Side-Channel Attacks on Intel SGX: How SGX Amplifies The Power of Cache Attacks

by

Ahmad Moghimi

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

April 27, 2017

APPROVED:

Professor Thomas Eisenbarth, Major Thesis Advisor

Professor Robert Walls, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

In modern computing environments, hardware resources are commonly shared, and parallel computation is more widely used. Users run their services in parallel on the same hardware and process information with different confidentiality levels every day. Running parallel tasks can cause privacy and security problems if proper isolation is not enforced. Computers need to rely on a trusted root to protect the data from malicious entities. Intel proposed the Software Guard eXtension (SGX) to create a trusted execution environment (TEE) within the processor. SGX allows developers to benefit from the hardware level isolation.

SGX relies only on the hardware, and claims runtime protection even if the OS and other software components are malicious. However, SGX disregards any kind of side-channel attacks. Researchers have demonstrated that microarchitectural side-channels are very effective in thwarting the hardware provided isolation. In scenarios that involve SGX as part of their defense mechanism, system adversaries become important threats, and they are capable of initiating these attacks.

This work introduces a new and more powerful cache side-channel attack that provides system adversaries a high resolution channel. The developed attack is able to virtually track all memory accesses of SGX execution with temporal precision. As a proof of concept, we demonstrate our attack to recover cryptographic AES keys from the commonly used implementations including those that were believed to be resistant in previous attack scenarios. Our results show that SGX cannot protect critical data sensitive computations, and efficient AES key recovery is possible in a practical environment. In contrast to previous attacks which require hundreds of measurements, this is the first cache side-channel attack on a real system that can recover AES keys with a minimal number of measurements. We can successfully recover the AES key from T-Table based implementations in a known plaintext and ciphertext scenario with an average of 15 and 7 samples respectively.

Acknowledgements

I would like to express my deepest acknowledgement to my advisor, Prof. Thomas Eisenbarth, for his professional tutoring and close collaboration during this research project. His support during every step of this work made this master thesis less challenging and more enjoyable. It was a great experience, and I am honored to continue to work with him.

I also want to thank my thesis reader, Prof. Robert Walls, for his time and attention and notable comments on the authorship of this work. I would like to thank my department head, Prof. Craig Wills, for his support on pursing this valuable degree.

I would like to give my kindest regards to Gorka Irazoqui Apecechea, Joshua Pritchett, Curtis Taylor, Hang Cai, Amit Srivastava, Berk Gulmezoglu, Mehmet Sinan Inci, Mohammad Essedik Najd and Okan Seker for creating a friendly and joyful environment here in WPI.

I would like to thank my father Mohammad-Hassan, my mother Zahra, my brother Amir and my sister Maryam. They have always supported me throughout the course of life. At the end, thank you to Nathalie Majcherczyk, for all of her love and support.

Part of this work are currently under submission at a peer-review venue, with a preprint available as *CacheZoom* [42].

Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
- 1.2 Contribution . . . 8
- 1.3 Outline of the work . . . 9

2 Background & Related Work **10**
- 2.1 Intel SGX . . . 10
  - 2.1.1 Memory Encryption Engine (MEE) . . . 11
- 2.2 Cache Memory . . . 12
  - 2.2.1 Cache Addressing Schema . . . 14
  - 2.2.2 Cache Coherency and Replacement policy . . . 14
- 2.3 Microarchitectural Attacks in SGX . . . 15
  - 2.3.1 Prime+Probe . . . 16
- 2.4 Side-Channel Attacks & SGX . . . 17

3 Cache Side-Channel Attack on SGX **20**
- 3.1 Attack Design . . . 20
  - 3.1.1 Adversarial Model . . . 20
  - 3.1.2 Cache Partitioning & Noise Reduction . . . 21
  - 3.1.3 Step by Step Execution . . . 22
  - 3.1.4 Summary . . . 23
- 3.2 Attack Implementation . . . 23
  - 3.2.1 Eviction Set . . . 24
  - 3.2.2 Noise Reducing Configuration . . . 25
  - 3.2.3 Malicious Kernel Driver . . . 27

4 AES in Practice **28**
- 4.1 T-Table Implementations . . . 29
- 4.2 Non-vulnerable Implementations . . . 30
- 4.3 Table Prefetching as a Countermeasure . . . 30
- 4.4 Susceptibility to Cache Attacks . . . 31

5 AES Key Recovery Attack **33**
5.1 Channel Measurement 33
5.1.1 Experimental Setup 33
5.1.2 Eviction Set Quality 34
5.1.3 Channel Bandwidth 35
5.2 AES T-Table KPA Attack 36
5.2.1 Cache Prefetching 39
5.3 AES T-Table KCA Attack 40
5.3.1 Last Round Attack on T-Table 40
5.3.2 Last Round Attack Improvement 41
5.4 AES S-Box KCA Attack 43

6 Conclusion **47**

List of Figures

2.1 Enclave Memory Access 11
2.2 Enclave Attack Surface 12
2.3 MEE Operation 13
2.4 Memory hierarchy of a modern computer 14
2.5 L1D Cache Addressing Schema 15

3.1 Isolated Core-private Channel 22
3.2 Initialized pointers to chase 24

4.1 Prime and Probe Prefetching Timeline 31

5.1 Eviction cycle count 34
5.2 Average Eviction Noise 34
5.3 Cache Hit Map 35
5.4 Cache Hit Map (Filtered) 35
5.5 AES Memory Footprint 37
5.6 First round attack key recovery success rate. 38
5.7 First round attack correct recovered key bits. 39
5.8 AES Memory Footpring with Prefetching 40
5.9 Last round attack recovered key bits. 41
5.10 Last round attack key recovery success rate. 42
5.11 Correlation between observed cache accesses and predicted accesses . 44
5.12 Correlation over key value 45

List of Tables

2.1 Comparison of different side-channels against enclave 16

5.1 Statistics on recovered memory accesses for T-table implementations. 39

Chapter 1

Introduction

1.1 Motivation

In the world of parallel and distributed computing, processes with various trust and criticality levels are allowed to run at the same time and share system resources. Proliferation of cloud computing technology elevated these phenomena to the next level. Cloud computers running many different services authored by various providers process user information on the same physical hardware.

Traditionally, the operating system (OS) provides security services such as access control, encryption and authentication to isolate concurrent tasks from interfering with data and computation of other tasks. The dependence of various computation units i.e., threads, processes, etc. on a privileged root of trust realizes the concept of Trusted Computing Base (TCB) on the OS. In cloud computing, hypervisor software and cloud service providers also become part of the TCB. High complexity of modern computing and variety of attack surfaces make it unachievable to keep an entire system secure and trusted [37, 22].

Minimizing the size of TCB on the software is practically ineffective against attacks targeting the privileged mode of execution. There are numerous system services, drivers and kernel interfaces vulnerable to privilege escalation attacks. Third parties also contribute to this weakness by adding new features and extending the kernel size. One workaround is to outsource security-critical services to a Secure Element (SE) which is a separate hardware. Trusted Platform Modules (TPM), for example, provide services such as cryptography and secure boot beyond the authority of the OS [43].

Trusted Execution Environments (TEE) as an alternative to TPM attempt to provide similar services directly within the CPU. A TEE is an isolated environment to run software with a higher trust level than the OS. The software running inside a TEE has full access to the system resources while it is protected from other applications and from the OS. Examples include ARM TrustZone [3] and Intel Software Guard eXtension (SGX) [54]. Software solutions to virtualize or emulate TPM-like features on commodity hardware have been proposed. However, they suffer from

security and usability challenges [17, 30, 48, 58].

Intel SGX provides a TEE on an untrusted system by only trusting the hardware on which the code is executed. The runtime execution is secured inside an *enclave* which is encrypted and authenticated by the processor. In particular, the CPU decrypts and verifies the authenticity of an enclave code and data as it is moved into the processor cache. Enclaves are logically protected from the OS and malicious applications, and physical adversaries monitoring system buses. However, Intel SGX is not protected against attacks that utilize hardware resources as a covert channel [33]. And indeed, first works demonstrating that microarchitectural side channels can be exploited have been proposed, including attacks using page table faults [62], the branch prediction unit [38], and caches [51, 11].

Caches have become a very popular side channel in many scenarios, including mobile [39] and cloud environments [31]. More specifically, last level cache (LLC) attacks perform well in cross-core scenarios on Intel machines. Cache side channels provide an information leakage channel with the granularity of a cache line size. The small line size over the wide range of memory addresses has a high *spatial* resolution. This *spatial* resolution becomes very advantageous for the adversaries. For example, with a line size of only 64 bytes, attackers are able to distinguish accesses to different RSA multiplicands for windowed exponentiation algorithms [40]. This high spatial resolution, combined with a good temporal resolution, have enabled attacks on all major asymmetric cryptographic implementations.

For symmetric cryptography, the scenario is more challenging. An optimized table-based AES implementation can be executed in a few hundred cycles, while performing a single cache access time measurement on an LLC set is computationally more expensive. To avoid undersampling, synchronized attacks trigger a single encryption and perform one measurement, yielding at best one observation per encryption [7]. This heavily limits the observed information to perform key recovery attacks.

1.2 Contribution

Our study explores practicality of cache side-channel attacks on SGX. This increases the awareness of security community on an important threat affecting TEE platforms and in particular SGX. In addition, application of the attack on implementations of a cryptographic algorithm like AES shows that there exist weaknesses in the software. Some of these weaknesses can only be exploited in certain security scenarios.

In this work, we demonstrate not only that Intel SGX is vulnerable to cache based attacks, but also that the quality of information leakage channel is significantly high in the SGX environment. The resolution of our channel enables attacks that were infeasible in previously exploited scenarios, e.g., cloud environments or smartphones. In particular, we take advantage of the capabilities that SGX assumes an attacker has, i.e., access to the OS resources. We construct a malicious OS kernel

module that periodically applies the popular `Prime and Probe` attack [46] in the local cache of physical processor core, i.e., core-private cache, thereby recovering high-resolution information about the memory addresses that the target enclave accesses. By default, the OS tries to maximize the resource usage by spreading the tasks among all the cores which reduces the applicability of core-private channels. On the contrary, as the compromised OS schedules only the victim and attacker in the same core, the usage of core-private resources is prone to the system activity noise.

Tracking memory accesses of an enclave with high temporal (precision over the execution time) and spatial (precision over the memory space) resolution can be exploited in many application scenarios, i.e., any security data driven computation. We attack several AES implementations running inside the SGX enclave to demonstrate the power of our proposed side channel setup. In particular, we take advantage of our high resolution channel to construct more efficient key recovery attacks against several AES implementations, including those exploited in previous works [35] and those that are impossible to exploit without a high spatial resolution channel. Furthermore, we show that cache prefetching and S-box implementations as countermeasures are ineffective to prevent key recovery attacks, and they facilitate the memory trace extraction in some cases. In brief, this work:

- Presents a powerful high-bandwidth covert channel implemented through the core-private cache by exploiting several system resources in lieu of a compromised OS. The covert channel can be applied against TEEs to recover fine grained information about memory accesses, which often carry sensitive information.
- Demonstrates the high quality of our covert channel by recovering AES keys with less traces than in previous practical attacks, and further, by attacking implementations considered resistant to cache attacks.
- Shows that some of the countermeasures that were supposed to protect AES implementations, e.g. prefetching and S-box implementations, are ineffective in the context of SGX. In fact, prefetching can even be helpful to the attacker.

1.3 Outline of the work

In Chapter 2 we start by discussing the background information on Intel architecture and particularly SGX, Micro-architectural Side-Channel Attack and `Prime and Probe` . Then, we introduce the design of our side channel in Chapter 3 and discuss other methods to create low noise channels in adversarial operating system scenarios. After explanation of the implementation of our attack, in chapter 4, we talk about AES and its various common implementations. We show the results of our attack on these implementation in chapter 5, and conclude the work in chapter 6.

Chapter 2

Background & Related Work

Intel introduced some modifications to its processors released after the Skylake generation, and proposed a new programming model to make hardware based TEE available on commodity processors. However, SGX is a compatible extension based on the Intel architecture and its core has remained untouched. Significant modifications include integration of a memory encryption engine (MEE) hardware and support for a new set of processor instructions designed for enclave specific operations. On top of that, previous concepts that apply to cache memory and cache based timing side-channels are still applicable.

This chapter covers topics that help understand the side channel used in our key recovery attacks against SGX. We discuss the basic functionality of Intel SGX and possible micro-architectural attacks that can be deployed. After a brief discussion of Intel cache architecture, we finally explain the main procedure of the `Prime and Probe` technique which is used as part of our attack. Finally, we discuss the related work in the literature.

2.1 Intel SGX

SGX is a new subset of CPU instructions. It allows execution of software inside isolated environments called *enclaves*. Enclaves are user-level modules isolated from other software components running on the same hardware, including compromised OSs. SGX has recently gained popularity among the research community and various security applications have been proposed [4, 5, 12, 44, 50, 53, 24].

