MicroScope: Enabling Microarchitectural Replay Attacks

Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery,
Josep Torrellas, and Christopher W. Fletcher
University of Illinois at Urbana-Champaign
{skarlat2,myan8,gopired2,spraber2,torrella,cwfletch}@illinois.edu

ABSTRACT

The popularity of hardware-based Trusted Execution Environments (TEEs) has recently skyrocketed with the introduction of Intel's Software Guard Extensions (SGX). In SGX, the user process is protected from supervisor software, such as the operating system, through an isolated execution environment called an enclave. Despite the isolation guarantees provided by TEEs, numerous microarchitectural side channel attacks have been demonstrated that bypass their defense mechanisms. But, not all hope is lost for defenders: many modern fine-grain, high-resolution side channels—e.g., execution unit port contention—introduce large amounts of noise, complicating the adversary's task to reliably extract secrets.

In this work, we introduce *Microarchitectural Replay Attacks*, whereby an SGX adversary can denoise nearly arbitrary microarchitectural side channels in a *single run* of the victim, by causing the victim to repeatedly replay on a page faulting instruction. We design, implement, and demonstrate our ideas in a framework, called *MicroScope*, and use it to denoise notoriously noisy side channels. Our main result shows how MicroScope can denoise the execution unit port contention channel. Specifically, we show how MicroScope can reliably detect the presence or absence of as few as two divide instructions in a single logical run of the victim program. Such an attack could be used to detect subnormal input to *individual* floating-point instructions, or infer branch directions in an enclave despite today's countermeasures that flush the branch predictor at the enclave boundary. We also use MicroScope to single-step and denoise a cache-based attack on the OpenSSL implementation of AES. Finally, we discuss the broader implications of microarchitectural replay attacks—as well as discuss other mechanisms that can cause replays.

CCS CONCEPTS

• Security and privacy → Side-channel analysis and countermeasures; **Trusted computing**; **• Software and its engineering → Virtual memory**.

KEYWORDS

Security, Side-channel, Operating System, Virtual Memory

ACM Reference Format:
Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, and Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322228

1 INTRODUCTION

The past several years have seen a surge of interest in hardware-based Trusted Execution Environments (TEEs) and, in particular, the notion of *enclave programming* [27, 28, 53]. In enclave programming, embodied commercially in Intel's Software Guard Extensions (SGX) [14, 21, 27, 28], outsourced software is guaranteed virtual-memory isolation from supervisor software—i.e., the Operating System (OS), hypervisor, and firmware. This support reduces the trusted computing base to the processor and the sensitive outsourced application. Since SGX's announcement five years ago, there have been major efforts in the community to map programs to enclaves, and to SGX in particular (e.g., [4, 8, 38, 43, 48, 49, 52, 54, 56, 66]).

Despite its promise to improve security in malicious environments, however, SGX has recently been under a barrage of microarchitectural side channel attacks. Such attacks allow co-resident software-based attackers to learn a victim process' secrets by monitoring how that victim uses system and hardware resources—e.g., the cache [36, 45, 62–64] or branch predictor [1, 17], among other structures [5, 7, 7, 20, 39, 46]. Some recent work has shown how SGX's design actually exacerbates these attacks. In particular, since the supervisor-level SGX adversary controls victim scheduling and demand paging, it can exert precise control on the victim and its environment [9, 15, 40, 58, 60].

Yet, not all hope is lost. There is scant literature on how much secret information the adversary can exfiltrate if the victim application only runs *once*, or for that matter if the instructions forming the side channel only execute once, i.e., not in a loop. Even in the SGX setting, many modern, fine-grain side channels—e.g., 4K aliasing [39], cache banking [64], and execution unit usage [5, 7]—introduce significant noise, forcing the adversary to run the victim many (potentially hundreds of) times to reliably exfiltrate secrets. Even for less noisy channels, such as the cache, SGX adversaries still often need more than one trace to reliably extract secrets [40]. This is good news for defenders. It is reasonable to expect that many outsourced applications, e.g., filing tax returns or performing tasks in personalized medicine, will only be run once per input. Further, since SGX can defend against conventional replay attacks using a combination of secure channels, attestation, and non-volatile counters [37], users have assurance that applications meant to run once will only run once.

