Software Grand Exposure: SGX Cache Attacks Are Practical

Ferdinand Brasser[1], Urs Müller[2], Alexandra Dmitrienko[2], Kari Kostiainen[2], Srdjan Capkun[2], and Ahmad-Reza Sadeghi[1]

[1]System Security Lab, Technische Universität Darmstadt, Germany
{ferdinand.brasser,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]Institute of Information Security, ETH Zurich, Switzerland
muurs@student.ethz.ch, {alexandra.dmitrienko,kari.kostiainen,srdjan.capkun}@inf.ethz.ch

Abstract

Intel SGX isolates the memory of security-critical applications from the untrusted OS. However, it has been speculated that SGX may be vulnerable to side-channel attacks through shared caches. We developed new cache attack techniques customized for SGX. Our attack differs from other SGX cache attacks in that it is easy to deploy and avoids known detection approaches. We demonstrate the effectiveness of our attack on two case studies: RSA decryption and genomic processing. While cache timing attacks against RSA and other cryptographic operations can be prevented by using appropriately hardened crypto libraries, the same cannot be easily done for other computations, such as genomic processing. Our second case study therefore shows that attacks on non-cryptographic but privacy sensitive operations are a serious threat. We analyze countermeasures and show that none of the known defenses eliminates the attack.

1 Introduction

Intel Software Guard Extension (SGX) [14, 23] enables execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software. SGX provides protections in the processor to prevent a malicious OS from directly reading or modifying enclave memory at runtime. The architecture is especially useful in cloud computing applications, where data and computation can be outsourced to an external computing infrastructure without having to fully trust the cloud provider and the entire software stack.

However, researchers have recently demonstrated that SGX isolation can be attacked by exploiting information leakage through various (software) side-channels. One type of information leakage is based on page faults: In SGX, memory management (including paging) is left to the untrusted OS [14]. Consequently, the OS can force page faults at any point of enclave execution and from the requested pages learn the secret-dependent enclave control flow or data access patterns [53]. Another type of information leakage is based on overseeing caches shared between the enclave and the untrusted software, as pointed out in [14] and by Intel [27, p. 35]. Cache attacks have been studied extensively independent of SGX [43, 39, 30, 36, 54, 21, 20]. Recently, a number of cache-based attacks targeted SGX platforms [44, 38, 19].

To tackle the information leakage problem in SGX, different countermeasures have also been proposed. A promising *system-level* approach is to detect when the OS is intervening with enclave execution as done in T-SGX [46] and Déjá Vu [10]. These solutions detect page faults and allow the enclave to defend itself from a possible attack (i.e., to stop its execution). Another approach against information leakage is obviously hardware redesign as taken by Sanctum [15]. Although new hardware design like Sanctum is out of our scope, we will elaborate on it in Section 6.

Our goals and contributions. First, we explore novel cache attack techniques customized for SGX that are easier to deploy than other SGX cache side-channel attacks and are significantly harder to detect/prevent, particularly by the recently proposed defenses [46, 10] mentioned above. Second, we demonstrate that information leakage is a serious concern, since it can defeat one of the core benefits of SGX, namely, secure computation over sensitive data on an untrusted platform. We show this on two case studies: first a cryptographic primitive and then a non-cryptographic privacy-preserving algorithm.

Novel attack techniques. Our attack enables the adversary to run both the victim enclave and its own process uninterrupted in parallel, so that the victim enclave is unaware of the attack and cannot take measures to defend itself. Uninterrupted attack execution imposes technical challenges such as dealing with significant noise in cache monitoring. To realize our attack effectively in this setting, we needed to develop a set of novel attack techniques. For instance, we leverage the capabilities of the privileged adversary to assign the victim process to a dedicated core, reduce the number of benign interrupts,

and perform precise cache monitoring using CPU performance counters. Note that the SGX adversary model includes the capabilities of the OS.

Our attack differs from other recently proposed cache-based attacks on SGX [44, 38, 19] in various ways: CacheZoom [38] interrupts the victim repeatedly and can therefore be easily detected by the above mentioned countermeasures T-SGX [46] and Déjá Vu [10]. Gotzfried et al. [19] require synchronization between the victim enclave and the attacker. Schwartz et al. [44] implement their attack on the L3 cache (i.e., cross-core attack). Our attack works on the L1 cache (i.e., same-core attack) and does not require interrupts or synchrony between the victim and the attacker, which makes it significantly harder to detect and easier to deploy in practice. We provide a more detailed comparison in Section 7.

Case studies. We show the effectiveness of our attack techniques for two different case studies. The first case study is the canonical example of RSA decryption where we extract 70% of the private key bits with approximately 300 repeated decryptions (70% is sufficient to recover the entire private key efficiently).

However, cache attacks can principally be mitigated at the *application level*. In particular, many recent cryptographic libraries provide implementations that have been hardened against cache monitoring. For example, the *scatter-and-gather* technique [8] is a widely deployed protection, where every secret-dependent lookup table access is manually changed to touch memory addresses corresponding to all monitored cache sets. Hence, the accessed table element is effectively hidden from the adversary. Also the SGX SDK includes cryptographic algorithm variants that use the scatter-gather protection [28]. Thus, cache attacks on cryptographic enclaves may not be a major threat in practice.

On the other hand, a more significant concern, and a problem that has not been studied extensively in the past, is information leakage of various and probably more complex computations that are not developed by security experts. While manual defenses like scatter-gather can effectively prevent cache attacks, they require significant expertise and effort from the developer. It seems unrealistic to assume that every enclave developer is aware of possible information leakage and able to manually harden his implementation against cache monitoring. Hence, as the second case study we demonstrate information leakage from non-cryptographic, but privacy-sensitive enclave for a genome indexing algorithm called PRIMEX [34] that uses a hash table to index a genome sequence. By monitoring the genome-dependent hash table accesses we can identify if the processed human genome (DNA) includes particular sequences that are often used in forensic analysis and genomic fingerprinting [4]. We show that the information leaked through caches during indexing is sufficient to identify the person whose DNA is processed with high probability.

