CacheZoom: How SGX Amplifies The Power of Cache Attacks

Ahmad Moghimi
Worcester Polytechnic Institute
amoghimi@wpi.edu

Gorka Irazoqui
Worcester Polytechnic Institute
girazoki@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Abstract

Intel SGX is a hardware extension proposed to provide a Trusted Execution Environment on commodity processors. SGX disregards microarchitectural side-channels as out of scope of its threat model. In this paper, we propose a high-resolution cache side-channel attack and demonstrate its impact and its capability in overcoming the security goals of SGX technology. Our tool tracks virtually all memory accesses of SGX enclaves with high spatial and temporal precision. The resulting attack succeeds on implementations that were previously believed to be resistant to realistic cache attacks. We demonstrate that these new attacks can recover AES keys from an enclave with as few as ten measurements.

1 Motivation

With an increasing proliferation of parallel computing platforms ranging from embedded and IoT systems to cloud computing, processes with various levels of trust and criticality are allowed to to run in parallel and share system resources.

Traditionally, systems relied on the operating system (OS) to provide security and privacy services. In cloud computing, cloud service providers and the hypervisor software also become part of the Trusted Computing Base (TCB). Due to the high complexity and the various attack surfaces in modern computing systems, keeping an entire system secure and trusted is usually unrealistic [20, 31].

One way to reduce the TCB is to outsource security-critical services to Secure Elements (SE), a separate trusted hardware which usually undergoes rigorous security testing. Trusted Platform Modules (TPM), for example, provide services such as cryptography, secure boot, sealing data and attestation beyond the authority of the system software [36]. However, SEs come with their own drawbacks: they are static components and connected to the CPU over an untrusted bus. An alternative are Trusted Execution Environments (TEE), which 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 operating system. The software running inside a TEE has full access to the system's resources while it is protected from other applications and from the OS. Examples include ARM TrustZone [1] and Intel Software Guard eXtension (SGX) [2]. Intel SGX creates a TEE on an untrusted system by only trusting the hardware in which the code is executed. Trusted code is secured in an *enclave*, which is encrypted and authenticated by the hardware. When loaded, the CPU decrypts and verifies enclave code and data as it is moved into the cache. That is, enclaves are logically protected from malicious applications, the OS, and physical adversaries monitoring system buses. However, Intel SGX is not protected against attacks that utilize hardware resources as a covert channel [3]. And indeed, first works showing that microarchitectural side channels can be exploited have been proposed, including attacks using page table faults [51], the branch prediction unit [32], and caches [43].

Note that caches have become a very popular side channel in many scenarios, including mobile [33] and cloud environments [26]. Reasons include that LLC cache attacks perform well in cross-core scenarios on Intel machines. Another advantage of cache attacks are the high *spatial* resolution they provide. With a line size of only 64 bytes, attacks are able to distinguish accesses to different RSA and even ECC multiplicands for windowed exponentiation algorithms [34]. This high spatial resolution, combined with a good temporal resolution, have enabled attacks on all major asymmetric implementations, unless they are optimized for constant memory accesses. For symmetric cryptography, the scenario is more challenging. A table-based AES implementation can execute in a few hundred cycles, while a `Prime and Probe` cycle on the LLC easily takes 2000 cycles to mon-

itor a single LLC set. To avoid the undersampling, synchronized attacks first prime, trigger a single encryption and then probe, yielding at best one observation per encryption [9]. Better resolution is only possible if the attacker manipulates the OS resources.

1.1 Our Contribution

In this work we demonstrate not only that Intel SGX is vulnerable to cache based attacks, but show that with SGX, the quality of information retrieved is significantly improved. The improved resolution enables attacks that are infeasible in previously exploited scenarios, e.g., cloud environments or smartphones. In particular, we utilize all the capabilities that SGX assumes an attacker has, i.e., full access to OS resources. We construct a malicious OS that is able to interrupt the victim process every few memory accesses, thereby recovering high-resolution information about all memory accesses that the target enclave makes by applying the popular `Prime and Probe` attack in the L1 cache. The usage of core-private resources does not reduce the applicability of the attack, as the malicious OS schedules both victim and attacker processes in the same core.