Enclave modules are developed as the trusted components of an application, and can be shipped and be loaded by untrusted components. The untrusted components interact with the system software, and the system software dedicates pages from a trusted memory region for the enclave by executing specific privileged instructions. The trusted memory region known as Enclave Page Cache (EPC) is configured through system BIOS and preallocated at boot time. After that, during the execution of the user application, the authenticity, integrity and confidentiality

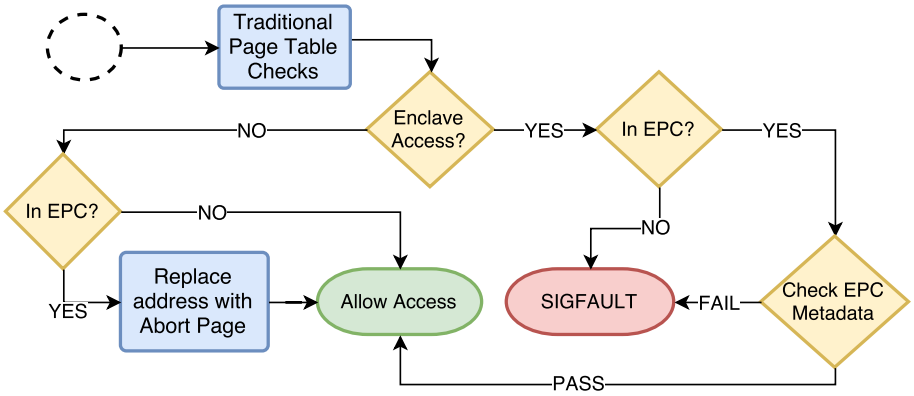


Figure 2.1: Enclave Memory Access: After the traditional page table checks, the processor verifies if memory access to the trusted memory region should be permitted

of a running enclave are provided and measured by the processor.

Any untrusted execution, including the OS, has no control over the trusted memory region. Untrusted applications can only use specific instructions to call trusted components through predefined interfaces. Memory accesses to an address within the EPC range goes through additional vetting, and the processor blocks unwanted memory accesses trying to access EPC pages. Figure 2.1 shows the enclave specific memory access checks.

In summary, SGX assumes only the hardware to be trusted; any other agent is considered susceptible of being malicious. Although, the (potentially) compromised OS is in charge of the memory page mapping, SGX detects any malicious mapping performed by it. In fact, any alteration of the enclave pages from the OS is stored by SGX and is verifiable by the third party agents. This design helps developers to benefit from the hardware isolation for security critical applications. Figure 2.2 differentiate the attack surface of enclave trusted modules and other untrusted components.

Apart from the processor level access control that protects enclave pages from software adversaries snooping on them, SGX is also designed to protect enclaves from malicious physical adversaries monitoring system buses. When the operating system requests from the processor to dedicate EPC pages and map the enclave module, those pages in DRAM memory are encrypted and signed by the hardware. Pages are only decrypted when they are moved to the cache and their signatures are verified at the same time. MEE which resides between DRAM and LLC is responsible for these cryptographic operations.

2.1.1 Memory Encryption Engine (MEE)

MEE is designed as a subcomponent of Memory Controller (MC) [27]. An internal cache that accommodates a small but fast memory is the first place for processor

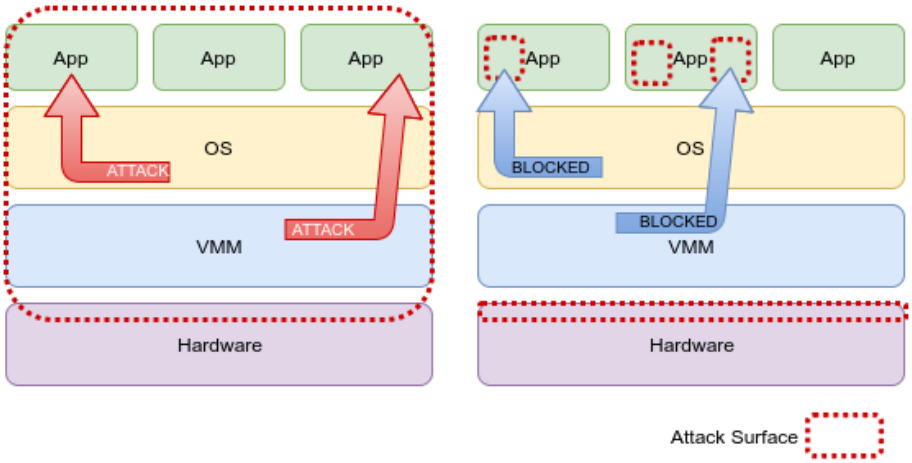


Figure 2.2: **a)** Left: attack surface without enclave. **b)** Right: Attack surface of enclave modules with Intel SGX technology

to resolve memory transactions. If the transaction can not be serviced by cache, it goes to MC. MC only utilizes cryptographic features provided by MEE to resolve memory accesses to the trusted region. The trusted region consists of EPC and its metadata which holds the mapped enclave pages and an integrity tree. In addition to the encryption/decryption of the cache lines on write/read operation from DRAM, MEE initiates additional transactions to update/verify the authentication tags with respect to the integrity tree. The root of trust for the MEE integrity tree is stored on an on-die SRAM which is not accessible outside of the processor package. Figure 2.3 illustrates MEE operations.

Note that, the aforementioned cryptographyic operations use two separate keys for authentication and encryption. These keys are generated randomly at boot from a random unique 256-bit entropy hardcoded into the processor, and they never leave the processor trusted boundary. Any modifications of the integrity tree to perform active attacks are detected at runtime and cause the processor to halt. The potential active (physical) adversary can not avoid the halt and reboot of the machine. Afterward, the processor has to generate new random keys.

2.2 Cache Memory

Modern computers use a hierarchical memory model with a different capacity and speed at each level. In this hierarchy, registers are the closest and fastest, but most expensive memories available to the processor. Most of the registers (depending on their role) are directly accessible from the software stack, i.e., OS, Compiler, etc. As processors have a limited number of registers, DRAMs are the next programmable memory in this hierarchy. The processor, with the help of the OS, maps memory pages from disk and other storage peripherals to DRAM. These peripherals and

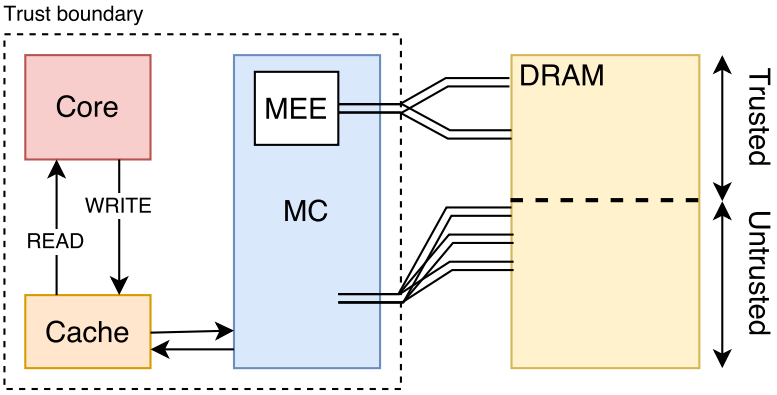


Figure 2.3: MEE Operation: Transactions to the trusted region are issued with MEE involvement

disks are significantly slower compared to DRAM memory.

In this hierarchy, there is still a gap between the processor and DRAM memory speed. To avoid high latency, due to the data dependent instructions, modern processors take advantage of some levels of internal cache memory. In particular, Intel processors use three levels of cache known as the L1, L2 core-private caches and the L3 or Last Level Cache (LLC). LLC is shared among all physical cores in the processor. There are two L1 caches to separately store data and instructions known as L1D and L1I respectively. Figure 2.4 shows the memory hierarchy of a modern Intel based computer.

In general, when the processor tries to read data from a memory address, the request is issued to the closest cache level in the hierarchy. If that cache level misses the access, it passes to the next cache level, and finally fetches the data from DRAM. The next time, the processor does not read the data from DRAM memory. It fetches the data from the closest available cache, unless the data has changed or the cache block has been evicted.

The cache read/write operations are performed in cache lines. The line size is 64 bytes in most processors. Caches are smaller and faster as they are closer to the processor, and they are in general much smaller than DRAM memory. To be able to cache the DRAM data to different levels of cache hierarchy and utilize it in a simple but efficient way, an addressing schema and a coherency and replacement policy are needed to be enforced.

Cache, unlike others, is not directly addressable and programmable from the software stack. However, the software can use some instructions and side effects to manage the usage of cache memory. The knowledge of the enforced addressing schema and policies are important to engineer cache usage in a predictable manner. In the next subsections, these concepts are briefly explained.

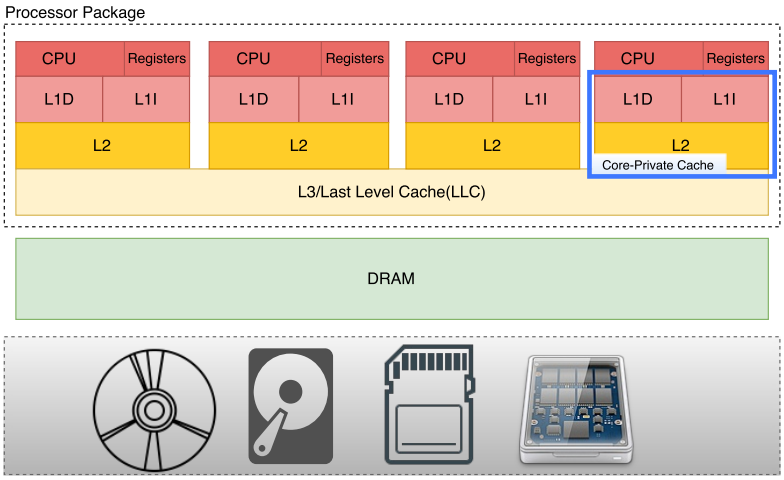


Figure 2.4: Memory hierarchy of a modern computer

2.2.1 Cache Addressing Schema

Each level of cache is grouped into n sets, and each set can store m number of lines, i.e., m-way associative. The physical address is used to position the data in the cache. With a line size of b-byte, the lower $\log_2 b$ bits of the physical address are used to index the byte in a line, and the next $\log_2 n$ bits select the set that the line is mapped to.

In our target architecture, we have 32 kB of L1D and 32 kB of L1I cache to store data and instructions respectively. In L1D, there exists 64 sets and each set is 8-way associative. It should be noted that the line size is 64 byte. We used L1D as our side channel primitive. Figure 2.5 illustrates the cache addressing schema for L1D.

2.2.2 Cache Coherency and Replacement policy

In a multi-processor system with different levels of cache, maintaining data for the same address between different caches is a challenge. For example, if two physical cores try to process data from the same address, there is a chance that one core updates its core-private caches with the latest information, while the other core uses the outdated information.

To avoid such synchronization problems, there are different policies that can be adopted, e.g., not to cache shared data. For simplicity, the general approach used on our target processor is for the L3 cache to be inclusive of the L1 and L2 cache, which means the same data as L1 and L2 should be present for L3. On top of that, if a cache line is evicted from the L3 cache, it will be evicted from L1 and L2 but not reversely. As a result, in our attack on L1, filling a cache line will fill corresponding L2 and L3 lines, but evicting the same cache line will not affect L2 and L3. This behavior is important for the construction of cache eviction policy in our attacks.

The policy that is used to evict a line within a set is another important behavior.

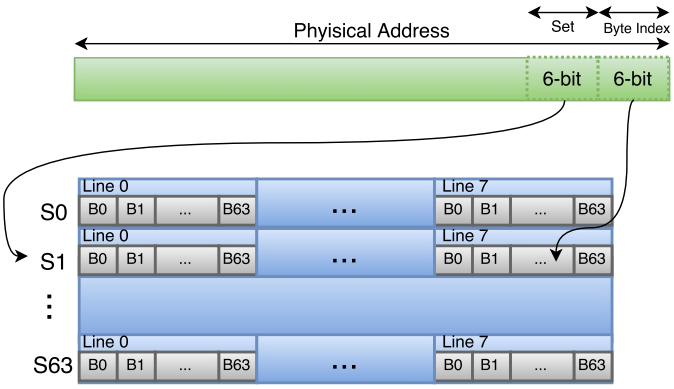


Figure 2.5: L1D Cache Addressing Schema

As mentioned, several addresses can be mapped to the same set. When a set is full, the processor frees space for the upcoming cache lines. It needs to decide which line to evict beforehand. The most common cache replacement policy is Least Recently Used (LRU), which discards the least recently used item first. This behavior makes `Prime and Probe` attack (explained in the next section) more reliable, since an adversary who tries to fill all the cache lines of a set will not be worried to accidentally evict its own cache lines. Our target processor uses an adaptive LRU approach. In our experiments, we assume that it is a simple LRU cache replacement, since the adaptive behavior is unknown. Our experiments show that the amount of noise added due to its adaptive behavior is negligible.