We argue that large classes of SGX enclaves, and therefore many practical cloud computing scenarios, are vulnerable to similar information leakage. Our analysis on existing countermeasures shows that none of the known defenses effectively prevents our attack.

Contributions. To summarize, this paper makes the following main contributions:

- **Novel SGX cache attack techniques.** We demonstrate that cache attacks are practical on SGX. In particular, we develop novel cache attack techniques for SGX that are easier to deploy and significantly harder to detect/prevent.
- **Leakage from non-cryptographic applications.** Through a case study on a genomic processing enclave we show that non-cryptographic, but privacy-sensitive applications deployed as SGX enclaves are vulnerable to cache attacks.
- **Countermeasure analysis.** We show that none of the known defenses mitigates our attack in practice.

The rest of the paper is organized as follows. In Section 2 we provide background information. Section 3 introduces the system and adversary model and Section 4 explains the attack design. Section 5 summarizes our RSA attack and details the genomic case study. We analyze countermeasures in Section 6, review related work in Section 7, and draw conclusions in Section 8.

2 Background

This section provides the necessary background on Intel SGX, cache architecture and performance monitoring counters.

2.1 Intel SGX

SGX introduces a set of new CPU instructions for creating and managing isolated software components [37, 25], called *enclaves*, that are isolated from all software running on the system including privileged software like the operating system (OS) and the hypervisor. SGX assumes the CPU itself to be the only trustworthy hardware component of the system, i.e., enclave data is handled in plain-text only *inside* the CPU. Data is stored unencrypted in the CPU's caches and registers, however, whenever data is moved out of the CPU, e.g., into the DRAM, it is encrypted and integrity protected.

The OS, although untrusted, is responsible for creating and managing enclaves. It allocates memory for the enclaves, manages virtual to physical address translation for the enclave's memory and copies the initial data and code into the enclave. However, all actions of the OS are recorded securely by SGX and can be verified by an external party through (remote) attestation [3]. SGX's seal-

ing capability enables persistent secure storage of data.

During enclave execution the OS can interrupt and resume the enclave like a normal process. To prevent information leakage, SGX handles the context saving of enclaves in hardware and erases the register content before passing control to the OS, called asynchronous enclave exit (AEX). When an enclave is resumed, again the hardware is responsible for restoring the enclave's context, preventing manipulations.

2.2 Cache Architecture

In the following we provide details of the Intel x86 cache architecture [26, 24].[1] We focus on the Intel Skylake processor generation, i.e., the type of CPU we used for our implementation and evaluation.[2]

Memory caching "hides" the latency of memory accesses to the system's dynamic random access memory (DRAM) by keeping a copy of currently processed data in cache. When a memory operation is performed, the cache controller checks whether the requested data is already cached, and if so, the request is served from the cache, called a *cache hit*, otherwise *cache miss*. Due to higher cost (production, energy consumption), caches are orders of magnitude smaller than DRAM and only a subset of the memory content can be present in the cache at any point in time. The cache controller aims to maximize the cache hit rate by predicting which data are used next by the CPU. This prediction is based on the assumption of temporal and spatial locality of memory accesses.

For each memory access the cache controller has to check if the data are present in the cache. Sequentially iterating through the entire cache would be very expensive. Therefore, the cache is divided into *cache lines* and for each memory address the corresponding cache line can be quickly determined, the lower bits of a memory address select the cache line. Hence, multiple memory addresses map to the same cache line. Having one cache entry per cache line quickly leads to conflicts and the controller has to evict data from cache to replace it with newly requested data. To minimize such conflicts caches are often (set) associative. Multiple copies of each cache line exist in parallel, also known as *cache sets*, thus #*cachesets* many data from conflicting memory locations can stay in the cache simultaneously.

The current Intel CPUs have a three level hierarchy of caches. The last level cache (LLC), also known as level 3 (L3) cache, is the largest and slowest cache; it is shared between all CPU-cores. Each CPU core has a dedicated L1 and L2 cache, but they are shared between the core's simultaneous multi-threading (SMT) execution units (also known as hyper-threading).

A unique feature of the L1 cache is the separation into data and instruction cache. Code fetches only affect the instruction cache and leave the data cache unmodified, and vice versa. In the L2 and L3 caches code memory and data memory compete for the available cache space.

2.3 Performance Monitoring Counters

Performance Monitoring Counters (PMC) are a feature of the CPU for recording hardware events. Their primary goal is to give software developers insight into their program's effects on the hardware in order for them to optimize their programs.

The CPU has a set of PMCs, which can be configured to monitor different events, for instance, executed cycles, cache hits or cache misses for the different caches, mispredicted branches, etc. PMCs are configured by selecting the event to monitor as well as the mode of operation. This is done by writing to model specific registers (MSR), which can only be done by privileged software. PMCs are read via the `RDPMC` instruction (read performance monitoring counters), which can be configured to be available in unprivileged mode.

Hardware events recorded by PMCs could be misused as side-channels, e.g., to monitor cache hits or misses of a victim process or enclave. Therefore, SGX enclaves can disable PMCs on entry by activating a feature called "Anti Side-channel Interference" (ASCI) [26]. This suppresses all thread-specific performance monitoring, except for fixed cycle counters. Hence, hardware events triggered by an enclave cannot be monitored through the PMC feature. For instance, cache misses of memory loaded by an enclave will not be recorded in the PMCs.

3 System and Adversary Model

We assume a system equipped with Intel SGX, i.e., a hardware mechanism to isolate data and execution of a software component from the rest of the system's software that is considered untrusted. The resources which are used to execute the isolated component (or enclave), however, are shared with the untrusted software on the system. The system's resources are managed by untrusted, privileged software (operating system – OS). Figure 1 shows an abstract view of the adversary model, an enclave executing on a system with a compromised OS, sharing a CPU core with an attacker process.

The adversary's objective is to learn secret information from the enclave, e.g., a secret key generated *inside* the enclave through a hardware random number generator, or sensitive data supplied to the enclave *after* initialization through a secure channel.

Adversary capabilities. The adversary is in control of all system software, except for the software executed in-

[1] We will use the terminology from Intel documents [1].