While tracking memory accesses of an enclave with high temporal and spatial resolution has many application scenarios, we demonstrate the power of the side channel setup to construct attacks against several AES implementations. Further, we also demonstrate that countermeasures that have found widespread adoption in popular cryptographic libraries, like cache prefetching and S-box implementations, not only do not prevent key recovery attacks, but can create a cleaner channel and ease key recovery. In short, this work:

- Presents a powerful and low-noise covert channel implemented through the L1 cache. We take advantage of several capabilities corresponding to malicious operating systems. This covert channel can be applied against trusted execution environments to recover fine grain information about memory accesses, which often carry sensitive information.

- Demonstrates the new capabilities of our covert channel by recovering AES keys with less traces than ever in previously attacked implementations, 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 not effective in the context of SGX. In fact, prefetching can even ease the retrieval of attack traces.

2 Background

This section covers topics that help understand the side channel used to retrieve sensitive information. We discuss the basic functionality of Intel SGX and possible micro-architectural attacks that can be deployed against it.

2.1 How Intel SGX Works

With the Skylake CPU generation, Intel released SGX, a new subset of CPU instructions that allows execution of software inside isolated environments called *enclaves*. Enclaves are isolated from other software components running on the same hardware, including malicious OSs. SGX has recently gained popularity among the research community and various SGX-based security applications have been proposed [6, 7, 13, 37, 42].

Enclave modules can be shipped as part of an untrusted application and be loaded by untrusted components of the application. The untrusted component interacts with the system software, which dedicates specific trusted memory regions for the enclave. After that, the authenticity, integrity and confidentiality of a running enclave are provided and measured by the hardware. Any untrusted code base, including the system software, has no control over the trusted memory region. Untrusted applications can only use specific instructions to call the trusted component through predefined interfaces. This design helps developers to benefit from the hardware isolation for security critical applications.

SGX is designed to protect enclaves from malicious users that gain access to an OS, e.g., by inserting malicious kernel code. Memory pages belonging to an enclave are encrypted in DRAM and protected from a malicious OS to snoop on them. Pages are only decrypted when they are processed by the CPU, e.g., when they are moved to the hardware caches. In short, SGX assumes only the hardware to be trusted; any other agent is considered susceptible of being malicious. Upon enclave creation, virtual memory pages that can only map to a protected DRAM region (called the Enclave Page Cache) are reserved. The (potentially) malicious OS is in charge of the memory page mapping; however, SGX detects any malicious mapping performed by it. In fact, any malicious action from the OS will be stored by SGX and is verifiable by third party agents.

2.2 Micro-architectural 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. 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, 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 (but no other hardware resources) 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 malicious OS can assign an enclave any affinity of its choice, and therefore use any core-private covert channel. Thus, while SGX can prevent untrusted software to perform Direct Memory Access (DMA) attacks, it also gives almost full resolution for the exploitation hardware covert channels. For instance, an attacker can exploit page faults to learn about the memory page usage of the enclave process. 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 [4, 34, 39]. 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 [11], enclave processes do not update the Hardware Performance Counters, 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. Further, because caches are not flushed when the enclave execution is interrupted, the OS can gain almost unlimited timing resolution.

2.3 The Prime+Probe Attack

The Prime+Probe attack was first introduced in [39] as a spy process capable of attacking core-private caches. It was later expanded to recover RSA keys [5], keystrokes in commercial clouds [41] and El Gamal keys across VMs [55]. Later the attack was shown to be applicable also in the LLC [28, 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. 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 key 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. This information can be used to mount a full key recovery attack if the algorithm has key-dependent memory accesses that are translated into different cache set accesses.