2.3 Microarchitectural Attacks in SGX

Despite all the protection that SGX offers, the documentation specifically claims that side channel attacks were not considered under the threat scope in its design [18, 33]. In fact, although dealing with encrypted memory pages, the cache utilization is performed in a decrypted mode and concurrently to any other process in the system. This means that the hardware resources can be utilized as covert channels by both malicious enclaves and OSs. While enclave-to-enclave attacks have several similarities to cross-VM attacks [51], malicious OS-to-enclave attacks can give attackers a new capability not observed in other scenarios: virtually unlimited temporal resolution.

The OS can interrupt the execution of enclave processes every small number of accesses to check the hardware utilization, as just the TLB cache (but no other cache) is flushed during context switches. Further, while cross-core attacks gained huge popularity in others scenarios (e.g. clouds or smartphones) for not requiring core co-residency, a compromised OS can assign an enclave execution any core of its choice, and therefore use any core-private channel. Thus, while SGX can prevent untrusted software to perform Direct Memory Access (DMA) attacks, it also

gives almost full resolution for the exploitation of hardware covert channels. For instance, an attacker can exploit page faults to learn about the memory page usage of the enclave process [56, 62]. Further she can create contention and snoop on the utilization of any core-private and core-shared hardware resource, including but not limited to Branch Prediction Units (BPUs), L1 caches or LLC caches [1, 46, 40].

The applicability of DRAM based attacks like the rowhammer attack seems infeasible due to the protection and isolation that the enclaves perform on DRAM memory pages. Further, although applicable in other scenarios [9], execution of enclave do not update the Branch Trace Store (BTS) and Processor Trace (PT) , and these can not provide (at least directly) information about the isolated process.

From the aforementioned resources, the resource that gives most information is the hardware cache. Unlike page faults, which at most will give a granularity of 4 kB, cache hits/misses can give 64 byte utilization granularity (depending on the cache line size). In addition, while other hardware resources like Branch Prediction Units (BPU) can only extract branch dependent execution flow information, cache attacks can extract information from any kind of memory access. Although most prior work targets the LLC for being shared across cores, this is not necessary in SGX scenarios, local caches are as applicable as LLC attacks.

In summary, because caches are not flushed when the enclave execution is interrupted, the OS can gain almost unlimited timing resolution. Table2.1 compare the discussed channels. In short, the processor core-private cache, and in particular L1 cache is the channel that gives us the highest resolution, lowest noise and can target both code and data.

Table 2.1: Comparison of different side-channels and their applicability to attack enclave.

Channel	**CPC[1]**	**LLC[2]**	**BP[3]**	**PF[4]**	**TLB[5]**	**RH[6]**
Possible	Yes	Yes	Yes	Yes	No	No
Resolution	64 byte	64 byte	Branches	4 kB	N/A	N/A
Noise	Local	Global	Local	N/A	N/A	N/A
Target	Data+Code	Data+Code	Code	Data+Code	N/A	N/A

[1] Core-Private Cache; [2] Last-Level Cache; [3] Branch Predictor Cache;
[4] Page Fault; [5] TLB Cache; [6] Rowhammering

2.3.1 Prime+Probe

The Prime+Probe attack was first introduced in [46] as a spy process capable of attacking core-private caches. It was later expanded to recover RSA keys [2], keystrokes in commercial clouds [49] and El Gamal keys across VMs [66]. Later the attack was shown to be applicable also in the LLC [40, 34]. As our attack is carried

out in the L1 caches, we do not describe the major hurdles (e.g. slices) that an attacker would have to overcome to implement it in the LLC. The Prime+Probe attack is mainly implemented in 3 stages:

- **Prime Stage:** in which the attacker fills the entire cache or a small portion of it with her own junk data.
- **Victim Access Stage:** in which the attacker waits for the victim to make accesses to particular sets in the cache, hoping to see data dependent cache set utilization. Note that, in any case, victim accesses to primed sets will evict at least one of the attackers junk memory blocks from the set.
- **Probe Stage:** in which the attacker performs a per-set timed re-access of the previously primed data. If the attacker observes a high probe time, she deduces that the cache set was utilized by the victim, as at least one of the memory blocks came from the memory. On the contrary, if the attacker observes low access times, she deduces that all the previously primed memory blocks still reside in the cache set, i.e., it was not utilized by the victim.

Thus, the `Prime and Probe` methodology allows an attacker to guess the cache sets utilized by the victim. In the L1 cache in particular, this information can be used to determine the lower bits of the accessed physical address. Further, the translation of virtual to physical address does not affect the lower 16 bits of the address. This lets an attacker to infer which offset within a memory page of the victim has been accessed. This information can be exploited in many privacy and security critical applications. For example, A full key recovery attack can be mounted if the algorithm has key-dependent memory accesses that are translated into different cache memory accesses.

2.4 Side-Channel Attacks & SGX

Side-channel attacks have been studied for many years. On a local area network having relatively accurate timing measurement, the timing of the decryption operation on a web server could reveal enough information about private keys stored on the server [15]. Timing attacks are capable of breaking important cryptography primitives, such as Diffie-Hellman exponent and factor of RSA keys [36]. More specifically, microarchitectural timing side channels have been explored extensively [25]. The first few attacks proposed were based on the timing difference between L1/L2 core private cache misses and hits.

Microarchitectural Side-Channels

Generally, microarchitectural timing attacks are based on the fact that a spy process is capable of using a resource shared with the victim, and utilize that to measure

timing differences of various victim operations. The difference of access time to the shared cache is sufficient for an attackers to create a side channel to a victim process and infer memory access patterns of that process. In the earlier days, these attacks reflected the effectiveness of this technique to recover cryptography keys of ciphers such as DES [60], AES [10] and RSA [47]. Although there exist solutions to make cryptographic implementation resistant to cache attacks [13, 46], the adoption of these solutions are limited due to worse performance. Further, cache attacks are capable of extracting information from non-cryptographic applications [65].

More recent proposals applied cache side channels on shared Last Level Caches (LLC), a shared resource among all the cores. This is important as, compared to previous core-private attacks, LLC attacks are applicable even when attacker and victim reside in different cores. The Flush+Reload [63, 6] LLC attack is only applicable to systems with shared memory. Flush+Reload can be applied across VMs [35], in Platform As a Service (PaaS) clouds [65] and on smartphones [39]. The Flush+Reload attack is less prone to noise, as it depends on the access to a single memory block. However, it is constrained by the memory deduplication requirement.

`Prime and Probe` [40], shows that in contrast to the previous L1/L2 core private cache side channels and the Flush+Reload attack, practical attacks can be performed without memory deduplication or a core co-residency requirement. The `Prime and Probe` attack, unlike Flush+Reload, can be implemented from virtually any cloud virtual machines running on the same hardware. The attacker can identify where a particular VM is located on the cloud infrastructure such as Amazon EC2, create VMs until a co-located one is found [49, 64] and perform cross-VM `Prime and Probe` attacks [34]. `Prime and Probe` can also be mounted from a browser using JavaScript [45] and as a malicious application on smartphones [39]. These side channels can be improved by causing a denial of service to the Linux CFS scheduler [28].

In addition to caches, Branch Target Buffer (BTB) is a shared processor cache used to predict the target of a branch before its execution. It can be exploited to determine if a branch has been taken by a target process or not [1, 38]. More recently, BTB has been exploited to bypass Address Space Layout Randomization (ASLR) [23].

Intel SGX Attack & Defence

Intel SGX has been analyzed based on the available resources from Intel [18]. A side channel resistant TCB is proposed in the literature [19]. However, the proposed solution requires significant changes to the design of the processor and it adds performance penalties that may not be desirable. Similar to Intel SGX, ARM TrustZone is vulnerable to side-channel attacks [39]. Control-Channel attack [62] has further been proposed using page-fault as a side channel. An adversarial OS can introduce page faults to a target application and, based on the timing of the accessed page, the

execution flow of a target can be inferred at page size granularity. Another possible attack on SGX can exploit race conditions between two running threads inside an enclave [61].

Runtime control flows inside SGX can be inferred using BTB [38]. However, data dependent table based computations, e.g., AES T-Table and S-Box implementations are not vulnerable to this side channel. It has been demonstrated that LLC side-channels can be mounted from one enclave to another one [51]. This is similar to previous LLC attacks proposed outside of the SGX context with a specific method to find physical address of the enclave page. Similar to our work, using core-private cache as a side channel has recently been released on eprint servers independent of our work [11] . In contrast, their attack is not based on the interruption of the target, is focused on stealthiness, and has a much lower resolution.

On the more defensive proposals, page fault side channels that are effective on SGX can be defeated using software solutions [56] or by using an additional extension of Intel processors named TSX [55]. In the latter, a user level application can detect the occurrence of a page fault before passing it to the OS and terminate itself in the face of malicious page fault patterns. In other proposals, SGX-Shield [52] adds ASLR protection and introduces software diversity inside an enclave. Lastly a tool named Moat [57] allows developers to verify security of an enclave application based on different adversarial models.

Chapter 3

Cache Side-Channel Attack on SGX

3.1 Attack Design

We explain how to establish a high resolution channel from a compromised OS to monitor an SGX enclave. Such an attack allows fine-grained information leakage. To achieve this goal with our system level adversarial model, we exploit features of the operating system to create an accurate low noise channel.

We construct our attack by: **(1)** partitioning the available cache between the victim task running an enclave module and the rest of the system running core operating system kernel threads and every other running tasks. Afterwards, we do not want to measure cache hit/miss of the whole enclave at once since the execution of the enclave module may cause lots of memory accesses in one run. And the bigger the computation of the enclave module, the more noise. That is one of the limitations of previous cache side-channel attacks that the adversary has to run the target service or task many times and perform the attack using statistical data which the leaked information is more limited.

To avoid that, **(2)** we measure the target enclave one step at a time and in each step, the processor executes the target for a short amount of time. After we measured all the steps (or only the necessary steps), we can map the collected traces to the target enclave binary to extract security critical information. In this section, we first describe attacker capabilities, and then our main design goals.

3.1.1 Adversarial Model

In our attack, we assume that the adversary has root level access to a system running Intel SGX SDK. Attacker is capable of installing kernel modules and configuring boot properties of the machine. In addition, the attacker is capable of scheduling the execution of an enclave and the measurement tool at the same time. These assumptions are reasonable, as SGX promises a trusted environment for code ex-

ecution on untrusted systems. For example, The adversary could be a malicious cloud provider hosting a malicious OS, the root level access could have been gained through exploiting kernel services or social engineering, or the adversary has gained physical access to the system and can modify the the OS image on the disk. All of these are practical examples that SGX is proposed to protect against.

The attacker also should have access to the target enclave binary, and for simplicity, we assume the binary is not obfuscated. Access to the binary is inherent to the root access on the OS. However, obfuscation could increase the defense bar and makes it harder to infer security critical information from the collected memory access patterns. This complexity is outside of the scope of this work, and is irrelevant to our goal of exploring the main security problem.

3.1.2 Cache Partitioning & Noise Reduction

By default, the OS tries to maximize the performance of the available hardware resources by sharing them among different tasks. Parallel execution of different tasks scheduled on the same hardware adds significant noise to the main attack channel. In the design of our malicious OS, we want to use part of the available cache as an isolated channel resistant to the outside noise. Although having an ideal channel is optimistic, we can exploit OS functionalities to reduce the noise as much as possible. We can create a noise resistant, high bandwidth channel using the following design choices:

- Scheduling only the target and the spy `Prime and Probe` code on one of the physical cores, and dedication of other physical cores to all other running operations. This isolates the core-private cache of a physical core from the rest of the system, thus leaves the spy and the target an isolated core-private channel.

- Core isolation will not work if we want to use the shared cache among the processor cores such as LLC. However, we can force the virtual to physical page mapping to always map specific pages to physical addresses that target specific sets in the shared cache. Having that, we can map the target and spy pages to a subset of shared cache sets, and map every other memory page on the system to the remaining sets.

- It is possible to disable caching behavior for specific physical addresses. This significantly reduces the performance of the running tasks that uses those memory addresses. This method can not be used as a global system-wide option to disable the noise for all OS relevant memory accesses. Yet, this can be used on a case by case basis to reduce noise of a specific physical page.

- Configuration of CPU features available on the modern processors that affect frequency to maintain a stable clock frequency. These configurations help to

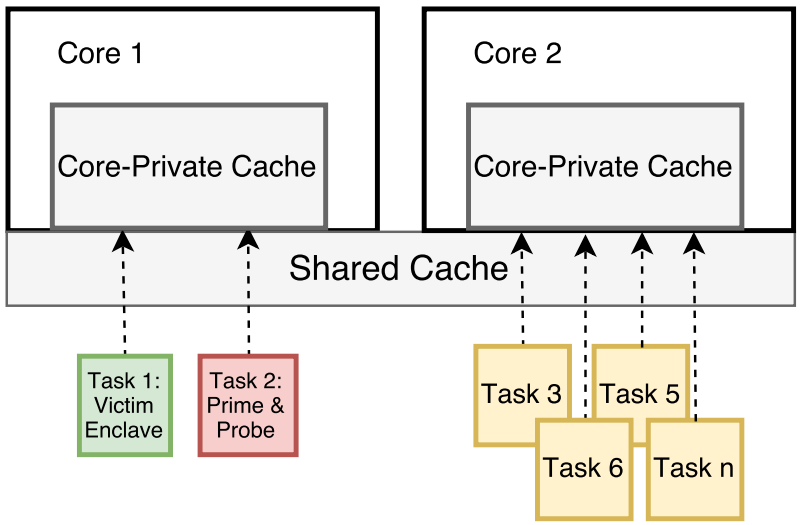


Figure 3.1: Scheduling the noisy tasks on a separate core from the victim & spy to create a core-private isolated channel.

have a pseudo-constant execution speed for all the instructions. This is useful to avoid variable time measurement during the probe step. Having a constant execution speed reduces the noise of eviction time thresholds.