[2] At the time of writing Intel SGX is available only on Intel Skylake and Kaby Lake CPUs. To the best of our knowledge there are no differences in the cache architecture between Skylake and Kaby Lake.

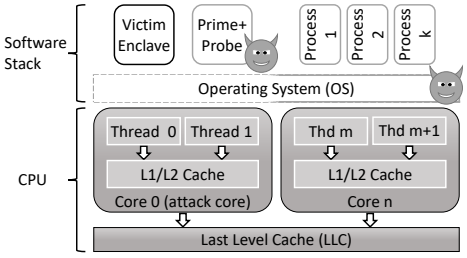

Figure 1: High-level view of our attack; victim and attacker's Prime+Probe code run in parallel on a dedicated core. The malicious OS ensures that no other code shares that core minimizing noise in L1/L2 cache.

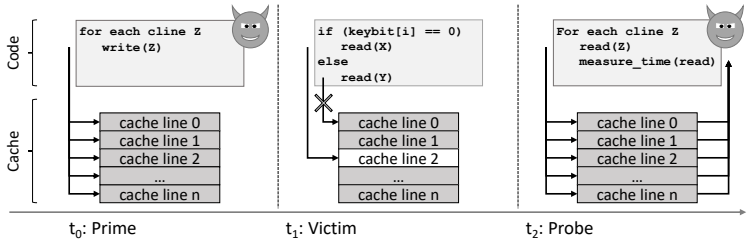

Figure 2: Prime+Probe side-channel attack technique; first the attacker primes the cache, next the victim executes and occupies some of the cache, afterwards the attacker probes to identify which cache lines have been used by the victim. This information allows the attacker to draw conclusion on secret data processed by the victim process.

side the enclave.[3] Although the attacker cannot control the program inside the enclave, he does know the initial state of the enclave, i.e., the program code of the enclave and its initial data. In particular, randomization through mechanisms like address space layout randomization (ASLR) are visible to the adversary. The attacker knows the mapping of memory addresses to cache lines and can reinitialize the enclave and replay inputs, hence, he can run the enclave arbitrarily often. Further, since the adversary has control over the OS he controls the allocation of resources to the enclave, including the time of execution, and the processing unit (CPU core) the enclave is running on. Similarly, the adversary can configure the system's hardware arbitrarily, e.g., define the system's behavior on interrupts, or set the frequency of timers. However, the adversary cannot directly access the memory of an enclave. Moreover, he cannot retrieve the register state of an enclave, neither during the enclave's execution nor on interrupts.

Attack goals. The adversary aims to learn about the victim's cache usage by observing effects on the cache availability to its own program. In particular, he leverages the knowledge of the mapping of cache lines to memory locations in order to infer information about access patterns of the enclave to the secret-dependent memory locations, which in turn allows him to draw conclusions about sensitive data processed by the victim. We show two concrete attacks for recovering an RSA key and identifying individuals in genome processing applications in Section 5.

4 Our Attack Design

Our attack technique is based on the Prime+Probe cache monitoring technique [39]. We will first explain the "classical" variant of Prime+Probe, then we discuss our improvements beyond the basic approach.

[3]Due to integrity verification, the adversary cannot modify the software executed inside the enclave, since SGX remote attestation would reveal tempering.

4.1 Prime+Probe

The main steps of the Prime+Probe attack are depicted in Figure 2. First, at time t_0, the attacker *primes* the cache, i.e., the attacker accesses memory such that the entire cache is filled with data of the attacker process.[4] Afterwards, at time t_1, the victim executes code with memory accesses that are dependent on the victim's sensitive data, like a cryptographic key. The victim accesses different memory locations depending on the currently processed key-bit. In the example in Figure 2 the key-bit is zero, therefore address *X* is read. Address *X* is mapped to *cache line 2*, hence, the data stored at *X* are loaded into the cache and the data that were present in *cache line 2* before get evicted. The data at address *Y* are not accessed and therefore the data in *cache line 0* remains unchanged.

At time t_2 the attacker *probes* which of his cache lines got evicted, i.e., which cache lines were used by the victim. A common technique to check for cache line eviction is to measure access times. The attacker reads from memory mapped to each cache line and measures the access time. If the attacker's data are still in the cache the read operation returns them fast, if the read operation takes longer, the data were evicted from the cache. In Figure 2, the attacker will observe an increased access time for *cache line 2*. Since the attacker knows the code and access pattern of the victim, he knows that address *X* of the victim maps to *cache line 2*, and that the sensitive key-bit must be zero. This cycle is repeated by the attacker for each sensitive key-bit that is processed by the victim until the attacker learns the entire key.

4.2 Prime+Probe for SGX

Cache monitoring techniques, like Prime+Probe, experience significant noise. Therefore, most of the previously reported attacks (that, e.g., extract a full cryptographic key) require thousands and even millions of re-

[4]To prime all cache sets the attacker needs to write to #*cachesets* cache pages, see Section 2.2 for details.

peated executions to average out the noise (e.g., [55, 56]). Our goal is build an efficient attack, i.e., one that works with much fewer executions. The key to this is reducing noise (or pollution) in the cache monitoring channel.

There are two main aspects that guide our selection of possible noise reduction techniques — and also distinguish us from most of the previous attacks. (1) Our goal is to build an attack that cannot be easily detected using the recently proposed detection approaches [46, 10]; this requirement limits the possible noise reduction techniques we can use (e.g., no interrupts). (2) In our setting the adversary is the privileged OS; this condition enables us to leverage new methods that were previously inaccessible to the attacker (e.g., performance counters).

Challenges. Given these conditions, we list the main challenges in our attack realization.

1. Minimizing cache pollution caused by other tasks.
2. Minimizing cache pollution by the victim itself.
3. Uninterrupted victim execution to counter side-channel protection techniques and prevent cache pollution by the OS.
4. Reliably identifying cache evictions.
5. Performing cache monitoring at a high frequency.

Next, we describe a set of new attack techniques that we developed to address each of the challenges above.