3 Related Work

Timing 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 [16]. Timing attacks are capable of breaking important cryptography primitives, such as Diffie-Hellman exponent and factor of RSA keys [30]. More specifically, **microarchitectural timing side channels** have been explored extensively [22]. The first few attacks proposed were based on the timing difference between L1/L2 core private cache misses and hits. Generally, cache timing attacks are based on the fact that a spy process is capable of measuring accurate differences in memory access times. In the earlier days, these attacks reflected the effectiveness of this technique to recover cryptography keys of ciphers such as DES [49], AES [12] and RSA [40]. Although there exist solutions to make cryptographic implementation resistant to cache attacks [14, 39], most of these

solutions result in worse performance. Further, cache attacks are capable of extracting information from non-cryptographic applications [54].

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 [8, 52] LLC attack is only applicable to systems with shared memory. These side channels can be improved by causing a denial of service to the Linux CFS scheduler [24]. Flush+Reload can be applied across VMs [29], in PaaS clouds [54] and on smartphones [33]. 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. On the other hand, `Prime and Probe` [34], 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 [41, 53] and perform cross-VM `Prime and Probe` attacks [28]. `Prime and Probe` can also be mounted from a browser using JavaScript [38] and as a smartphone malicious application [33].

In addition to caches, other microarchitectural components such as **Branch Target Buffers (BTB)** are vulnerable to side-channel attacks [4, 32]. 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. More recently, BTB has been exploited to bypass Address Space Layout Randomization (ASLR) [21].

Security of 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 cache side-channel attacks [33]. Control-Channel attacks [51] have 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. Page fault side channels are effective on SGX and can be defeated using software solutions [46] or by using an additional extension of Intel processors named TSX [45]. 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. Another possible attack on SGX can exploit race conditions between two running threads inside an enclave [50]. In other defensive proposals, SGX-Shield [44] adds ASLR protection and introduces software diversity inside an enclave. Lastly a tool named Moat [47] allows developers to verify security of an enclave application based on different adversarial models.

4 Creating a High-resolution Side Channel on Intel SGX

We proceed to explain how to establish a high resolution channel for a malicious OS to monitor an SGX enclave. We first describe attacker capabilities, then our main design goals and how our malicious kernel driver is implemented. We finally we test the resolution of our newly gained covert channel.

4.1 Attacker Capabilities

In our attack, we assume that the adversary has root level access to a Linux OS running Intel SGX SDK. The attacker is capable of installing kernel modules and configuring boot properties of the machine. In addition, the attacker is capable of triggering the victim enclave at a specific time or install user level hooks into a running application that uses enclave interfaces for trusted operations. These assumptions are reasonable, as SGX promises a trusted environment for code execution on untrusted systems.

4.2 CacheZoom Design

To be able to create a high bandwidth channel with minimal noise, **(1)** we need to isolate the attackers' malicious spy process code and the target enclave's trusted execution from the rest of the running operations and **(2)** we need to perform the attack on small units of execution. By having these 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. In order to avoid OS noise, we dedicate one physical core to to our experimental setup, i.e., to the attacker `Prime and Probe` code and the victim enclave process. All other running operations on the system, including OS services and interrupts, run on the remaining cores. Furthermore, CacheZoom forces the

enclave execution to be interrupted after short time intervals, in order to identify all enclave memory accesses. Note that, 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. CacheZoom 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.

4.3 CacheZoom Implementation

We proceed to explain the technical details behind the implementation of CacheZoom, in particular, how the noise sources are limited and how we increase the time resolution to obtain cleaner traces.

4.3.1 Enclave-Attack Process Isolation

The target Linux OS schedules different tasks among available logical processors by default. 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. 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 except for the non-maskable interrupts (NMIs), which can't be avoided. However, in our experience, the occurrence of 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. These features, in addition to hyperthreading (concurrent execution of two threads in the same physical core), make the actual measurement of cycles through *rdtsc* less reliable. Cache-side channel attacks that use this cycle counter are affected by the dynamic CPU frequency, as it affects the number of cycles. 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 hyperthreading 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.