1.1 This Paper

Despite the assurances made in the previous paragraph, this paper introduces *Microarchitectural Replay Attacks*, which enable the SGX adversary to denoise (nearly) any microarchitectural side channel inside of an SGX enclave, even if the victim application is *only run once*. The key observation is that a fundamental aspect to SGX's design enables an adversary to replay (nearly) arbitrary victim code, without needing to restart the victim after each replay, thereby bypassing SGX's replay defense mechanisms.

At a high level, the attack works as follows. In SGX, the adversary manages demand paging. We refer to a load that will result in a page fault as a *replay handle*—e.g., one whose data page has the Present bit cleared. In the time between when the victim issues a replay handle and the page fault is triggered, i.e., after the page table walk concludes, the processor will have issued instructions that are younger than the replay handle in program order. Once the page fault is signaled, the adversary can opt to *keep the present bit cleared*. In that case, due to precise exception handling and in-order commit, the victim will resume execution at the replay handle and the process will repeat a potentially unbounded number of times.

The adversary can use this sequence of actions to denoise microarchitectural side channels by searching for replay handles that occur before sensitive instructions or sensitive sequences of instructions. Importantly, the SGX threat model gives the adversary sufficient control to carry out these tasks. For example, the adversary can arrange for a load to cause a page fault if it knows the load address, and can even control the page walk time by priming the cache with select page table entries. Each replay provides the adversary with a noisy sample. By replaying an appropriate number of times, the adversary can disambiguate the secret from the noise.

We design and implement *MicroScope*, a framework for conducting microarchitectural replay attacks, and demonstrate our attacks on real hardware.[1] Our main result is that MicroScope can be used to reliably reveal execution unit port contention, i.e., similar to the PortSmash covert channel [5], even if the victim is only run once. In particular, with SMT enabled, our attack can detect the presence or absence of *as few as two divide instructions* in the victim. With further tuning, we believe we will be able to reliably detect one divide instruction. Such an attack could be used to detect subnormal input to *individual floating-point* instructions [7], or infer branch directions in an enclave despite countermeasures to flush the branch predictor at the enclave boundary [12]. Beyond port contention, we also show how our attack can be used to single-step and perform zero-noise cache-based side channels in AES, allowing an adversary to construct a denoised trace given a single run of that application.

Contributions. This paper makes the following contributions.

(1) We introduce microarchitectural replay attacks, whereby an SGX adversary can denoise nearly arbitrary microarchitectual side channels by causing the victim to replay on a page-faulting instruction.
(2) We design and implement a kernel module called *MicroScope*, which can be used to perform microarchitectural replay attacks in an automated fashion, given attacker-specified replay handles.
(3) We demonstrate that MicroScope is able denoise notoriously noisy side channels. In particular, our attack is able to detect the presence or absence of two divide instructions. For completeness, we also show single-stepping and denoising cache-based attacks on AES.
(4) We discuss the broader implications of microarchitectural replay attacks, and discuss different attack vectors beyond denoising microarchitectural side channels with page faults.

The source code for the MicroScope framework is available at https://github.com/dskarlatos/MicroScope.

[1]The name MicroScope comes from the attack's ability to peer inside nearly any microarchitectural side channel.

2 BACKGROUND

2.1 Virtual Memory Management in x86

A conventional TLB organization is shown in Figure 1. Each entry contains a Valid bit, the Virtual Page Number (VPN), the Physical Page Number (PPN), a set of flags, and the Process Context ID (PCID). The latter is unique to each process. The flags stored in a TLB entry usually include the Read/Write permission bit, the User bit that defines the privilege level required to use the entry, and other bits. The TLB is indexed using a subset of the virtual address bits. A hit is declared when the VPN and the PCID match the values stored in a TLB entry. Intel processors often deploy separate instruction and data L1 TLBs and a unified L2 TLB.

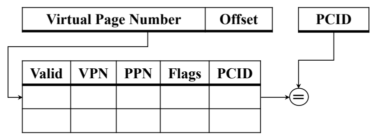


Figure 1: Conventional TLB organization.