4.3 Noise Reduction Techniques

(1.) Isolated attack core. We isolate the attack core from other processes in order to minimize the noise in the side channel. Figure 1 shows our approach to isolate the victim enclave on a dedicated CPU core, which only executes the victim and our attacker Prime+Probe code.

By default Linux schedules all processes of a system to run on any available CPU core, hence, impacting all caches. The attacker cannot distinguish between cache evictions caused by the victim and those caused by any other process. By modifying the Linux scheduler, the adversary can make sure that one core (we call it *attacker core*) is exclusively used by the victim and the attacker ("Core 0" in Figure 1). This way no other process can pollute this core's L1/L2 cache.

(2.) Self-pollution. The attacker needs to observe specific cache lines that correspond to memory locations relevant for the attack. From the attacker's point of view it is undesirable if those cache lines are used by the victim for any other reason than accessing these specific memory locations, e.g., by accessing unrelated data or code that map to the same cache line.

In our attack we use the L1 cache. It has the advantage of being divided into a data cache (L1D) and an instruction cache (L1I). Therefore, code accesses, regardless of the memory location of the code, never map to the cache lines of interest to the attacker. Victim accesses to unrelated data mapping to relevant cache lines leads to noise in the side channel. This noise source cannot be influenced by the attacker given that the memory layout of the victim is fixed.

(3.) Uninterrupted execution. Interrupting the victim enclave yields two relevant problems. (1) When an enclave is interrupted, an asynchronous enclave exit (AEX) is performed and the operating system's interrupt service routine (ISR) in invoked (see Section 2.1). Both, the AEX and the ISR use the cache, and hence, induce noise. (2) By means of transactional memory accesses an enclave can detect that it has been interrupted. This feature has been used for a side channel defense mechanism [46, 10]. We discuss the details in Section 6. Hence, making the enclave execute uninterrupted ensures that the enclave remains unaware of the side-channel attack.

In order to monitor the changes in the victim's cache throughout the execution, we need to access the cache of the attack core in parallel. For this we execute the attacker code on the same core. The victim is running on the first SMT (simultaneous multi-threading) execution unit while the attacker is running on the second SMT execution unit (see Figure 1). As the victim and attacker code compete for the L1 cache, the attacker can observe the victim's effect on the cache.

The attacker code is, like the victim code, executed uninterrupted by the OS. Interrupts usually occur at a high frequency, e.g., due to arriving network packages, user input, etc. By default interrupts are handled by all available CPU cores, including the attack core, and thus the victim and attacker code are likely to be interrupted.

To overcome this problem we configured the interrupt controller such that interrupts are not delivered to the attack core, i.e., it can run uninterrupted. The only exception is the timer interrupt which is delivered per-core. Each CPU core has a dedicated timer and the interrupt generated by the timer can only be handled by the associated core. However, we reduced the interrupt frequency of the timer to $100Hz$, which allows victim and attacker code to run for $10ms$ uninterrupted. This time frame is sufficiently large to run the complete attack cycle undisturbed (with high probability).[5] As a result, the OS is not executed on the attack core while the attack is in progress (depicted by the dashed-line OS-box in Figure 1). Also, the victim is not interrupted, thus, it remains unaware of the attack.

(4.) Monitoring cache evictions. In the previous Prime+Probe attacks, the attacker determines the eviction of a cache line by measuring the time required for accessing memory that maps to that cache line. These timing based measurements represent an additional source

[5] When an interrupt occurs, by chance, the attack can be repeated. If the time frame is too short the timer frequency can be reduced further.

of noise to the side channel. Distinguishing between cache hit and miss requires precise time measurements. For instance for the L1 cache a cache hit takes at least 4 *cycles*. If the data got evicted from the L1 cache, they can still be present in the L2 cache and read from there, which takes 12 *cycles* in the best case.[6] This small difference in access times makes it challenging to distinguish a cache hit in L1 cache and a cache miss in L1 that is served from L2 cache. Reading the time stamp counter itself suffers from noise which is in the order of the difference between L1 and L2 cache accesses. Thus, when the timing measurement does not allow for a definitive distinction between a cache hit and a cache miss, the observation has to be discarded. To eliminate this noise we use Performance Monitoring Counters (PMC) to determine if a cache line got evicted by the victim. This is possible in the SGX adversary model because the attacker controls the OS and can freely configure and use the PMCs.

Usage of performance counters for cache attacks was previously explored in [49, 6]. For instance, [49] demonstrated that measurements collected using PMC and L1 cache misses require the least amount of traces compare to other measurement methods, such as time stamp counters. One should, however, note that PMCs are only beneficial if adversary is privileged, but cannot directly read the victim memory. To the best of our knowledge, we are the first to use PMCs in such a setting.

We recall that Intel processors provide Anti Side-Channel Interference (ASCI) feature (cf. Section 2.3) that prevents monitoring of cache related events caused by enclave execution. This, however, does not prevent our attack, since we do not monitor cache activity of the victim, but instead observe cache events of the attacker process, which shares the cache with the victim.

(5.) Monitoring frequency. As discussed before, the victim should run uninterrupted while its cache accesses are monitored in parallel. Hence, we need to execute priming and probing of the cache at a high frequency to not miss relevant cache events. In particular, probing each cache line to decide whether it has been evicted by the victim is time consuming and leads to a reduced sampling rate. The required monitoring frequency depends on the frequency at which the victim is accessing the secret-dependent memory locations. To not miss any access the attacker has to complete one prime and probe cycle before the next access occurs. In our implementation the access to PMCs is the most expensive operation in the Prime+Probe cycle.

To tackle this challenge we monitor individual (or a small subset of) the cache lines over the course of multiple executions of the victim. In the first run we learn the victim's accesses to the first cache line, in the second run accesses to the second cache line, and so on. By aligning the results of all runs we learn the complete cache access pattern of the victim.

[6] Reported values for Skylake architecture, however, "Software-visible latency will vary depending on access patterns and other factors" [24].

5 Attack Instantiations

We implemented and evaluated two concrete attacks. Our evaluation platform was a Dell Latitude E5470 with an Intel Core i7-6600U CPU @ 2.60 GHz running Linux 14.04 with a custom 4.4.0-57 kernel and Intel SGX software developer kit (SDK) version 1.6.