4.3.2 Increasing The Time Resolution

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 order to enable and disable our attack as needed at runtime, we install hooks on these two functions. A look at the disassembly of these functions reveals that there is call 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:  callq  0xffffffff81800f30
ffffffff81050905:  push   %rbp

ffffffff81050c90  local_apic_timer_interrupt
ffffffff81050c90:  callq  0xffffffff81800ef0
ffffffff81050c95:  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. 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 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.

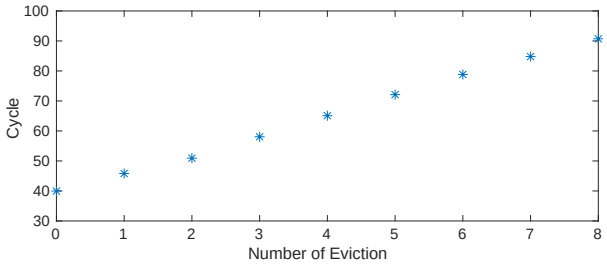

Figure 1: Average cycle count based on the number of evictions in a set.

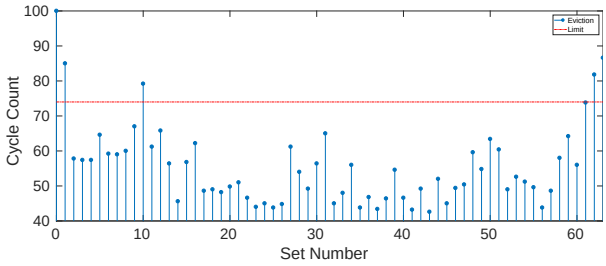

Figure 2: Average cycle count for each sets. Variations are due to channel noise, making 4 sets unusable for our attack.

For this purpose, we enable the attack timer interrupt before the call to the enclave interface and disable it right after.

4.4 Testing the Performance of CacheZoom

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 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.

Even though Skylake processors do not always use the LRU cache replacement policy and have a more adaptive undocumented cache replacement policy [23], 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 1, where the access time is increased by roughly 5 cycles for every additional. Thus, our results show that our eviction policy gives us an accurate measurement on the number of evicted lines from a specific cache set.

Our isolated CPU core and the L1D eviction set have the minimal possible noise and are resistant against various noises such as CPU frequency, OS and enclave 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 2, show that every set has a consistent number of evictions. Among the 64 sets, there are only 4 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 exfiltration process can be observed in Figure 3, in which the enclave process was consecutively accessing different sets. The left hand figure shows the hit access map, without taking into account the appropriate set threshold. The right hand figure shows the access pattern retrieved from the enclave once the context switch noise access has been taking into account and removed.

5 Attack on AES

The following gives a detailed description of different implementation styles for AES to help the reader understand the attacks that we later perform:

5.1 Cache Attacks on Different AES Implementations

AES is a widely used 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. The main leakage source in AES implementations comes from the fact that they utilize state-dependent table look up operations for the SubBytes operation. These lookups result in secret-dependent memory accesses, which can be exploited by cache attacks.

S-Box: Software implementations that implement the 4 stages independently base the SubBytes operation in a 256 entry substitution table, each entry being 8 bits long. In this implementation, a total of a 160 accesses are performed to the S-box during a 128-bit