In our attack, we use the isolated core-private cache as our channel, and configure the processor to have a constant frequency. Figure 3.1 illustrates the design of this channel.

3.1.3 Step by Step Execution

The longer the victim enclave runs without interruption, the higher the number of accesses made to the cache, implying higher noise and less temporal resolution. We force the enclave execution to be interrupted after short time intervals, in order to identify all enclave memory accesses. We name a small number of instructions inside and enclave executed without interruption, an execution unit. Here are the steps that we take in our attack tool:

1. Initialize the required memory pointers used during `Prime and Probe` .
2. Start the interrupt service to interrupt an isolated core after every small number of cycles.
3. Schedule the victim enclave on the same core.
4. The victim enclave will be interrupted.
5. Probe access times for all of the cache sets (which is potentially affected by the previous execution unit).

6. Store the measured access times to a separate buffer.

7. Prime all the target cache sets for probing the next execution unit.

8. Continue the execution until the next interrupt at step 4.

The loop in the aforementioned steps happen as far as we are interested in the measurement. Afterward, we stop the interrupt service to restore the core behavior to its normal state. Parallel to the above operations, we can schedule a task on another core without adding noise to our isolated channel to transfer the measurement buffer to the disk for real time or offline analysis of measured memory accesses.

3.1.4 Summary

By having the mentioned two elements, even a comparably small cache memory like the L1 cache turns into a high capacity channel with 64 byte granularity, as every major processor features 64 byte cache lines. Note that our spy process monitors the L1D cache, but can also be implemented to monitor in the L1I or last level caches. Our spy process is designed to profile all the sets in the L1D cache with the goal of retrieving the maximum information possible.

We should further reduce other possible sources of noise, e.g., context switches. The main purpose is that the attacker can cleanly retrieve most of the secret dependent memory accesses made by the target enclave. Note that the attacker can directly identify the set number that static data and instructions from the victim binary will occupy, as she has access to it. Since the L1 cache is virtually addressed, knowing the offset with respect to a page boundary is enough to know the accessed set.

3.2 Attack Implementation

We explain the technical details behind the implementation of our attack tool, in particular, how the noise sources are limited and how we increase the time resolution to obtain cleaner traces. We separate the implementation of our attack to the following three main components:

- Eviction Set: The construction of the logic and code that we used to accurately perform `Prime and Probe` on L1D set.

- Noise Reducing Configuration: The configuration and modification that we make dynamically to the OS.

- Malicious Kernel Driver: The malicious kernel driver that program the OS interrupt handler and perform the attack steps.

We explain the detail of each of the components in the following subsections.

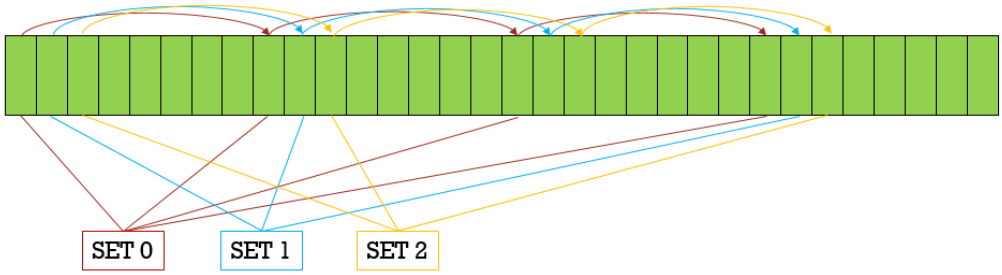


Figure 3.2: Pointers with addresses associated to the same set are initialized. It helps to reduce the number of instructions required to fill each set.

3.2.1 Eviction Set

The access times to L1 and L2 caches only differ in around 5 cycles and we want to `Prime and Probe` the entire L1D cache at each sampling, any unwanted memory access adds some amount of noise and reduces our channel capacity. To cope with this issue we use pointer chasing to `Prime and Probe` the 8-way sets, and C macro programming to repeat our `Prime and Probe` code stubs; which removes any unwanted memory access to the local stack pointer that might add noise and reduce the footprint of the obtained trace.

In pointer chasing approach, we initialize a memory buffer with linked lists of pointers associated to the same set. Having a buffer initialized like Figure 3.2, we can easily read all the addresses with only one instruction per read. Reducing the number of instructions that we execute on each sampling decreases the measurement time. Further, the dependency of access to each pointer on the previous one helps to avoid out-of-order execution of the instructions. Listing 3.1, 3.2, show the assembly code stubs that we used for our prime and probe stub. Note that, out-of-order execution of `Prime and Probe` operations adds an unpredictable noise, which is not acceptable.

```
mfence;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
```

Listing 3.1: Prime assembly code (AT&T syntax): Traversing the pointers.

```
mov %rax, %r10;
mfence;
rdtsc;
mov %eax, %ecx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
mov (%rbx), %rbx;
lfence;
rdtsc;
sub %rax, %rcx;
mov %r10, %rax;
neg %rcx;
```

Listing 3.2: Probe assembly code (AT&T syntax): Traversing the pointers and measuring the time using rdtsc instruction.

3.2.2 Noise Reducing Configuration

The target Linux OS schedules different tasks among available logical processors by default. The core of the Linux scheduler which has been implemented in `../kernel/sched/core.c` and the main scheduler function `__schedule` is triggered on every tick of the logical processor's local timer interrupt. One way to remove a specific logical processor of the default scheduling algorithm is through the kernel boot configuration `isolcpus` which accepts a list of logical cores to be excluded from scheduling. To avoid a logical core from triggering the scheduling algorithm on its local timer interrupt, we can use `nohz_full` boot configuration option, which similarly receives a list of logical processors as parameter.

Recall that reconfiguring the boot parameters and restarting the OS is included in our attackers capabilities. However, these capabilities are not necessary, as we can walk through the kernel task tree structure and turn the `PF_NO_SETAFFINITY` flag off for all tasks. Then, by dynamically calling the kernel `sched_setaffinity` interface for every task, we are able to force all the running kernel and user tasks to execute on specific processors. In addition to tasks and kernel threads, interrupts also need to be isolated from the target core. Most of the interrupts can be restricted to specific cores by configuring `smp_affinity`, except for the non-maskable interrupts (NMIs), which cannot be avoided. However, in our experience, the occurrence of

```
# Boot Configurations:
# isolcpus=1,3
# nohz_full=1,3

# Disable logical CPU pairs
echo 0 > /sys/devices/system/cpu/cpu2/online
echo 0 > /sys/devices/system/cpu/cpu3/online

# Set IRQ affinity mask to CPU 0
cd /proc/irq/
echo 1 | tee */smp_affinity

# Set CPU frequency scaling to performance mode
cd /sys/devices/system/cpu/
echo performance | tee cpu*/cpufreq/scaling_governor
echo 3050000 | tee cpu*/cpufreq/scaling_max_freq
echo 3010000 | tee cpu*/cpufreq/scaling_min_freq
```

Listing 3.3: Malicious OS Configuration

them is negligible and does not add noise to our channel.

CPU frequency has a more dynamic behavior in modern processors. Our target processor has **Speedstep** technology which allows dynamic adjustment of processor voltage and **C-state**, which allows different power management states [20]. These features, in addition to hyper-threading (concurrent execution of two threads in the same physical core), make the actual measurement of cycles through *rdtsc* less reliable. *rdtsc* instruction returns the number of cycle since reboot. Cache-side channel attacks that use this cycle counter are affected by the dynamic CPU frequency, as it affects the number of cycles it takes to execute an instruction. In non-system adversarial scenarios, these noise sources have been neglected thus forcing the attacker to do more measurements. In our scenario, these processor features can be disabled through the computer BIOS setup or can be configured by the OS to avoid unpredictable frequency. In our attack, we simply disabled every second logical processor to practically disable the hyper-threading feature.

To have a stable frequency in spite of the available battery saving and frequency features, we set the CPU scaling governor to **performance** and limited the maximum and minimum frequency range. In our experiment, these configurations that can be done from a root shell effectively reduces the noise of our isolated channel. Listing 3.3 shows the configurations during our experiments.

3.2.3 Malicious Kernel Driver

Aiming at reducing the number of memory accesses made by the victim between two malicious OS interrupts, we use the local APIC programmable interrupt, available on each processor core. The APIC timer has different programmable modes but we are only interested in the **TSC-Deadline** mode. In TSC deadline mode, the specified TSC value will cause the local APIC to generate a timer IRQ once the CPU reaches it. In the Linux kernel, the function `lapic_next_deadline` is responsible for setting the next deadline on each interrupt. The actual interrupt handler code for this IRQ is the function `local_apic_timer_interrupt`. We exploit the first function to set the TSC deadline to an arbitrary value and the second function to execute our `Prime and Probe` attack code.

In order to enable and disable our measurment tool as needed at runtime, we install hooks on these two functions. A look at the disassembly of these functions reveals that there are calls to a null function at the beginning that can be replaced by calls to the malicious functions of our kernel module.

```
ffffffff81050900 lapic_next_deadline
ffffffff81050900: e8 2b 06 7b 00        callq  0xffffffff81800f30
ffffffff81050905: 55                    push   %rbp

ffffffff81050c90 local_apic_timer_interrupt
ffffffff81050c90: e8 5b 02 7b 00        callq  0xffffffff81800ef0
ffffffff81050c95: 55                    push   %rbp
```

In the modified `lapic_next_deadline` function, we set the timer interrupt to specific values such that the running target enclave is interrupted every short period of execution time, thus the target processor trigger the interrupt in short periods and leave the actual running target enclave a short period of execution time.

In the modified `local_apic_timer_interrupt`, we first probe the entire 64 sets of the L1D cache to gather information of the previous execution unit and then prime the entire 64 sets for the next execution. After each probe, we store the the retrieved cache information from the sampling of the L1 D cache to a separate buffer in kernel memory. Our kernel driver is capable of performing 50000 circular samplings.

To be able to run our attack code against the execution inside the enclave, we need to start and end our attack at the time the target process is inside the enclave. For this purpose, our kernel driver supports two *ioctl* interfaces to install/uninstall the attack timer hooks. In the user level, we enable the attack timer interrupt before the call to the enclave interface and disable it right after.

Chapter 4

AES in Practice

Advanced Encryption Standard (AES) is a subset of an original cipher named Rijndael after Vincent Rijmen and Joan Daemen [21]. It is a standard by National Institute of Standards and Technology (NIST). The following gives a detailed description of AES and its various implementations to help the reader understand the attacks that we later perform. AES is a widely used symmetric block cipher that supports three key sizes from 128 bit to 256 bits. Our description and attacks focus on the 128-bit key version, AES-128, but most attacks described can be applied to larger-key versions as well. AES is based on 4 main operations: AddRoundKey, SubBytes, ShiftRows and MixColumns and a state table initialized to the plain text. Algorithm 1 shows the main procedure of AES encryption function.

Algorithm 1 AES Encryption

```
1:  procedure ENCRYPT
2:      i ← 0
3:      ExpandKeys
4:      AddRoundKey(i)
5:  round:
6:      SubBytes
7:      ShiftRows
8:      MixColumns
9:      AddRoundKey(i)
10:     i ← i + 1
11:     if i < n − 1 then
12:         goto round
13:     SubBytes
14:     ShiftRows
15:     AddRoundKey(i)
```

The input block size for AES is 128 bits but the operations are performed on a byte granularity. AES consists of a state table and n rounds of repeated operations

which n depends on the key size. In case of 128-bit key version, n is equal to 10. At the initial step of the procedure, the cipher key is expanded to the round keys used in each round individually and the state table is initialized with the input block. The last round ends with a final key addition without any MixColumns operation. The main leakage source in AES implementations comes from the fact that they utilize state-dependent table look up operations for the SubBytes operation. SubBytes replace each byte of the state table with the corresponding output of a non-linear function.

Rather than calculating the output of the function each time at runtime, a pre-computed 256 entry substitution table known as S-Box is used. These look-ups result in secret-dependent memory accesses that can be exploited by cache attacks and leak information about the round keys. In practice, a total of a 160 accesses are performed to the S-box during a 128-bit AES encryption, 16 accesses per round. We refer to this implementation as the S-box implementation.