RSA attack. Our first attack targets the RSA decryption. Since cache attacks on cryptographic algorithms are well understood from previous literature [43, 39, 30, 36, 54, 21, 20], we only summarize our RSA attack here. The full details can be found in an extended version of this paper [7].

We attacked a standard fixed-window RSA implementation from the SGX SDK version 1.6 [29]. The implementation uses the Chinese Remainder Theorem (CRT) optimization and performs two 1024-bit exponentiations per decryption. By monitoring private key dependent memory accesses to a pre-computed multiplier table we were able to extract 70% of the complete 2048-bit RSA key with 300 repeated decryptions. From the extracted bits, the full RSA key can be effectively recovered [22]. We note that the adversary can easily repeat decryption with the same ciphertext, since she controls the OS ()re-run the enclave and replay all inputs).

Comparable attacks on cryptographic implementations are described in parallel work [44, 38, 19]. We describe the main differences to our attack in Section 7.

Genomic attack. As already mentioned, many cryptographic libraries are hardened against cache monitoring (e.g., scatter-gather technique), and thus such attacks can be prevented by using a suitable cryptographic library. Unfortunately, similar protections cannot be easily added to all enclaves, as manual application hardening requires both developer expertise and effort. Consequently, many non-cryptographic, but privacy-sensitive SGX applications remain vulnerable to cache attacks. We demonstrate this problem using a genomic processing enclave as our second attack case study.

Genome data analysis is an emerging field that highly benefits from cloud computing due to the large amounts of data being processed. At the same time, genome data is highly sensitive, as they may allow the identification of persons and carry information about a person's predisposition to specific diseases. Thus, maintaining the confidentiality of genomic data is paramount, in particular when processed in untrusted cloud environments.

Genome sequences are represented by the order of the

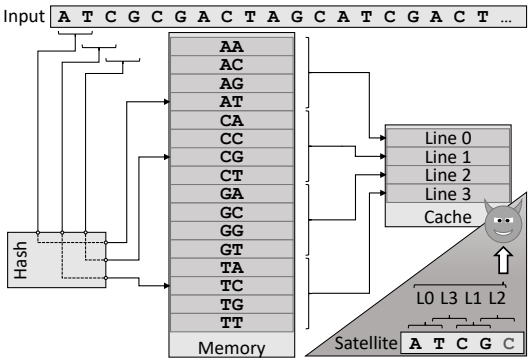

Figure 3: Genome sequence analysis based on hash tables; subsequences positions of the genome (called *k*-mers) are inserted into a hash table for statistical analysis and fast search for *k*-mers.

four nucleotides adenine, cytosine, guanine and thymine, usually abbreviated by their first letter (A, C, G, T). Microsatellites or short tandem repeats (STR) are repetitive nucleotides base sequences. STR analysis is a common genomic forensics technique, where the length of the microsatellite at specific locations are used to identify an individual [9]. For example, many US forensics labs use STR lengths in 13 standardized locations to define a genotype for an individual.

5.1 Victim Enclave

Efficient search of genome sequences is vital for many analysis methods. Therefore, the genomic data is usually preprocessed before the actual analysis is performed. One common way of preprocessing is to divide the genome sequence into substrings of a fixed length *k*, called *k-mer*. The *k*-mers represent a sliding window over the input string of genome bases.

In Figure 3 the input `AGCGC...` is split into 2-mers. Starting from the left the first is `AG`, next the sliding window is moved by one character resulting in the second 2-mer `GC`, and so on. The *k*-mers are inserted into a hash table, usually, for each *k*-mer its position in the genome sequence is stored in the hash table. Thus, given a *k*-mer that is part of a microsatellite one can quickly lookup at which position it appears in the input genome sequence.

Another use case is statistics of the input genome sequence, for instance, the distribution of *k*-mers in the sequence can easily be extracted from the hash table.

Primex. Our victim enclave implements the preprocessing step for a genome sequence analysis, as described above. We used an open source *k*-mer analysis tool called PRIMEX [34]. The tool inserts each *k*-mer position into the hash table. Each hash table entry holds a pointer to an array, which is used to store the positions of each *k*-mer.

Algorithm 1 Hash-Index Generation

Require: Genome G with $G_i \in \{A,C,G,T\}$, $k \in \mathbb{N}_{>0}$
Ensure: Hash-Index H

```
Let H ← HashTable with 4^k entries
for each k-mer M ∈ G do
    Let pos be the offset of M in G
    Let idx ← 0
    for each nucleotide n ∈ M do
        switch n do
            case A: ñ ← 0
            case C: ñ ← 1
            case G: ñ ← 2
            case T: ñ ← 3
        idx ← 4 · idx + ñ
    end for
    H[idx].append(pos)
end for
```

5.2 Attack Details

Our attack aims at leaking the length of microsatellites at the standardized locations used for STR analysis when an input genome sequence is preprocessed (indexed) by the victim enclave. Due to the controlled environment of our attack the execution time of the victim is deterministic, allowing precisely correlating of cache monitoring observations with position in the input sequence.

Through our cache side channel we can observe cache activities that can be linked to the victim's insertion operation into the hash table (Algorithm 1). Figure 3 shows that insertions into the hash table affect different cache lines. For each *k*-mer the victim looks up a pointer to the respective array from the hash table. From the source code we learn the hash function used to determine the table index for each *k*-mer, and by reversing this mapping we can infer the input based on the accessed table index.

Unfortunately, individual table entries do not map to unique cache lines. Multiple table entries fit within one cache line, so from observing the cache line accesses we cannot directly conclude which index was accessed. This problem is illustrated in Figure 3. Here four table indexes map to a single cache line. When the attacker observes the eviction of cache line 0, it does not learn the exact table index of the inserted *k*-mer, but a set of candidate *k*-mers that could have been inserted ($\{$`AA,AC,AG,AT`$\}$).