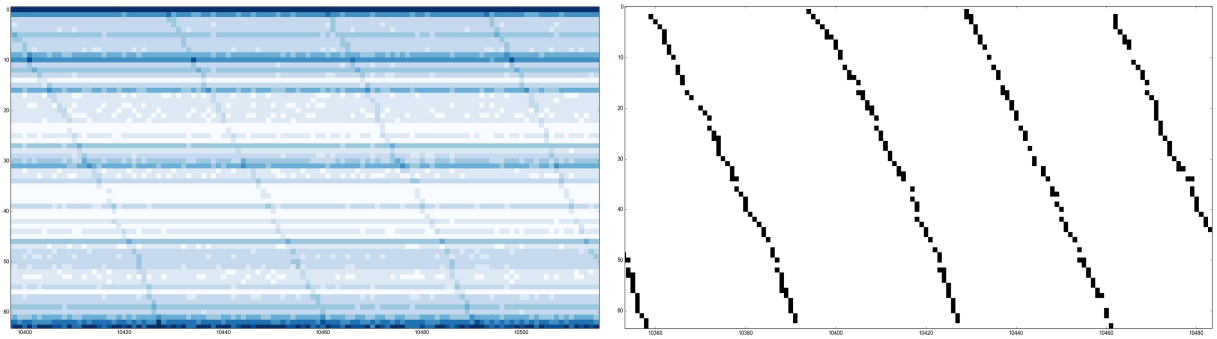

Figure 3: Cache hit map before (top) and after (bottom) filtering for context switch noise. Enclave memory access patterns are clearly visible once standard noise from context switch has been eliminated

AES encryption, 16 accesses per round. We refer to this implementation style as the *S-box* implementation.

4 T-tables: To achieve a better performance, 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. 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 column of the AES state. We refer to this style as *T-table implementation*. We refer to this as the 4 T-table implementation.

Big T-table 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.

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 [29] 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 [29] 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 [17] 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.

5.2 Non-vulnerable AES Implementations

There are further efficient implementations of AES that are not automatically susceptible to cache attacks, as they avoid secret-dependent memory accesses. These implementation styles include bit-sliced implementations [35], implementations using vector instructions [25], constant memory access implementations and implementations using AES instruction set extensions on modern Intel CPUs [27]. However, they 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 [10]. 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 [48], and further might be disabled in BIOS configuration options.

5.3 Cache 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 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. 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 Section 2, attacking SGX gives a malicious OS almost full temporal resolution, which can reverse the effect of prefetching mechanisms.

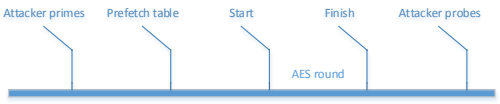

Figure 4: Prefetching and the timeline effect for a regular `Prime and Probe` attack.

6 CacheZooming SGX-based AES

We use CacheZoom to retrieve secret keys of different implementations of AES running inside an enclave. 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 further assume the enclave is performing encryptions over a set of known plaintext bytes or ciphertext bytes.

6.1 T-table Implementations

Our first attacks target the T-table implementations. To recover the AES key from as few traces as possible, we recover the memory access pattern of the first 2 rounds of the AES function. A perfect single trace first round attack 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 the minimal number of traces, we also retrieve the information from the accesses in the second round and use the relation between the first and second round key. In particular, we utilize the relations described in [17], who utilized simulated data to demonstrate the effectiveness of their AES key recovery algorithm.