4.1 T-Table Implementations

For performance improvements, some implementations combine the MixColumns and SubBytes in a single table lookup. At the cost of bigger pre-computed tables (and therefore, more memory usage) the encryption time can be significantly reduced. The most common type uses 4 T-tables: 256 entry substitution tables, each entry being 32 bits long. Listing 4.1 shows the summarized T-Table implementation of AES. The entries of the four T-tables are the same bytes but rotated by 1, 2 and 3 positions, depending on the position of the input byte in the AES state table ❶. Each rounds of operation is summarized to four statements consist of a series of 32-bit operations ❷. We refer to this as the *4 T-table* implementation.

Aiming at improving the memory usage of T-table based implementations, some designs utilize a single 256 entries T-table, where each entry is 64 bits long. Each entry contains two copies of the 32 bit values typically observed with regular size T-tables. This design reads each entry *with a different byte offset*, such that the values from the 4 T-tables can be read from a single bigger T-table. The performance of the implementation is comparable, but requires efficient non word-aligned memory accesses. We refer to this as the *Big T-table* implementation.

T-Table based implementations of AES have extensive memory footprints. The memory accesses to the 4 T-tables and Big T-Table will be issued over 64 and 32 sets respectively. As distinguishing memory accesses over more sets are easier to observe, these implementation are highly vulnerable to cache side channels.

```
❶ 1  static const u32 Te0[256] = {0xc66363a5U, ...};
   2  static const u32 Te1[256] = {0xa5c66363U, ...};
   3  static const u32 Te2[256] = {0x63a5c663U, ...};
   4  static const u32 Te3[256] = {0x6363a5c6U, ...};
   5  ...
   6  for (;;) {
❷  7    t0 = Te0[(s0>>24)] ^ Te1[(s1>>16)&0xff] ^ Te2[(s2>>8)&0xff]
   8      ^ Te3[(s3)&0xff] ^ rk[4];
   9    t1 = Te0[(s1>>24)] ^ Te1[(s2>>16)&0xff] ^ Te2[(s3>>8)&0xff]
  10      ^ Te3[(s0)&0xff] ^ rk[5];
  11    t2 = Te0[(s2>>24)] ^ Te1[(s3>>16)&0xff] ^ Te2[(s0>>8)&0xff]
  12      ^ Te3[(s1)&0xff] ^ rk[6];
  13    t3 = Te0[(s3>>24)] ^ Te1[(s0>>16)&0xff] ^ Te2[(s1>>8)&0xff]
  14      ^ Te3[(s2)&0xff] ^ rk[7];
  15  ...
```

Listing 4.1: AES 4 T-Table Encryption, Each T-Table consists of a shifted variation of the same 32 bit values. The round keys stored as 32 bit values (rk) will be added on each round.

4.2 Non-vulnerable Implementations

There are further efficient implementations of AES that are not susceptible to cache attacks, as they avoid secret-dependent memory accesses. These implementation styles include bit-sliced implementations [41], implementations using vector instructions [29], constant memory access implementations and implementations using AES instruction set extensions on modern Intel CPUs [32].

However, the aforementioned implementations all come with their separate drawbacks. The bit-sliced implementations need data to be reformatted before and after encryption and usually show good performance only if data is processed in large chunks [8]. Constant memory access implementations also suffer from performance as the number of memory accesses during an encryption significantly increases. While hardware support like AES-NI combines absence of leakage with highest performance, it is only an option if implemented and if the hardware can be trusted [59], and further might be disabled in BIOS configuration options.

4.3 Table Prefetching as a Countermeasure

In response to cache attacks in general and AES attacks in particular, several cryptographic library designers implement cache prefetching approaches, which just load the key dependent data or instructions to the cache prior to their possible utilization. In the case of AES this simply means loading all the substitution tables to the cache, either once during the encryption (at the beginning) or before each round

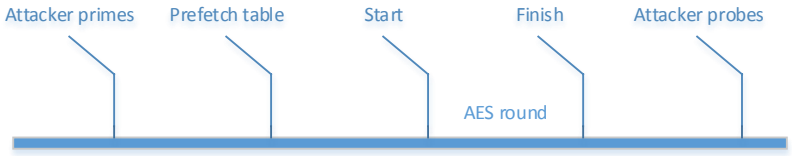


Figure 4.1: Prefetching and the timeline effect for a regular `Prime and Probe` attack

execution.

Prefetching takes advantage of the low temporal resolution that an attacker obtains when performing a regular non-OS controlled cache attack, as it assumes that an attacker cannot conduct an attack faster than the prefetching action. Translated to AES, prefetching assumes that a cache attack does not have enough temporal granularity to determine which positions in the substitution table have been used if they are prefetched, e.g., at the beginning of each round.

An example of the implications that such a countermeasure will have on a typical cache attack can be observed in Figure 4.1. The `Prime and Probe` process cannot be executed within the execution of a single AES round. Thanks to prefetching, the attacker is only able to see cache hits on all the Table entries.

We analyze whether those countermeasures, implemented in many cryptographic libraries, resist the scenario in which an attacker fully controls the OS and can interrupt the AES process after every small number of accesses. As it was explained in the background, attacking SGX gives a malicious OS almost full temporal resolution, which can reverse the effect of prefetching mechanisms.

4.4 Susceptibility to Cache Attacks

Depending on the implementation style, implementations can be more susceptible to cache attacks or less. The resolution an attacker gets depends on the cache line size, which is 64 bytes on all relevant modern CPUs, including Intel and most ARM cores.

For the **S-box** implementation, the S-box occupies a total of 4 cache lines (256 bytes). That is, an attacker able to observe each access to a table entry can learn at most two bits per access. Attacks relying on probabilistic observations of the S-box entries not being accessed during an entire encryption [35] would observe such a case with a probability of $1.02 \cdot 10^{-20}$, making a micro-architectural attack nearly infeasible. For a **4 T-tables** implementation, each of the T-tables gets 40 accesses per encryption, 4 per round, and occupies 16 cache lines. Therefore, the probability of a table entry not being accessed in an entire encryption is 8%, a fact that was

exploited in [35] to recover the full key.

In particular, all these works target either the first or the last round to avoid the MixColumns operation. In the first round, the intermediate state before the MixColumns operation is given by $s_i^0 = T_i[p_i \oplus k_i^0]$, where p_i and k_i^0 are the plaintext and first round key bytes i, T_i is the table utilization corresponding to byte i and s_i^0 is the intermediate state before the MixColumns operation in the first round. We see that, if the attacker knows the table entry being utilized x_i and the plaintext he can derive equations in the form $x_i{=}p_i \oplus k_i^0$ to recover the key. A similar approach can be utilized to mount an attack in the last round where the output is in the form $c_i = T_i[s_i^9] \oplus k_i^{10}$. The maximum an attacker can learn, however, is 4 bit per lookup, if each lookup can be observed separately. The scenario for attacks looking at accesses to a single cache line for an entire encryption learn a lot less, hence need significantly more measurements.

For a Big T-table implementation, the T-table occupies 32 cache lines, and the probability of not accessing an entry is reduced to 0.6%. This, although not exploited in a realistic attack, could lead to key recovery with sufficiently many measurements. An adversary observing each memory access separately, however, can learn 5 bits per access, as each cache line contains only 8 of the larger entries.

Note that an attacker that gets to observe every single access of the aforementioned AES implementations would succeed to recover the key with significantly fewer traces, as she gets to know the entry accessed at every point in the execution. This scenario was analyzed in [16] with simulated cache traces. Their work focuses on recovering the key based on observations made in the first and second AES rounds establishing relations between the first and second round keys. As a result, they succeed on recovering an AES key from a 4 T-table implementation with as few as six observed encryptions in a noise free environment. These attacker capabilities could even turn micro-architectural attacks against S-box implementations practical.

Chapter 5

AES Key Recovery Attack

In this chapter, we discuss our key recovery attack analysis and results. First, we show the results on quality of our side channel. Afterward, we discuss three different cryptographic key recovery attacks on different implementations of AES using our attack tool. We also explain the effect of table prefetching in this context. In short we demonstrate the following attacks:

- Known plaintext attack (KPA) on T-Table using the leakage from first two rounds of AES operation.
- Known ciphertext attack (KCA) on T-Table using the leakage from last two rounds of AES operation.
- Known ciphertext attack (KCA) on S-Box using the leakage from last round of AES operation.

In all of the above attacks, we assume no knowledge of the key used inside the enclave, but we assume to have access to the enclave binary, and thus to the offset of the substitution tables in the enclave module. We obtained these implementations from latest OpenSSL & wolfSSL libraries, and compiled them with available flags to make different characteristics enabled/disabled.

5.1 Channel Measurement

We analyzed the capacity of our side channel. This section discusses our results on our channel quality, and explains how we used our observations to infer memory accesses.

5.1.1 Experimental Setup

Our experimental setup is a Dell Inspiron 5559 laptop with Intel(R) Skylake Core(TM) i7-6500U processor running Ubuntu 14.04.5 LTS and SGX SDK 1.7. Our target processor has 2 hyper-threaded physical cores. Each physical core has 32 kB of L1D

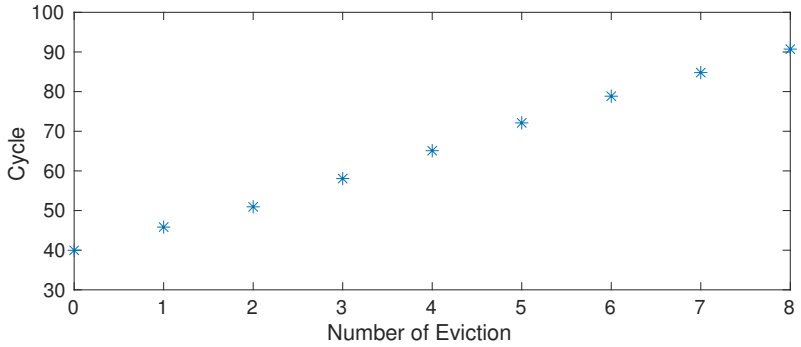


Figure 5.1: Average cycle count based on the number of evictions in a set.

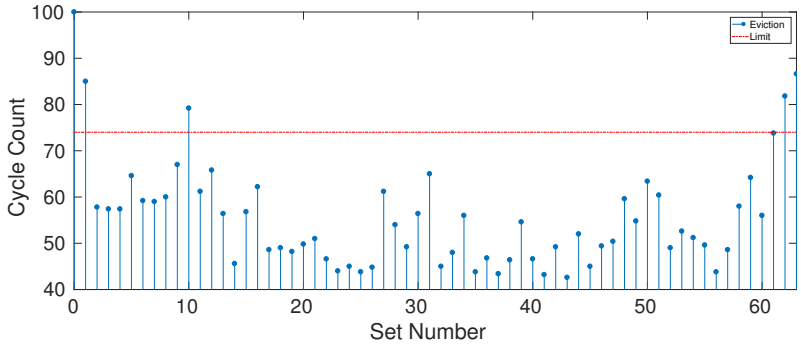


Figure 5.2: Average cycle count for each sets. Variations are due to channel noise, making 5 sets unusable for our attack.

and 32 kB of L1I local cache memory. The L1 cache, used as our side channel, is 8-way associative and consist of 64 sets.

5.1.2 Eviction Set Quality

Even though Skylake processors do not always use the LRU cache replacement policy and have a more adaptive undocumented cache replacement policy [26], our results show that we can still use the pointer chasing eviction set technique to detect memory accesses. In the specific case of our L1D cache, the access time for chasing 8 pointers associated to a specific set is 40 cycles on average.

In order to test the resolution of our covert channel, we took an average of 50000 samples of all the sets and varied the number of evictions from 0 to 8. The results can be seen in Figure 5.1, where the access time is increased by roughly 5 cycles for every additional eviction. Thus, the results show that our eviction technique gives us an accurate measurement on the number of evicted lines within a cache set.

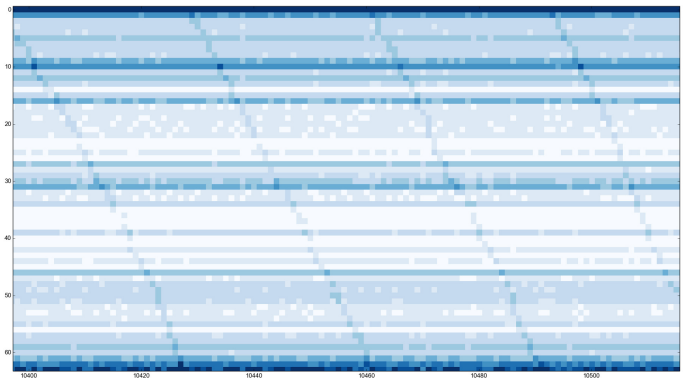


Figure 5.3: Cache hit map before filtering for context switch noise.