However, the attacker can split up the microsatellite he is interested in into *k*-mers and determine which cache lines will be used when it appears in the input sequence. In Figure 3 the microsatellite is split into four 2-mers, where the position of the first 2-mer (`AT`) will be inserted in the first quarter of the table, hence, cache line 0 will be used by the victim enclave. The position of the second 2-mer (`TC`) will be inserted into the last quarter of the hash table, thus activating cache line 3. Following this scheme the attacker learns a sequence of cache lines used by the enclave, which will reveal to her that sequence of processed *k*-mers.

5.3 Attack Results

We provided a genome sequence string to the victim enclave and ran it in parallel to our attack code. We chose $k = 4$ for the k-mers leading to $4^4 = 256$ 4-mers (four nucleotides possible for each of the four position). Each 4-mer is represented by a unique table entry, each table entry is a pointer ($8\,byte$), and thus each cache line contains $64\,byte/8\,byte = 8$ table entries.

The attack attempts to determine the microsatellite length at each of the 13 standardized locations. For example, in location CSF1PO we try to extract the length of the expected repeating sequence `TAGA`. First, the four 4-mers occurring repeatedly in the microsatellite are determined, and for each 4-mer the corresponding cache lines: `TAGA` $\Rightarrow$ cache line 7; `AGAT` $\Rightarrow$ cache line 28; `GATA` $\Rightarrow$ cache line 9; `ATAG` $\Rightarrow$ cache line 20.

We monitor these four cache lines individually and align them, as shown in Figure 4. When the microsatellite appears in the input string, the cache lines 7, 28, 9 and 20 will all be used repeatedly by the victim enclave. This increase in utilization of these cache sets can be observed in the measurements. In Figure 4 the increased density of observed cache events is visible, marked by the solid line rectangle. Since all four cache lines are active at the same time, one can conclude that the microsatellite did occur in the input sequence.

False positives. False positives due to monitoring noise are very unlikely due to the fact that we are observing four cache lines. Figure 4 shows extensive activation in the top cache line (pink) marked by the dashed line rectangle. However, in all three other cache lines there is low activity making this event clearly distinguishable from a true positive event.

Length accuracy. For STR analysis one should, ideally, know the exact microsatellite length at sufficiently many standard locations. Due to cache monitoring noise, as seen in Figure 4, our attack is unable to extract precise microsatellite lengths.

We determine possible microsatellite lengths by manually comparing the attack traces (see Figure 4) to precomputed reference traces for known lengths. Through a manual verification we confirm that our attack is able to extract the length with an accuracy of ± 1 in the vast majority of test samples. That is, if the correct length is 11 in a location CSF1PO, our attack gives us three possible alternatives (10, 11, 12) for that location.

Target identification. Given the three possible lengths for each standard location, one can compute the probability that a random person from a given population would match the leaked genomic information as the Combined Probability of Inclusion (CPI) [9]. The probability of identifying an individual depends on the genotype of the attack target. The worst case, and therefore the lower bound for attack accuracy, is when the target individual has the most common genotype, i.e., the most frequently seen microsatellite length in each of the 13 locations. Assuming that the attack target is a Caucasian person with the most common genotype, from the known frequencies [9, Chapter 11] we compute a CPI value 7.027×10^{-5}. This means that from a population of 10 million people (e.g., a small country), statistically 703 people would have a genotype that matches the information leaked through the attack.

The average case is one where the attack target has more variation in his genotype. In the majority of such cases, statistically less than one person from a population of 10 million match the leaked information. We conclude that our attack is, therefore, able to identify the person whose genome is processed with high probability.

Other applications. Similar information leakage is likely to apply to many other applications as well. Especially, programs that build or access lookup tables, or similar data structures, are ideal targets for our attack. Such patterns are often seen in database systems, medical and scientific data processing applications, and machine learning models.

6 Countermeasure Analysis

In this section we discuss potential countermeasures against cache-based side-channel attacks and elaborate on their applicability to protection of SGX enclaves.

Cache disabling. The most straightforward countermeasure against cache-based side channels is to disable caching entirely [2], which, however, would lead to severe performance degradation. Even cache disabling during enclave execution only might be prohibitively expensive, given that enclaves may need to process large datasets (e.g., human DNA), perform expensive computation (e.g., cryptography), or run large applications [5].

Architectural changes to cache organization. Another approach to mitigate cache-based side channels is to introduce countermeasures through redesign of the cache hardware. Respective techniques largely fall into two categories, the first one relying on access randomization within cache memory [51, 52, 31, 35], and the second one using cache partitioning, so that security sensitive code never shares caches with untrusted processes [40, 41, 50, 51, 17]. Moreover, hardware approaches can also co-exist with software defenses. For instance, the Sanctum [15] architecture, which provides protected enclave execution for RISC-V platforms, applies cache partitioning for the last level cache (LLC), while flushing the per-core L1 cache upon enclave exit.[7] However,

[7] Flushing is sufficient to ensure that L1 it is never shared between an enclave and any other code on systems like Sanctum, that do not support simultaneous multi-threading SMT (or hyper-threading). On Intel

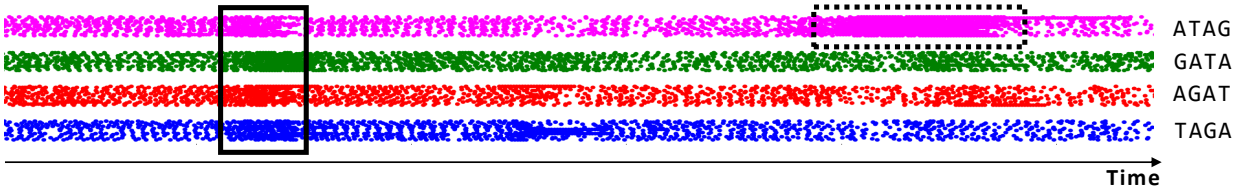

Figure 4: Access pattern of hash table accesses by PRIMEX processing a genome sequence [34]. Four cache sets are shown in different colors with 20 repeated measured for each cache set. The cache sets are correspond to the 4-mers of the microsatellite TAGA. Increased activity in all four cache sets (marked by the solid line rectangle) indicates the occurrence of the microsatellite in the processed genome sequence.

hardware changes can only be incorporated by hardware manufacturers, which is hard to achieve in practice. In particular, Intel SGX does not incorporate any protections against side-channel attacks at the architectural level.