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

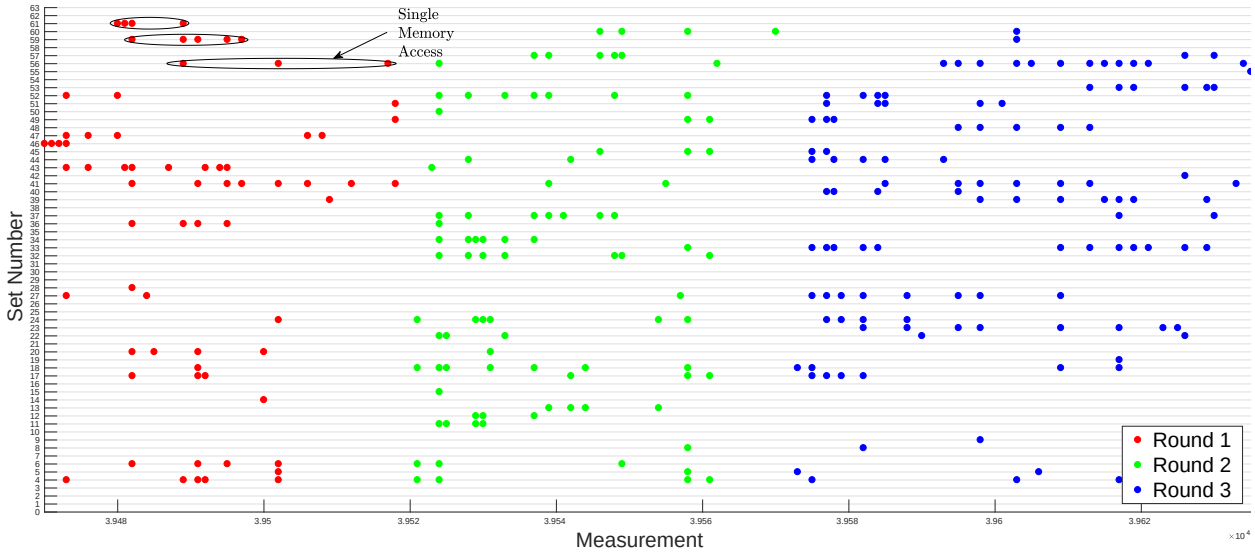

Figure 5: Memory footprint of the AES execution inside enclave.

can be challenging. These first two facts can be observed in Figure 5, which 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 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 every round start involves the utilization of a substantially higher number of sets than the rest, also observable in Figure 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. After applying this filter to the first and second round traces, we apply the key recovery algorithm, as explained in [17]. The accuracy of our measurements with respect to the aforementioned issues can be seen in Table 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 2.

With these statistics and after applying our key recovery algorithms with varying number of traces we obtained the results presented in Figure 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.

Table 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%

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 re-

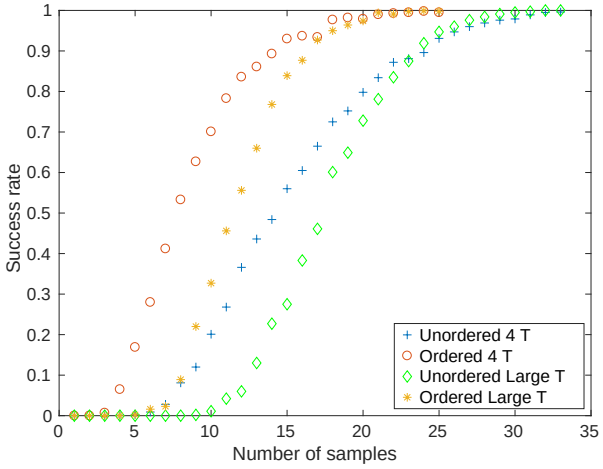

Figure 6: Key recovery success rate.

liable, 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 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.

6.1.1 Cache Prefetching,

as explained in Section 5, is implemented to prevent passive attackers from recovering AES keys. CacheZoom, 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 7. 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, CacheZoom not only bypasses but further benefits from mechanisms that mitigated previous cache attacks.

6.2 S-Box Implementation

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 [15, 24]. However, CacheZoom 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. 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.

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 CacheZoom. Note that due to a lack of alignment, the collection of a large number of observations and the extraction of the last round information still requires manual intervention. When performing the key recovery attack, 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 Fig-