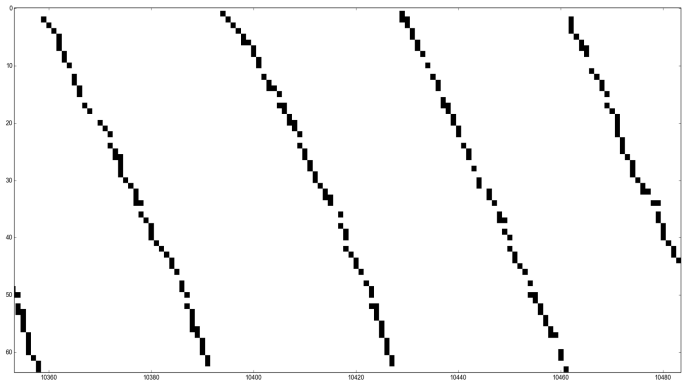


Figure 5.4: Cache hit map after filtering for context switch noise. Enclave memory access patterns are clearly visible once standard noise from context switch has been eliminated.

5.1.3 Channel Bandwidth

Our isolated CPU core and the L1D eviction set have the minimal possible noise due to the removal of various source of noises such as variable CPU frequency, and the OS noise; however, the actual noise from the context switch between enclave process and attacker interrupt is mostly unavoidable.

The amount of noise that these unwanted memory accesses add to the observation can be measured by running an enclave with an empty loop under our attack measurement. Our results, presented in Figure 5.2, show that every set has a consistent number of evictions. Among the 64 sets, there are only 5 sets that get filled as a side effect of the context switch memory accesses. For the other sets, we observed either 0 or less than 8 unwanted accesses. Due to the consistency of the number of evictions per set, we can conclude that only sets that get completely filled are blind and do not reveal any information about the target enclave, 4 out of 64 sets in our particular case.

An example of the applied noise ex-filtration process can be observed in Fig-

ure 5.4, in which the enclave process was consecutively accessing different sets. It shows the access pattern retrieved from the enclave once the context switch noise access has been taking into account and removed.. Figure 5.3 shows the hit access map, without taking into account the appropriate thresholds.

5.2 AES T-Table KPA Attack

Our first attack target the T-table implementations. To recover the AES key from as few traces as possible in a known plaintext scenario, we observe the memory access pattern of the first 2 rounds of the AES function. A perfect single trace of the first round cache access information reveals at most the least significant 4 and 5 bits of each key byte in 4 T-table (16 entries/cache line) and big T-table implementations (8 entries/cache line) respectively.

As we want to retrieve the key with a minimal number of traces, we also retrieve the information from the accesses in the second round and use the relations between the first and second round key. In particular, we benefit from relations described in [16], who utilized simulated data to demonstrate the effectiveness of their AES key recovery algorithm. As our attack is close to getting all specific T-table accesses, their equations serve us the most efficient mechanisms to retrieve the key from the first two rounds information.

$$
\begin{aligned}
x_0^1 &= 2s(p_0 \oplus k_0) \oplus 3s(p_5 \oplus k_5) \oplus s(p_{10} \oplus k_{10}) \oplus s(p_{15} \oplus k_{15}) \oplus s(k_{13}) \oplus k_0 \oplus 1 \\
x_1^1 &= s(p_0 \oplus k_0) \oplus 2s(p_5 \oplus k_5) \oplus 3s(p_{10} \oplus k_{10}) \oplus s(p_{15} \oplus k_{15}) \oplus s(k_{14}) \oplus k_1 \\
x_2^1 &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2s(p_{10} \oplus k_{10}) \oplus 3s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \\
x_3^1 &= 3s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus s(p_{10} \oplus k_{10}) \oplus 2s(p_{15} \oplus k_{15}) \oplus s(k_{12}) \oplus k_3 \\
x_4^1 &= 2s(p_4 \oplus k_4) \oplus 3s(p_9 \oplus k_9) \oplus s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus 1 \\
x_5^1 &= s(p_4 \oplus k_4) \oplus 2s(p_9 \oplus k_9) \oplus 3s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \\
x_6^1 &= s(p_4 \oplus k_4) \oplus s(p_9 \oplus k_9) \oplus 2s(p_{14} \oplus k_{14}) \oplus 3s(p_3 \oplus k_3) \oplus s(k_{15}) \oplus k_2 \oplus k_6 \\
x_7^1 &= 3s(p_4 \oplus k_4) \oplus s(p_9 \oplus k_9) \oplus s(p_{14} \oplus k_{14}) \oplus 2s(p_3 \oplus k_3) \oplus s(k_{12}) \oplus k_3 \oplus k_7 \\
x_8^1 &= 2s(p_8 \oplus k_8) \oplus 3s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_9^1 &= s(p_8 \oplus k_8) \oplus 2s(p_{13} \oplus k_{13}) \oplus 3s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \oplus k_9 \\
x_{10}^1 &= 2s(p_8 \oplus k_8) \oplus 3s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{15}) \oplus k_2 \oplus k_6 \oplus k_{10} \\
x_{11}^1 &= 2s(p_8 \oplus k_8) \oplus 3s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{12}) \oplus k_3 \oplus k_7 \oplus k_{11} \\
x_{12}^1 &= 2s(p_{12} \oplus k_{12}) \oplus 3s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \oplus s(k_{13}) \oplus k_{12} \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_{13}^1 &= s(p_{12} \oplus k_{12}) \oplus 2s(p_1 \oplus k_1) \oplus 3s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \oplus s(k_{14}) \oplus k_{13} \oplus k_1 \oplus k_5 \oplus k_9 \\
x_{14}^1 &= 2s(p_{12} \oplus k_{12}) \oplus 3s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \oplus s(k_{15}) \oplus k_{14} \oplus k_2 \oplus k_6 \oplus k_{10} \\
x_{15}^1 &= 2s(p_{12} \oplus k_{12}) \oplus 3s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}
\end{aligned}
\tag{5.1}
$$

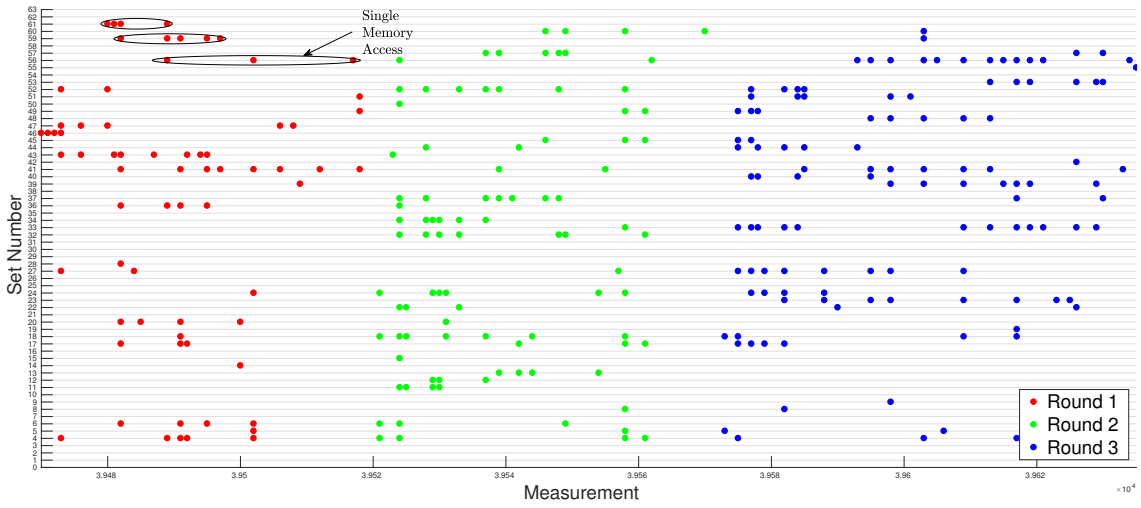


Figure 5.5: Memory footprint of the AES execution inside enclave over time steps. At each step, a few instructions has been executed.

Equations 5.1 show the relations of the first round key bytes. k_i is a key byte, and x_i^1 is the index of each T-Table lookup in the second round. We first gather enough information for the high nibbles of the key bytes from the first round memory accesses. At the next step, solving the above equations using the information from the second round traces reduce the key space to a computationally small size. An experiment on a set of simulated data gathered using the binary instrumentation tool, Pin, shows that an ideal cache access trace of the first two AES rounds (32 memory accesses) reduce the key size to 4-8 bits.

In our specific practical attack, we face three problems:

1. Even in our high resolution attack, we have noise that adds false positives and negatives to our observed memory access patterns.

2. Our experiments show that the out-of-order execution and parallel processing of memory accesses does not allow for a full serialization of the observed memory accesses.

3. Separating memory accesses belonging to different rounds can be challenging.

The first two facts can be observed in Figure 5.5, which potentially shows 16 memory accesses to each round of a 4 T-table (4 access per table) AES. Due to our high resolution channel and the out-of-order execution of instructions, we observe that we interrupt the out-of-order execution pipeline while a future memory access is being fetched. Thus, interrupting the processor and evicting the entire L1D cache on each measurement forces the processor to repeatedly load the cache line memory until the target read instruction execution completes.

Hence, attributing observed accesses to actual memory accesses in the code is not trivial. Although this behavior adds some confusion, we show that observed

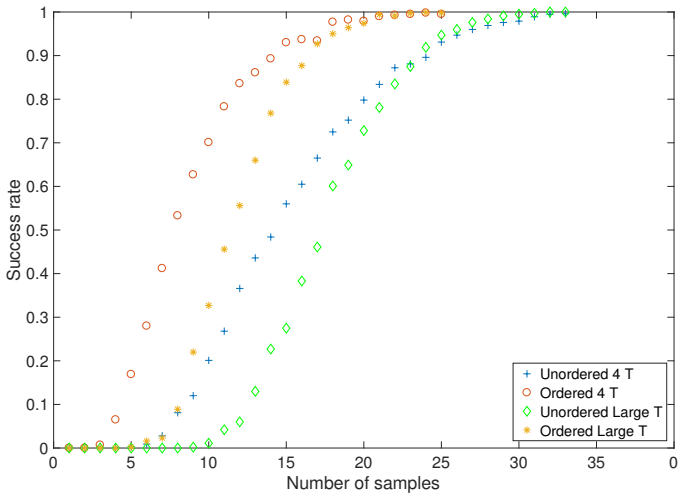


Figure 5.6: First round attack key recovery success rate.

accesses still have minimal order that we can take into account. As for the third fact, it involves thorough visual inspection of the collected trace. In particular, we realized that beginning of every round involves the utilization of a substantially higher number of sets than the rest, also observable in Figure 5.5.

In the first implementation of our key recovery algorithm, we just use the set access information without taking into account the ordering of our observed accesses. Recall that we have access to the binary executed by the enclave, and thus, we can map each set number to its corresponding T-table entry. This means that all our accesses can be grouped on a T-table basis. Duplicated accesses to a set within a round are not separated and are considered part of the same access, due to its difficulty with out-of-order execution. After applying this filter to the first and second round traces, we apply the key recovery algorithm, as explained in [16].

The accuracy of our measurements with respect to the aforementioned issues can be seen in Table 5.1. For the 4 T-table implementation, 55% of the accesses correspond to true accesses (77% of them were ordered), 44% of them were noisy accesses and 56% of the true accesses were missed. For the single Big T-table implementation, 75% of the T-table accesses corresponded to true accesses (67% ordered), 24% were noisy accesses and 12% of the true accesses were missed. The quality of the data is worse in the 4 T-table case because they occupy larger number of sets and thus include more noisy lines, as explained in Figure 5.2.

With these statistics and after applying our key recovery algorithms with varying number of traces, we obtained the results presented in Figure 5.6. If we do not consider the order in our experiments, we need roughly 20 traces (crosses and diamonds) to get the entire correct key with 90% probability in both the 4 T-table and single T-table implementations. In a different statistic using the first round information, Figure 5.7 shows the amount of recovered correct bits per number of samples on average.

Table 5.1: Statistics on recovered memory accesses for T-table implementations.

Implementation	4 T-table	Large T-table
True Positive	55%	75%
False Positive	44%	24%
False Negative	56%	12%
Ordered	77%	67%

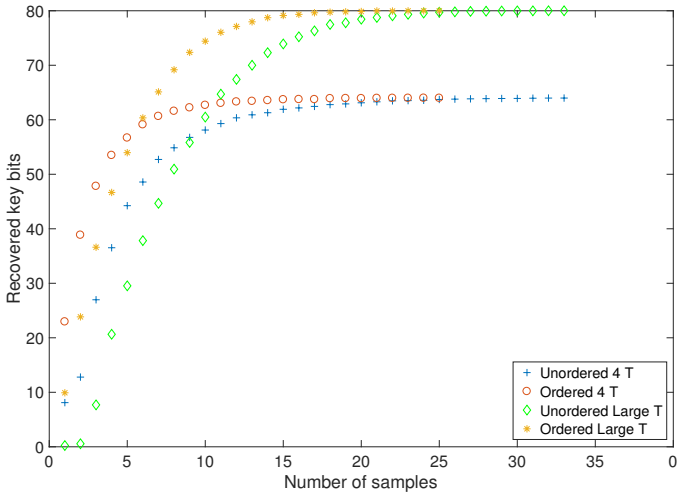


Figure 5.7: First round attack correct recovered key bits.