Obfuscation techniques. Oblivious RAM (ORAM) [18, 47, 48], hides memory access patterns of programs by continuously shuffling and re-encrypting data as they are accessed in RAM memory, disk or from a remote server. ORAM is typically applied in server-client models, and requires the client to store some state that is updated throughout the execution. While one could think of using similar techniques for cache protection, they are not directly applicable, as it is challenging to store ORAM-internal state securely. Without hardware support this would require storing client state in a cache side-channel oblivious way, which is unfeasible given the small size of every cache line.

Other obfuscation techniques suggest to perform periodic scrubbing and flushes of shared caches [57] or add noise to memory accesses [40, 39] to interfere with the signal observable by the attacker. These techniques, however, introduce a significant overhead and are less effective on systems supporting simultaneous multithreading, where two threads or processes can be executed literally simultaneously, not in a time-sharing fashion. In this case the attacker process running in parallel with the victim can still observe memory access patterns between scrubbing and flushing rounds. Furthermore, an attacker may collect multiple execution traces and process them to filter out the injected noise.

Application hardening. Application-level hardening techniques modify application code to protect secrets from side-channel leakage. Such solutions can be classified into two categories: (i) Side-channel free implementations (e.g., for cryptographic algorithms, such as AES and RSA [8, 32]) and (ii) automated transformation tools that can be applied to existing programs [13, 12, 16]. Side-channel free implementations, like scatter-gather, are application-specific and require significant manual effort and expert knowledge about side-channel attacks (all application developers cannot be expected to be security experts). On the other hand, approaches that rely on automated compiler transformations are either probabilistic [13], i.e., making attacks harder but not impossible, or target only a specific type of side-channel attacks, like execution-time-based attack [12, 16].

Randomization. Recently, Seo et al. [45] proposed the SGX Shield framework that enables code randomization for SGX enclaves. While the primary goal of SGX Shield is to protect enclaves from exploitable software bugs, authors mention that randomization imposes additional burden to side channel attackers, and in particular it provides reasonable protection against page-fault side-channel attacks, as it forces an attacker to brute force 2^7 times in order to identify a single address value. However, this argumentation does not directly apply to our attack, because SGX Shield concentrates on randomization of code, but does not randomize data. Hence, SGX Shield cannot hide data access patterns leveraged in our attack. More generally, randomization of data segments is challenging due to dynamic data allocations, large data objects (e.g., tables) that need to be split up and randomized, and pointer arithmetic which is typically used to access parts of large data objects (e.g., base-pointer relative offsets are often used to access table entries).

Attack detection. Previous works [42, 11] suggested to use system-level monitoring of performance counters to detect cache performance anomalies as a signature of ongoing cache-based attacks. However, this method is not applicable in the context of the SGX adversary model, since an attacker has sufficient privileges to disable any monitoring at system level.

Recently, two interesting works, T-SGX [46] and Déjá Vu [10], proposed detection methods for side-channel attacks that are based on frequent interruption of the victim enclave. A prime example of such *privileged* attacks is the deterministic side channel based on page-faults [53]. Here the OS incurs page faults during enclave execution and learns the execution flow or data access patterns of the enclave from the requested pages. Both

systems, however, the attacker can read the L1 cache without causing an enclave exit.

works suggest using a hardware implementation of *transactional memory* in Intel processors called Intel Transactional Synchronization Extensions (TSX) to notify an enclave about a (page fault) exception without interference by the system software.

T-SGX [46] modifies the enclave code such, that any interruption is detected and execution is terminated. However, this approach requires, in order to be effective, that the enclave cannot be restarted after the attack attempt was detected. To achieve this T-SGX requires one-time tokens provided by an external party over a secure channel. If the adversary targets the cryptographic protocol used in the establishment of that secure channel, this condition cannot be enforced (the attacker can initiate and replay the protocol). Once the attacker has extracted a valid token he can misuse it to run the enclave arbitrarily often, i.e., extract information despite the self-termination of the enclave.

Déjá Vu [10] extends enclave programs with execution time checks in order to detect delays caused by interruption of the enclave. SGX does not provide a reliable, fine-grained time source to enclaves, therefore, Déjá Vu uses a counting thread as a timer. The timer thread guards itself from being interrupted through the use of TSX. While cache eviction might slow down the victim, leading to a detectable delay, the timer thread can be slowed down as well, without interrupting it. The authors acknowledge that the timer thread can run on a core with the CPU's lowest frequency setting while the victim runs on a core set to maximum frequency. This can lead to a discrepancy of factor two or more, while, based on our experiments, L1 cache eviction due to constant priming slows down the victim by only 27%.

Summary. To summarize, the existing defense mechanisms are either not directly applicable in the context of Intel SGX, or have various drawbacks such as prohibitive performance penalty, limited scalability and effectiveness, or have to be integrated by hardware manufacturers, which hinder their applicability in practice.

7 Related Work

In this section we review works related to side-channel attacks mounted against SGX enclaves. In the extended version of this paper [7] we additionally survey works on SGX applications and cache-based side-channel attacks targeting non-SGX platforms.

SGX side channels. Costan and Devadas [14] analyzed SGX architecture and hypothesized that side-channel attacks could be mounted against SGX enclaves, though they did not provide any concrete evidence. Xu et al. [53] demonstrated page-fault side-channel attacks on SGX, where an untrusted operating system exfiltrates secrets from protected applications by tracking memory accesses at the granularity of memory pages.

Lee et al. [33] use branch shadowing to infer the control flow of an enclave. Their approach requires the victim enclave to be interrupted at a high frequency, which enables effective detection methods [46, 10].

Parallel work on SGX cache attacks. Parallel to us, other works on SGX cache attacks have been published.

First, Schwarz et al. [44] study a scenario, where an unprivileged attacker process (hiding in an enclave) is spying on the L3-cache utilization of another process (or enclave). The main difference to our work is that their attack monitors L3 (cross core), while our attack works on L1 (same core). As a result, our attack techniques are completely different.