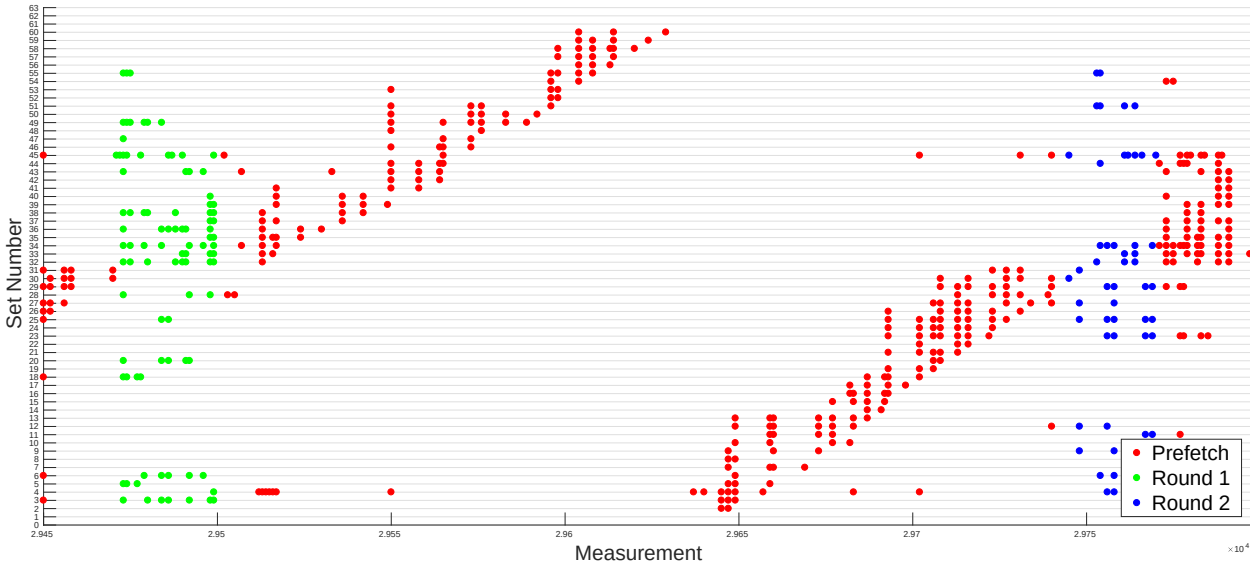

Figure 7: Memory footprint of the AES execution inside an enclave with prefetch countermeasure. The prefetch is clearly distinguishable and helps to identify the start of each round. Further, it also highlights out-of-order execution and in-order completion.

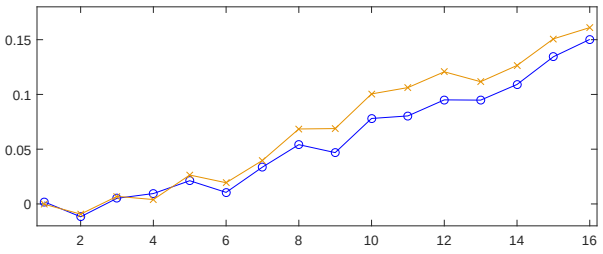

Figure 8: Correlation between observed cache accesses and predicted accesses caused by one byte position. Leakage is much stronger for later byte positions. Correlation for raw observed accesses (blue) vs. relative accesses (amber).

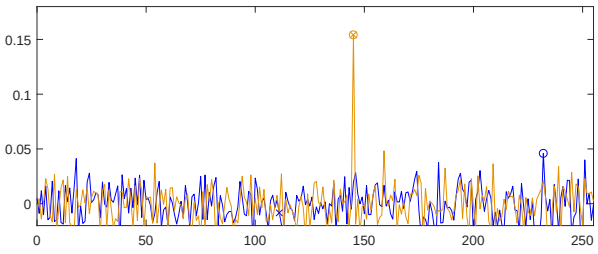

Figure 9: Correlation over key value for the best (k_{15}, amber) and worst (k_0, blue) byte positions based on 1500 traces. The guess with the highest correlation (o) and the correct key (x) a match only for k_{15}.

ure 8. 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 9. 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 result 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 CacheZoom's 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.

7 Conclusion

This work presented CacheZoom, a new tool to analyze memory accesses of SGX enclaves. To gain maximal resolution, CacheZoom combines a L1 cache `Prime and Probe` attack with OS modifications that greatly enhance the time resolution. SGX makes this scenario realistic, as both a modified OS and knowledge of the unencrypted binary are realistic for enclaves. We demonstrate that CacheZoom can be used to recover key bits from all major software AES implementations, including ones that use prefetches for each round as a cache-attack countermeasure. Furthermore, keys can be recovered with as few as 10 observations for T-table based 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.

CacheZoom serves as evidence that security-critical code needs constant execution flows and secret-independent memory accesses. As SGX's 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.

8 Acknowledgments

This work is supported by the National Science Foundation, under the grant CNS-1618837.

References

[1] ARM TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`. Accessed: March 22, 2017.

[2] Intel SGX. `https://software.intel.com/en-us/sgx`. Accessed: March 22, 2017.