To further improve our results, we attempt to utilize the order of the observed accesses. We obtain the average position for all the accesses observed to a set within one round. These positions are, on average, close to the order in which sets were accessed. The observed order is then mapped to the order in which each T-table should have been utilized. Since this information is not very reliable, we apply a score and make sure misorderings are not automatically discarded. After applying this method, the result for our key recovery algorithm can be observed again in Figure 5.6, for which we needed around 15 traces for the 4 T-table implementation (represented with stars) and 12 traces for the single Big T-table implementation (represented circles) to get the key with 90% probability. Thus, we can conclude that using the approximate order helped us to recover the key with fewer traces.

5.2.1 Cache Prefetching

Cache prefetching is implemented to prevent passive attackers from recovering AES keys. Our attack tool, in theory, should bypass such a countermeasure by being able to prime the cache after the T-tables are prefetched. The observation of a trace when cache prefetching is implemented before every round can be observed in Figure 5.8.

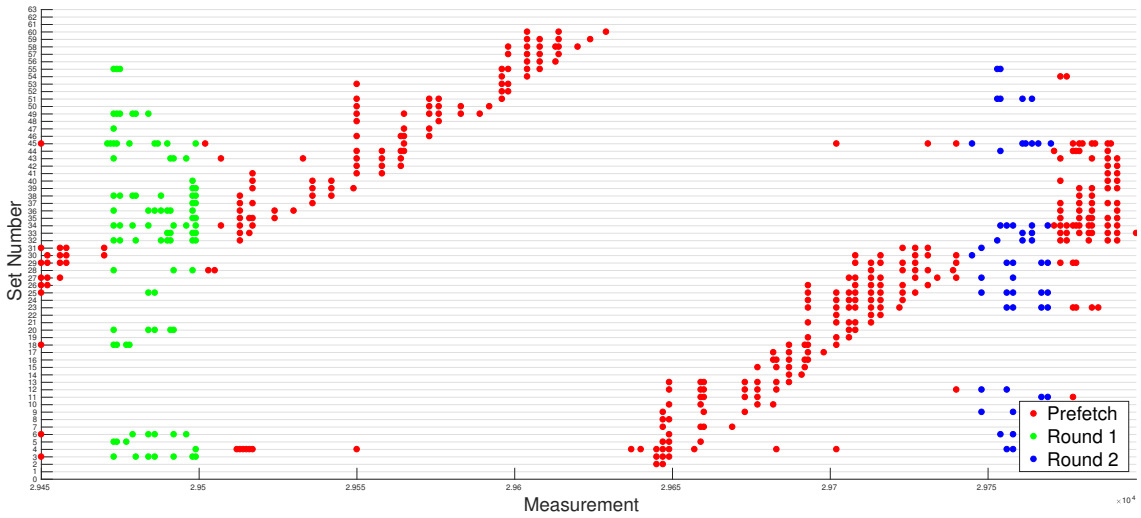


Figure 5.8: Memory footprint of the AES execution inside an enclave with prefetch countermeasure over time. The prefetch is clearly distinguishable and helps to identify the start of each round. Further, it also highlights out-of-order execution and in-order completion.

We can see how cache prefetching is far from preventing us to recover the necessary measurements. In fact, it eases the realization of our attack, as we now can clearly distinguish accesses belonging to different rounds clearly, allowing for further automation of our key recovery step. Thus, our attack not only bypasses but further benefits from mechanisms that mitigated previous cache attacks.

5.3 AES T-Table KCA Attack

The attack we demonstrated in the previous section on T-Table implementations can be improved in KCA scenario. Here, rather than looking at the the first two rounds information leakage, we use the information leakage from the last two rounds and knowledge of the ciphertext.

5.3.1 Last Round Attack on T-Table

In contrast to the KPA that we could only gather information about the high nibble of each key bytes with the first round information, we can recover more diverse key information from the last round attack described here. As a result, instead of 64 bits (80 bits in Big T-Table), the entire key will be recovered through repeated observation of the first round with different known ciphertexts. Measurement samples for the last round serve to increase the correctness recovered key.

This is because of the way AES rounds are designed. As we described before, the initial AES round only consists of a key addition step. The key addition is a

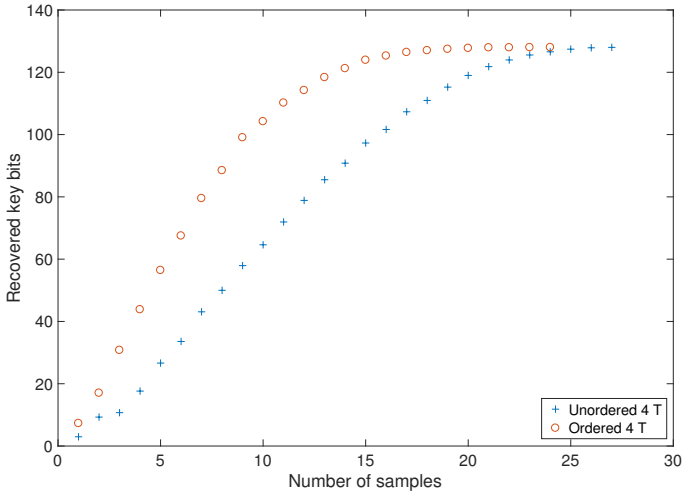


Figure 5.9: Last round attack recovered key bits.

linear operation. The first round memory access is only dependent on the plaintext and the initial round key. On the contrary, the last round memory accesses that we observe happens before a non-linear S-Box operation.

As a result, in an reverse view of AES operation, the last round information is dependent on the inverse S-Box of the last round key bytes and ciphertext bytes. We use the equation $x_i^{10} = s^{-1}(c_i \oplus k_i)$, which i is the index of the byte in the last round key, ciphertext and the sequence of last round memory accesses x^{10}.

We built a scoring histogram by implementing an algorithm based on this equation. Our results shows that the last round information is much more powerful in recovering the key compared to the combined information of the first two rounds. Figure 5.9 shows the number of correct key bits recovered in this attack with and without the partial access order information. On average, we can reduce the key to a brute-forcible space with around 7 partially ordered samples.

The key recover computation is also much more efficient in this attack. Previously, we had to solve the second round equations to recover the same key space, which it takes some computation time (5-8 minutes). However, recovering the key using the last round information, only takes a few seconds. Figure 5.10 shows the success rate of this attack on recovering the entire key.

5.3.2 Last Round Attack Improvement

We showed that using only the last round information is much more efficient than using the combined first two rounds information. In addition, we can still use the second last round information to improve the last round attack. There are scenarios, e.g., using a new symmetric key on each encryption operation, that we only have access to a few samples for a unique key. Reducing the number of samples make attacks on such scenarios more practical.

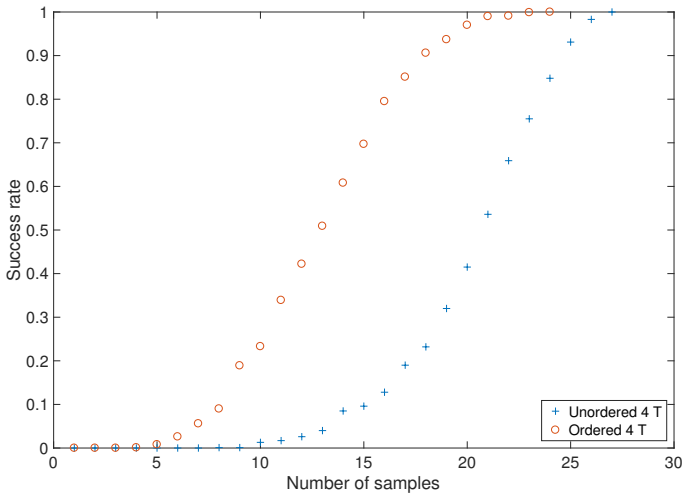


Figure 5.10: Last round attack key recovery success rate.

In this section, we propose our improvement to the last round attack. we hope to reduce the number of samples in the key recovery to ideally one. We created the relations between the first and second round keys as shown in Equation 5.2.

$$
\begin{aligned}
&2s(x_0^9) \oplus 3s(x_1^9) \oplus s(x_2^9) \oplus s(x_3^9) = s^{-1}(c_0 \oplus k_{160}) \oplus k_{160} \oplus s(k_{173} \oplus k_{169}) \oplus 0x36\\
&s(x_0^9) \oplus 2s(x_1^9) \oplus 3s(x_2^9) \oplus s(x_3^9) = s^{-1}(c_{13} \oplus k_{173}) \oplus k_{161} \oplus s(k_{174} \oplus k_{170})\\
&s(x_0^9) \oplus s(x_1^9) \oplus 2s(x_2^9) \oplus 3s(x_3^9) = s^{-1}(c_{10} \oplus k_{170}) \oplus k_{162} \oplus s(k_{175} \oplus k_{171})\\
&3s(x_0^9) \oplus s(x_1^9) \oplus s(x_2^9) \oplus 2s(x_3^9) = s^{-1}(c_7 \oplus k_{167}) \oplus k_{163} \oplus s(k_{172} \oplus k_{168})\\
&2s(x_4^9) \oplus 3s(x_5^9) \oplus s(x_6^9) \oplus s(x_7^9) = s^{-1}(c_4 \oplus k_{164}) \oplus k_{160} \oplus k_{164}\\
&s(x_4^9) \oplus 2s(x_5^9) \oplus 3s(x_6^9) \oplus s(x_7^9) = s^{-1}(c_1 \oplus k_{161}) \oplus k_{161} \oplus k_{165}\\
&s(x_4^9) \oplus s(x_5^9) \oplus 2s(x_6^9) \oplus 3s(x_7^9) = s^{-1}(c_{14} \oplus k_{174}) \oplus k_{162} \oplus k_{166}\\
&3s(x_4^9) \oplus s(x_5^9) \oplus s(x_6^9) \oplus 2s(x_7^9) = s^{-1}(c_{11} \oplus k_{171}) \oplus k_{163} \oplus k_{167}\\
&2s(x_8^9) \oplus 3s(x_9^9) \oplus s(x_{10}^9) \oplus s(x_{11}^9) = s^{-1}(c_8 \oplus k_{168}) \oplus k_{164} \oplus k_{168}\\
&s(x_8^9) \oplus 2s(x_9^9) \oplus 3s(x_{10}^9) \oplus s(x_{11}^9) = s^{-1}(c_5 \oplus k_{165}) \oplus k_{165} \oplus k_{169}\\
&s(x_8^9) \oplus s(x_9^9) \oplus 2s(x_{10}^9) \oplus 3s(x_{11}^9) = s^{-1}(c_2 \oplus k_{162}) \oplus k_{166} \oplus k_{170}\\
&3s(x_8^9) \oplus s(x_9^9) \oplus s(x_{10}^9) \oplus 2s(x_{11}^9) = s^{-1}(c_{15} \oplus k_{175}) \oplus k_{167} \oplus k_{171}\\
&2s(x_{12}^9) \oplus 3s(x_{13}^9) \oplus s(x_{14}^9) \oplus s(x_{15}^9) = s^{-1}(c_{12} \oplus k_{172}) \oplus k_{168} \oplus k_{172}\\
&s(x_{12}^9) \oplus 2s(x_{13}^9) \oplus 3s(x_{14}^9) \oplus s(x_{15}^9) = s^{-1}(c_9 \oplus k_{169}) \oplus k_{169} \oplus k_{173}\\
&s(x_{12}^9) \oplus s(x_{13}^9) \oplus 2s(x_{14}^9) \oplus 3s(x_{15}^9) = s^{-1}(c_6 \oplus k_{166}) \oplus k_{170} \oplus k_{174}\\
&3s(x_{12}^9) \oplus s(x_{13}^9) \oplus s(x_{14}^9) \oplus 2s(x_{15}^9) = s^{-1}(c_3 \oplus k_{163}) \oplus k_{171} \oplus k_{175}
\end{aligned}
\tag{5.2}
$$

In the equations, every 4 of them consist of 4 memory access, i.e, index of the T-Tables and some last round keys. In our algorithm, we group them to four based

on the memory accesses as follow:

- Group 1: x_4^9, x_5^9, x_6^9, x_7^9.
- Group 2: x_8^9, x_9^9, x_{10}^9, x_{11}^9.
- Group 3: x_{12}^9, x_{13}^9, x_{14}^9, x_{15}^9.
- Group 4: x_0^9, x_1^9, x_2^9, x_3^9.

In our algorithm, we first solve the first three group of equations individually to reduce the sub keys in each group. Then we merge the results of the sub keys for each group to build the reduced full last round key. Our experiment, with an idealistic data, shows that solving the first three group of equations, and merging them reduce the entire key space to 16 bits. We do not use Group 4 of equations at the first step, since there are more key bytes involved and solving them is not computationally efficient. Moreover, We do not necessarily need to use them if we have an ideal memory trace as 16 bits is a very small key space.

However, we can use the first group of equation to verify the correct key. At the second step in our algorithm, we verify the 16 bits key with the information gathered from the first 4 memory accesses in group 4. Our experiment shows that we can reduce the key space to less than 1 bit, i.e, recover the exact last round key. Note that AES key schedule is a reversible algorithm, and we can recover the cipher key having the last round key without any heavy computation.