Second, CacheZoom [38] attacks an AES implementation through L1 cache by interrupting the victim, and thus increasing the temporal resolution of the attack. Enclave exits introduce noise in a subset of cache lines rendering them unobservable. Additionally, the interrupts make the attack easily detectable [46, 10].

Third, Götzfried et al. [19] also attack AES on L1. Similar to our attack, they run the victim uninterrupted to avoid disturbance due to enclave exits. However, their attack assumes synchronization (collaboration) between the victim and the attacker – an assumption which typically does not hold in practice. In particular, they assume that (1) the victim and attacker code run as a single process in two separate threads; (2) victim and attacker have a shared memory which they used to communicate and exchange data, e.g., the attacker provides cipher text which needs to be decrypted by the victim; (3) the victim synchronizes with the attacker by indicating to the attacker when the last round of AES decryption is performed. This allows the attacker to prime the cache immediately before the last decryption round is executed, and probe it directly after it has finished.

Compared to these parallel works, the main benefit of our attack is that it requires no interrupts or synchrony assumptions, which makes it harder to detect and easier to deploy in practice.

8 Conclusion

In this paper we demonstrate that cache attacks on SGX are indeed practical and pose a serious threat on the core security benefit of SGX. Our goal was to develop an attack that cannot be mitigated by the known countermeasures, and therefore we mount the attack on uninterrupted enclave execution. We developed a set of novel attack techniques and demonstrated our attack on RSA decryption and genome indexing. Effectively defending non-cryptographic, but privacy-sensitive enclaves from cache attacks remains an open problem.

Acknowledgements

This work was supported in part by the German Science Foundation (project P3, CRC 1119 CROSSING), the European Union's Horizon 2020 Research and Innovation Programme (grant agreement No. 643964 – SUPERCLOUD), the Intel Collaborative Research Institute for Secure Computing (ICRI-SC), the German Federal Ministry of Education and Research within CRISP, and Zurich Information Security and Privacy Center (ZISC).

References

[1] An overview of cache. `http://download.intel.com/design/intarch/papers/cache6.pdf`.

[2] O. Acıiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2010.

[3] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[4] K. Ballantyne, M. Goedbloed, R. Fang, et al. Mutability of Y-Chromosomal Microsatellites: Rates, Characteristics, Molecular Bases, and Forensic Implications. *The American Journal of Human Genetics*, 2010.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[6] S. Bhattacharya and D. Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on Intel platforms. In *Cryptographic Hardware and Embedded Systems*, 2015.

[7] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. Technical report, arXiv:1702.07521 [cs.CR], 2017. `https://arxiv.org/abs/1702.07521`.

[8] E. Brickell, G. Graunke, and J.-P. Seifert. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference 2006, session DEV-203*, 2006.

[9] J. Butler. *Fundamentals of Forensic DNA Typing*. Academic Press, 2009.

[10] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*, 2017.

[11] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 2016.

[12] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization*, 2012.

[13] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE Symposium on Security and Privacy*, 2009.

[14] V. Costan and S. Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. `https://eprint.iacr.org/2016/086.pdf`.

[15] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[16] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Network and Distributed Systems Security Symposium*, 2015.

[17] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 2012.

[18] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 1996.

[19] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, EuroSec, 2017.

[20] D. Gruss, C. Maurice, and K. Wagner. Flush+Flush: A stealthier last-level cache attack. *Computing Research Repository (CoRR)*, abs/1511.04594, 2015.

[21] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.

[22] N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. In *Advances in Cryptology - CRYPTO 2009*, 2009.

[23] Intel. Intel software guard extensions. Tutorial slides. `https://software.intel.com/sites/default/files/332680-002.pdf`. Reference Number: 332680-002, revision 1.1.

[24] Intel. Intel 64 & IA-32 AORM. Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2012.

[25] Intel. Intel Software Guard Extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014.

[26] Intel. Intel 64 and IA-32 architectures software developer's manual. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html`, 2016.

[27] Intel. Software Guard Extensions Developer Guide, 2016. `https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf`.

[28] Intel. Software Guard Extensions for Linux OS: Intel IPP Cryptography library, 2016. https://github.com/01org/linux-sgx/blob/master/external/crypto_px/sources/ippcp/src/pcpngrsamontstuff.c#L438.

[29] Intel. Software Guard Extensions for Linux OS: Intel IPP Cryptography library, 2016. https://github.com/01org/linux-sgx/blob/master/external/crypto_px/sources/ippcp/src/pcpngrsamontstuff.c#L336.

[30] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*, 2015.

[31] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 2008.

[32] R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2008.

[33] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium*, USENIX Security, 2017.

[34] M. Lexa and G. Valle. PRIMEX: Rapid identification of oligonucleotide matches in whole genomes. *Bioinformatics*, 2003. https://www.researchgate.net/publication/233734306_mex-099tar.

[35] F. Liu and R. B. Lee. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[36] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, 2015.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[38] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. Technical report, arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[39] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[40] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 2003.

[41] D. Page. Partitioned cache architecture as a side-channel defence mechanism. In *IACR Eprint archive*, 2005.

[42] M. Payer. HexPADS: a platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems*, 2016.

[43] C. Percival. Cache missing for fun and profit. In *BSDCon*, 2005.

[44] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2017.

[45] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Network and Distributed System Security Symposium*, 2017.

[46] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium*, 2017.

[47] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[48] S. Tople, H. Dang, P. Saxena, and E. C. Chang. PermuteRam: Optimizing oblivious computation for efficiency. http://www.comp.nus.edu.sg/~shruti90/papers/permuteram.pdf, 2015.

[49] L. Uhsadel, A. Georges, and I. Verbauwhede. Exploiting hardware performance counters. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008.

[50] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference*, 2006.

[51] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th Annual International Symposium on Computer Architecture*, 2007.

[52] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[53] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, 2015.

[54] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[55] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. Technical report, Cryptology ePrint Archive. Report 2016/224, 2016. https://eprint.iacr.org/2016/224.pdf.

[56] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[57] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.