[3] ISCA 2015 tutorial slides for Intel SGX. `https://software.intel.com/sites/default/files/332680-002.pdf`. Accessed: March 22, 2017.

[4] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), ACM, pp. 312–320.

[5] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Topics in Cryptology–CT-RSA 2008*. Springer, 2008, pp. 256–273.

[6] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., OKEEFFE, D., STILLWELL, M. L., ET AL. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[7] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[8] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. 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* (2014), Springer, pp. 75–92.

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

[10] BERNSTEIN, D. J., AND SCHWABE, P. *New AES Software Speed Records*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 322–336.

[11] BHATTACHARYA, S., AND MUKHOPADHYAY, D. *Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 248–266.

[12] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2006), Springer, pp. 201–215.

[13] BRENNER, S., WULF, C., GOLTZSCHE, D., WEICHBRODT, N., LORENZ, M., FETZER, C., PIETZUCH, P., AND KAPITZA, R. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference* (2016), ACM, p. 14.

[14] BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive 2006* (2006), 52.

[15] BRIONGOS, S., MALAGÓN, P., RISCO-MARTÍN, J. L., AND MOYA, J. M. Modeling side-channel cache attacks on AES. In *Proceedings of the Summer Computer Simulation Conference, SummerSim 2016, Montreal, QC, Canada, July 24-27, 2016* (2016), p. 37.

[16] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks 48*, 5 (2005), 701–716.

[17] C, A., GIRI, R. P., AND MENEZES, B. Highly efficient algorithms for aes key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)* (March 2016), pp. 261–275.

[18] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Tech. rep., Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[19] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 857–874.

[20] DAHBUR, K., MOHAMMAD, B., AND TARAKJI, A. B. A survey of risks, threats and vulnerabilities in cloud computing. In *Proceedings of the 2011 International conference on intelligent semantic Web-services and applications* (2011), ACM, p. 12.

[21] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. Jump Over ASLR: Attacking Branch PredShared Cache Attictors to Bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[22] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *IACR Eprint* (2016).

[23] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[24] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 490–505.

[25] HAMBURG, M. *Accelerating AES with Vector Permute Instructions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 18–32.

[26] İNCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. *Cache Attacks Enable Bulk Key Recovery on the Cloud*. 2016.

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

[28] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. 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* (2015), IEEE, pp. 591–604.

[29] IRAZOQUI, G., İNCİ, M. S., EISENBARTH, T., AND SUNAR, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.

[30] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[31] LANGNER, R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy 9*, 3 (2011), 49–51.

[32] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. Tech. rep., arxiv Archive, 2016. https://arxiv.org/pdf/1611.06952.pdf.

[33] LIPP, M., GRUSS, D., SPREITZER, R., AND MANGARD, S. Armageddon: Last-level cache attacks on mobile devices. *arXiv preprint arXiv:1511.04897* (2015).

[34] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy* (2015), pp. 605–622.

[35] MATSUI, M., AND NAKAJIMA, J. *On the Power of Bitslice Implementation on Intel Core2 Processor*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 121–134.

[36] MORRIS, T. Trusted platform module. In *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1332–1335.

[37] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security* (2016).

[38] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. 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* (2015), ACM, pp. 1406–1418.

[39] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference* (2006), Springer, pp. 1–20.

[40] PERCIVAL, C. Cache missing for fun and profit, 2005.

[41] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[42] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 38–54.

[43] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *ArXiv e-prints* (Feb. 2017).

[44] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. 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).

[45] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. 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).

[46] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 317–328.

[47] SINHA, R., RAJAMANI, S., SESHIA, S., AND VASWANI, K. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1169–1184.

[48] TAKEHISA, T., NOGAWA, H., AND MORII, M. 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`.

[49] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2003), Springer, pp. 62–76.

[50] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 440–457.

[51] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 640–656.

[52] YAROM, Y., AND FALKNER, K. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 719–732.

[53] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 313–328.

[54] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[55] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*.