Our experiment over simulated data show that we can recover the cipher key in 2 hours having only one ideal sample. We did not use real data as our current implementation is not efficient, and more noise entropy causes more computation. However, the discussed algorithm can significantly be improved to recover key using real data, and this is enough evidence to show that using the first and second last information is much more powerful.

5.4 AES S-Box KCA Attack

Using the S-box implementation is seen as a remedy to cache attacks, as all S-box accesses use only a very small number of cache lines (typically 4). With 160 S-Box accesses per encryption, each line is loaded with a very high likelihood and thus prevents low resolution attackers from gaining information. Adding a prefetch for each round does not introduce much overhead and also prevents attacks that attempt interrupting the execution [14, 28]. However, our attack can easily distinguish S-box accesses during the rounds, but due to the out-of order execution, it is not possible to distinguish accesses for different byte positions in a reliable manner, especially since all accesses hit the same 4 cache lines. With or without the prefetch, we see accesses to all 4 cache lines even if not all 4 cache lines are accessed in that round.

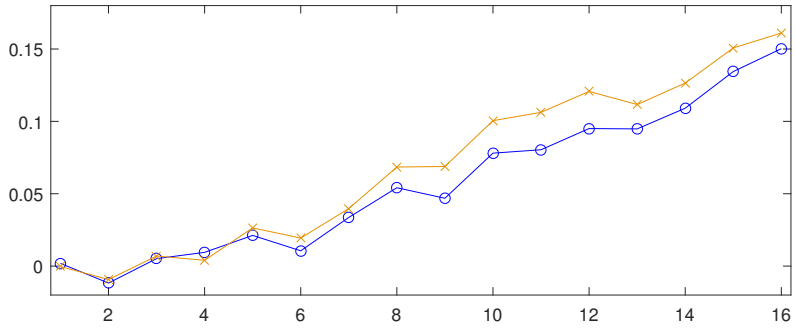


Figure 5.11: Correlation between observed cache accesses and predicted accesses caused by one byte position. Leakage is much stronger for later byte positions. Correlation for raw observed accesses (blue) vs. relative accesses (amber).

However, one distinguishable feature is the number of accesses each set sees during a round. We hypothesize that the number of observed accesses correlates with the number of S-box lookups to that cache line. If so, a classic DPA correlating the observed accesses to the predicted accesses caused by one state byte should recover the key byte. Hence we followed a classic DPA-like attack on the last round, assuming known ciphertexts. Since observed leakage is no longer distinguishable by byte position, more traces will be needed to recover key information.

The used model is rather simple: for each key byte k, the accessed cache set during the last round for a given ciphertext byte c is simply given as $set = S^{-1}(x \oplus k) \gg 6$, i.e. the two MSBs of the corresponding state byte before the last SubBytes operation. The access profile for a state byte position under an assumed key k and given ciphertext bytes can be represented by a matrix A where each row corresponds to a known ciphertext and each column indicates whether that ciphertext resulted in an access to the cache line with the same column index. Hence, each row has four entries, one per cache line, where the cache line with an access is set to one, and the other three columns are set to zero (since that state byte did not cause an access).

Our leakage is given as a matrix L, where each row corresponds to a known ciphertext and each column to the number of observed accesses to one of the 4 cache lines. A correlation attack can then be performed by computing the correlation between A and L, where A is a function of the key hypothesis. We used synthetic, noise-free simulation data for the last AES round to validate our approach, where accesses for 16 bytes are accumulated over 4 cache lines for numerous ciphertexts under a set key. The synthetic data shows a best expectable correlation of about .25 between noise-free cumulative accesses L and the correct accesses for a single key byte A. As little as 100 observations yield a first-order success rate of 93%.

Next, we gathered hundreds of measurements using our attack tool. Note that due to a lack of alignment, the collection of a large number of observations and the extraction of the last round information is not trivially automated and at the moment still requires manual intervention. When performing the key recovery at-

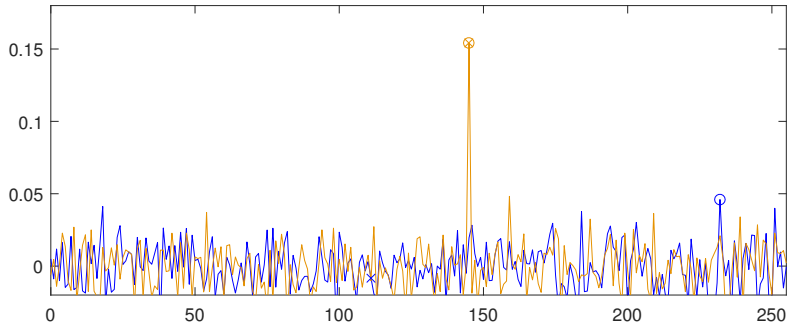


Figure 5.12: Correlation over key value for the best (k_{15}, amber) and worst (k_0, blue) byte positions based on 1500 traces. The guess with the highest correlation (o) and the correct key (x) a match only for k_{15}.

tack, even 200 observations yielded 4-5 key bytes correctly. However, the first-order success rate only increases very slowly with further measurements.

We further observed that **(1)** more traces always recover later key bytes first and **(2)** key ranks for earlier lookups are often very low, i.e. the correct key does not even yield a high correlation. To analyze this behavior, we simply correlated the expected leakage A for each byte position to the observed leakage L. The result is shown in Figure 5.11. It can be observed that the correlation for the later key bytes is much stronger than for the earlier key bytes. This explains why later key bytes are much easier to recover. The plot also shows a comparison of using the absolute number of observed accesses (ranging between 10 and 80 observed accesses per round, blue) an the relative number of accesses per cache set (amber) after removing outliers.

Results for the best and the worst key guess are shown in Figure 5.12. For k_{15} (amber), the correlation for the correct key guess is clearly distinguishable. For k_0 however, the correct key guess does not show any correlation with the used 1500 observations. In summary, 500 traces are sufficient to recover 64 key bits, while 1500 recover 80 key bits reliably. While full key recovery will be challenging, recovering 12 out of 16 key bytes is easily possible with thousands of observations. The remaining key bytes can either be brute-forced or can be recovered by exploiting leakage from the second last round.

Next, we explain the reason why we believe bytes processed first are harder to recover. The Intel core i7 uses deep pipelines and speculative out-of-order execution. Up to six micro-instructions can be dispatched per clock cycle, and several instructions can also complete per cycle. As a result, getting order information for the accesses is difficult, especially if 16 subsequent S-box reads are spread over only 4 cache lines. While execution is out-of-order, each instruction and its completion state are tracked in the CPU's reorder buffer (ROB). Instruction results only affect the system state once they are completed *and* have reached the top of the ROB. That is, micro-ops *retire* in-order, even though they *execute* out-of-order. The re-

sult of micro-ops that have completed hence do not immediately affect the system. In our case, if the previous load has not yet been serviced, the subsequent completed accesses cannot retire and affect the system until the unserviced load is also completed.

Every context switch out of an enclave requires the CPU to flush the out-of order execution pipeline of the CPU [18]. Hence the interrupt causes a pipeline flush in the CPU, all micro-ops on the ROB that are not at the top and completed will be discarded. Since our scheduler switches tasks very frequently, many loads cannot retire and thus the same load operation has to be serviced repeatedly. This explains why we see between 9 and 90 accesses to the S-box cache lines although there are only 16 different loads to 4 different cache lines. The loads for the first S-box are, however, the least affected by preceding loads. Hence, they are the most likely to complete and retire from the ROB after a single cache access. Later accesses are increasingly likely to be serviced more than once, as their completion and retirement is dependent on preceding loads. Since our leakage model assumes such behavior (in fact, we assume one cache access per load), the model becomes increasingly accurate for later accesses.

Chapter 6

Conclusion

This work presented a high resolution attack, a new tool to analyze memory accesses of SGX enclaves. To gain maximal resolution, we combined a L1 cache Prime and Probe attack with OS modifications that greatly enhance the channel bandwidth. SGX makes this scenario realistic, as both a compromised OS and knowledge of the unencrypted binary are realistic for enclaves.

We demonstrate that our attack can be mounted to recover key bits from well accepted software implementations of AES, including ones that use prefetches for each round as a cache-attack countermeasure. Furthermore, keys can be recovered with as few as 7 observations for T-table based implementations. We also improved the number of required samples in our key recovery attack on these implementations. For the trickier S-box implementation style, 100s of observations reveal sufficient key information to make full key recovery possible.

Prefetching is in this scenario beneficial to the adversary, as it helps identifying and separating the accesses for different rounds. Our tool serves as evidence that security-critical code needs constant execution flows and secret-independent memory accesses. As SGXs intended use is the protection of sensitive information, enclave developers must thus use the necessary care when developing code and avoid microarchitectural leakages. For AES specifically, SGX implementations must feature constant memory accesses. Possible implementation styles are thus bit-sliced or vectorized-instruction-based implementations or implementations that access all cache lines for each look-up.

Bibliography

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[2] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Topics in Cryptology–CT-RSA 2008*, pages 256–273. Springer, 2008.

[3] ARM TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`. Accessed: April 27, 2017.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel OKeeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[6] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014.

[7] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[8] Daniel J. Bernstein and Peter Schwabe. *New AES Software Speed Records*, pages 322–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] Sarani Bhattacharya and Debdeep Mukhopadhyay. *Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms*, pages 248–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[10] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv preprint arXiv:1702.07521*, 2017.

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[13] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[14] Samira Briongos, Pedro Malagón, José L. Risco-Martín, and José Manuel Moya. Modeling side-channel cache attacks on AES. In *Proceedings of the Summer Computer Simulation Conference, SummerSim 2016, Montreal, QC, Canada, July 24-27, 2016*, page 37, 2016.

[15] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[16] A. C, R. P. Giri, and B. Menezes. Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 261–275, March 2016.

[17] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 2–13. ACM, 2008.

[18] Victor Costan and Srinivas Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086, 2015.

[19] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[20] Power Management States: P-States, C-States, and Package C-States. `https://software.intel.com/en-us/articles/`

power-management-states-p-states-c-states-and-package-c-states. Accessed: April 27, 2017.

[21] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[22] Kamal Dahbur, Bassil Mohammad, and Ahmad Bisher Tarakji. A survey of risks, threats and vulnerabilities in cloud computing. In *Proceedings of the 2011 International conference on intelligent semantic Web-services and applications*, page 12. ACM, 2011.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump Over ASLR: Attacking Branch PredShared Cache Attictors to Bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[24] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX. *arXiv preprint arXiv:1703.04583*, 2017.

[25] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *IACR Eprint*, 2016.

[26] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[27] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[28] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[29] Mike Hamburg. *Accelerating AES with Vector Permute Instructions*, pages 18–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[30] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 265–278. ACM, 2013.

[31] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[32] Intel. Intel Data Protection Technology with AES-NI and Secure Key.

[33] ISCA 2015 tutorial slides for Intel SGX. `https://software.intel.com/sites/default/files/332680-002.pdf`. Accessed: April 27, 2017.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing–and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[35] Gorka Irazoqui, Mehmet Sinan İncİ, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID*, pages 299–319, 2014.

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

[37] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Mark Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. Technical report, arxiv Archive, 2016. https://arxiv.org/pdf/1611.06952.pdf, 2017.

[39] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Armageddon: Last-level cache attacks on mobile devices. *arXiv preprint arXiv:1511.04897*, 2015.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[41] Mitsuru Matsui and Junko Nakajima. *On the Power of Bitslice Implementation on Intel Core2 Processor*, pages 121–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[42] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *arXiv preprint arXiv:1703.06986*, 2017.

[43] Thomas Morris. Trusted platform module. In *Encyclopedia of Cryptography and Security*, pages 1332–1335. Springer, 2011.

[44] Olga Ohrimenko, Felix Schuster, Cdric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[45] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[46] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.

[47] Colin Percival. Cache missing for fun and profit, 2005.

[48] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[49] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[51] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *ArXiv e-prints*, February 2017.

[52] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[53] Kristoffer Severinsen, Christian Johansen, and Sergiu Bursuc. Securing the End-points of the Signal Protocol using Intel SGX based Containers. *Security Principles and Trust Hotspot 2017*, page 1, 2017.

[54] Intel SGX. `https://software.intel.com/en-us/sgx`. Accessed: April 27, 2017.

[55] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[56] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[57] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184. ACM, 2015.

[58] Mario Strasser and Heiko Stamer. A software-based trusted platform module emulator. In *International Conference on Trusted Computing*, pages 33–47. Springer, 2008.

[59] Tatsuya TAKEHISA, Hiroki NOGAWA, and Masakatu MORII. AES Flow Interception: Key Snooping Method on Virtual Machine - Exception Handling Attack for AES-NI. Cryptology ePrint Archive, Report 2011/428, 2011. `http://eprint.iacr.org/2011/428`.

[60] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer, 2003.

[61] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[62] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[63] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[64] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328. IEEE, 2011.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[66] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2014.