, although less expensive than an enclave exit triggered by software through an ECALL, still causes enough evictions in the L1-cache to evict the whole T_4 table and to ultimately destroy our probing measurements. In fact, this leads to a contradiction: In order to work on the same L1-cache we need to operate both threads on the same core and simultaneously they need to operate on different cores to omit the enclave exit. At a first glance, it seems impossible to satisfy both requirements, but luckily modern Intel CPUs include a feature which helps to indeed satisfy both requirements, called *HyperThreading*. With *HyperThreading* usually two logical CPUs per physical core are offered, for example, by duplicating the *Arithmetic Logic Unit* (ALU) but sharing everything else including the L1-cache. Hence, we decided to pin the attacker thread and the victim thread running in enclave mode to two different logical CPUs which are mapped to the same physical core. This way enclave exits are omitted while still sharing the same L1-cache between both threads. With superuser privileges controlling the hyperthreading affinity, i.e., pinning the threads to appropriate logical CPUs, is easily possible, for example, by using the `sched_setaffinity()` system call.

Enclave Interaction.
To plausibly demonstrate that our implementation is able to extract unknown secrets from enclaves, our victim enclave generates an AES key with the help of `sgx_read_rand()` provided by the SGX SDK *within* the enclave. Additionally, the victim enclave contains functions to safely store and load this key afterwards by utilizing SGX's sealing mechanism. Furthermore, it contains a function for encrypting single blocks as well as an endless encryption loop. In the actual attack, the attacker thread creates random plaintext and triggers the encryption inside the enclave, which runs just until the start of the last round and waits for the next flag. In the next step it issues its priming algorithm which fills all the T_4 table's cache lines. After priming, the attacker thread sets the particular flag, in order to signal the execution of the last round to the enclave. When the encryption has finished, it continues the attack, by issuing the probing algorithm and feeding the results to the elimination method. Those steps are repeated until all key bytes have been reduced to one remaining possibility. Finally, the attacker thread verifies the result by creating a decryption key, which is used to decrypt a string encrypted by the enclave beforehand. Note, that the AES key is never leaving the secured enclave and without the attack would not be accessible besides within the victim enclave itself.

4. EVALUATION
All attacks have been executed on a notebook with a 2.60GHz Intel Core i7-6700HQ CPU and 16GB of RAM, running Ubuntu Linux 14.04 LTS (Trusty Tahr). We evaluate our implementation regarding performance (Sect. 4.1) and practicability with respect to real-world scenarios (Sect. 4.2).

4.1 Performance
In order to measure the performance of our attack, we take two different scenarios into account. For each scenario, we perform 5000 attacks with different keys and let each attack run until the respective key can be retrieved. To evaluate the performance we measured for each attack the overall time required to leak a secret key, as well as the amount of required elimination rounds, i.e., how many encryptions and thus ciphertext blocks are needed.
For the first scenario, we perform the eviction detection with a threshold of 25% by repeating the probing step for every cache line 20 times and handling every line which has been evicted more than five times as an accessed one. On average, this approach requires 30 elimination rounds to reduce the number of possible key candidates down to one. With each round requiring $20 \cdot 16 = 320$ encryptions, the attack needs an average number of 9600 encryptions to leak the secret key. On our target system this takes an average time of 5 minutes and 29 seconds. Figure 2 shows the average number of remaining key candidates after each elimination round. It can be seen that the number of possible key candidates decreases almost exponentially and after approximately 15 rounds only half of all possible key candidates remain. Furthermore, the minimum and maximum number of remaining candidates shows that our average number of candidates stays close to the minimum. Although the elimination method is not prone to false positives, the threshold helps to prevent arbitrary evictions to be included in the result. Consequently, the deviations for the number of remaining candidates per elimination round are rather low compared to the next scenario.
For the second scenario, we measure the performance without any threshold, i.e., we only probe once per elimination round. Interestingly, on average this setup requires the same amount of elimination rounds, but due to the reduced probing frequency, only $16 \cdot 30 = 480$ encryptions are necessary for all elimination rounds together. Consequently, the time to leak the secret key is drastically reduced down to only 10 seconds on average. When comparing the results from Figure 3 to the previous scenario, however, we observe a significant increase in the deviation of remaining candidates per elimination

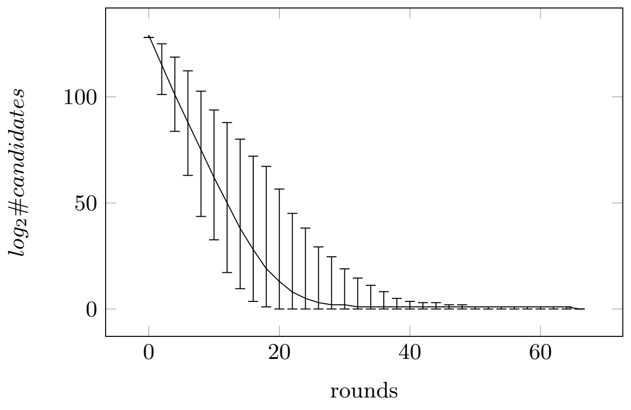

Figure 2: Number of remaining key candidates after each elimination round with 25% threshold.

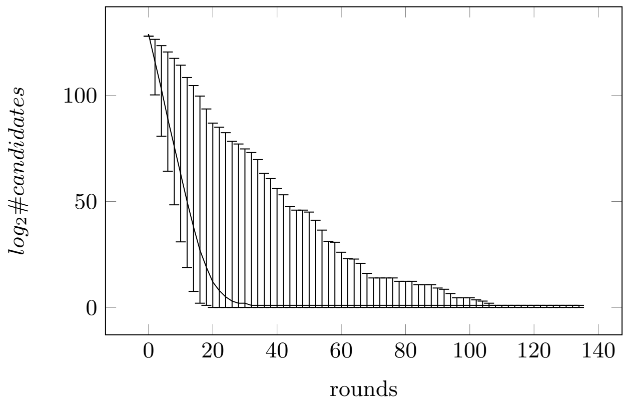

Figure 3: Number of remaining key candidates after each elimination round without threshold.

round. This means that it is more likely that without a threshold a larger amount of elimination rounds is needed for certain secret keys. Nevertheless, the average number of elimination rounds remains roughly constant which means that faster results are usually achieved without threshold.

4.2 Practicability

The victim enclave is currently run in debug mode. While the functionality SGX provides in debug mode is similar to production mode, certain security features are disabled. In production mode, Intel provides the possibility to suppress performance monitoring activities by entering a so-called *opt-out enclave* (see Sect. 43.6 of the Intel SDM [6]) which sets the *Anti Side-channel Interference* (ASCI) bit within the `IA32_PERF_GLOBAL_STATUS` MSR. If this bit is set, the PMC are not accumulated normally, but instead activities by the enclave are not counted. While we could not test the effect of this bit on our attack, we are confident that it does not affect the attack. The reason is that we first prime the cache, then execute the last round of AES within the enclave, and finally probe the cache from the attacker thread *outside* of the enclave. The PMC are only used during the probing phase, i.e., the last phase, and only from the attacker thread (see Sect. 3.2). Consequently, even if the ASCI bit is set and the cache activities by the victim enclave are not counted, the activities by our attacker thread are counted. This is based on the assumption that for two threads running in parallel on a single core (hyperthreading case) the ASCI bit does not affect counting activities of the core running non-enclave code.

For our implementation, we use a Gladman AES implementation adopted from an old version of OpenSSL (version 0.9.7a) known to be vulnerable against certain types of cache-timing attacks. As our goal is to demonstrate that SGX enclaves in general are indeed vulnerable to cache-timing attacks, this choice is reasonable.

Interestingly, the AES implementation within the official Intel SGX SDK for Linux is not making use of AES-NI and implements AES in software instead, even though AES-NI instructions would be usable within enclaves. The software implementation used is almost a textbook version of AES without the use of Gladman tables. Intel manually included instructions, though, which perform an access to all S-Box indices in each round causing cache evictions for all S-Box entries. Consequently, Neve and Seifert's elimination method is not applicable to the AES implementation from the SGX SDK either, because there are no cache lines which have not been accessed and would allow to eliminate key byte candidates. However, the question is raised, why Intel did not just use AES-NI when they apparently were aware of the threat through cache-timing attacks.

5. CONCLUSION

In this paper, we demonstrated an access-driven cache-timing attack against an AES implementation running within an SGX enclave. In particular, our implementation is able to derive the secret key of an AES Gladman implementation taken from OpenSSL. The attack deterministically retrieves the key within an average time of less than 10 seconds.

Our work stresses the difficulties of protecting sensitive information such as encryption keys on untrusted platforms. Although SGX is often expected to secure an application against all kinds of software or hardware attacks with only the CPU package considered trusted, this is not true for cache-timing attacks and forces developers to protect their applications themselves.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

6. REFERENCES

[1] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), p. 10.

[2] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014), pp. 267–283.

[3] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. arXiv:1702.07521, Feb. 2017.

[4] Costan, V., and Devadas, S. Intel sgx explained. Tech. rep., Cryptology ePrint, Report 2016/086.

[5] Fog, A. Test programs for measuring clock cycles and performance monitoring. http://www.agner.org/optimize/#testp, Sept. 2016.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 325462-061us ed., 2016.

[7] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[8] Neve, M., and Seifert, J.-P. Advances on access-driven cache attacks on aes. In *International Workshop on Selected Areas in Cryptography* (2006), Springer, pp. 147–162.

[9] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA, San Jose, CA, USA, Proceedings* (2006), pp. 1–20.

[10] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 38–54.

[11] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. arXiv:1702.08719, Feb. 2017.

[12] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security* (2016), Springer.

[13] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy* (2015), pp. 640–656.