If an access misses on both L1 and L2 TLBs, a page table walk is initiated to locate the missing translation. The hardware Memory Management Unit (MMU) performs this process. Figure 2 shows the page table walk for address *A*. The hardware first reads a physical address from the CR3 control register. This address corresponds to the process-private Page Global Directory (PGD). The page walker hardware adds the 40-bit CR3 register to bits 47-39 of the requested virtual address. The result is the physical address of the relevant pgd_t entry. Then, the page walker issues a request to the memory hierarchy to obtain the pgd_t. This memory request either hits in the data caches or is sent to main memory. The contents of pgd_t is the address of the next page table level, called Page Upper Directory (PUD). The same process is repeated for all the page table levels. Eventually, the page walker fetches the leaf pte_t entry that provides the PPN and flags. The hardware stores such informantion in the TLB.

Modern MMUs have a translation cache called the Page Walk Cache (PWC) that stores recent page table entries of the three upper levels. This can potentially reduce the number of memory accesses required to fetch a translation.

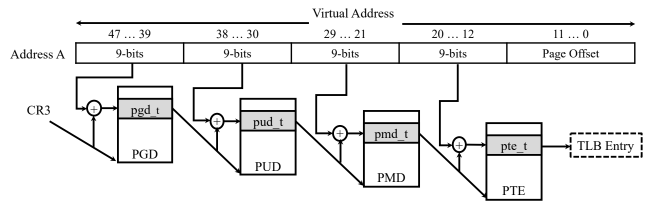


Figure 2: Page table walk.

A pte_t entry includes the present bit. If the bit is cleared, then the translation process fails and a page fault exception is raised. The OS is then invoked to handle it. After the OS services the page fault and updates the pte_t entry, control is yielded back to the process. Then, the memory request that caused the page fault is re-issued by the core. Once again, the request will miss in the TLB and initiate a page walk. At the end of the page walk, the updated pte_t will be stored in the TLB.

The OS is responsible for maintaining TLB coherence. This is done by flushing potentially-stale entries from the TLB. The INVLPG instruction [29] allows the OS to selectively flush a single TLB entry. When the OS needs to update a page table entry, it locates the leaf page table entry by performing a page walk following the same steps as the hardware page walker. Updating the page table causes the corresponding TLB entry to become stale. Consequently, the OS also invalidates the TLB entry before yielding control back to the process.

2.2 Out-of-Order Execution

Dynamically-scheduled processors execute instructions in parallel and out of program order to improve performance [55]. Instructions are fetched and enter the scheduling system in program order. However, they perform their operations and produce their results possibly out of program order. Finally, they retire—i.e., make their operation externally visible by irrevocably modifying the architected system state—in program order. In-order retirement is implemented by queueing instructions in program order in a reorder buffer (ROB) [30], and removing a completed instruction from the ROB only once it reaches the ROB head, i.e., after all prior instructions have retired.

Relevant to this paper, out-of-order machines continue execution during a TLB miss and page walk. When a TLB miss occurs, the access causing the miss queues a hardware page walk. The processor continues fetching and executing younger instructions, potentially filling up the ROB to capacity. If a page fault is detected, before it can be serviced, the page-faulting instruction has to reach the head of the ROB. Then, all the instructions younger than it are squashed. After the page fault is serviced, the program restarts at the page-faulting instruction.

2.3 Shielded Execution via Enclaves

Secure enclaves [53], such as Intel's Software Guard Extensions (SGX) [14, 27, 28], are reverse sandboxes that allow sensitive user-level code to run securely on a platform alongside an untrusted supervisor (i.e., an OS and/or hypervisor).

Relative to earlier TEEs such as Intel's TPM+TXT [26] and ARM TrustZone [6], a major appeal in enclave-based TEEs is that they are compatible with mostly unmodified legacy user-space software, and expose a similar process-OS interface to the supervisor as a normal user-level process. To run code in enclaves, the user writes enclave code and declares entry and exit points into that code, which may have arguments and return values. User-level code can jump into the enclave at one of the pre-defined entry points. This is similar to context switching into a new hardware context from the OS point of view. While the enclave code is running, the OS performs demand paging on behalf of the enclave context as if it were a normal process.

Enclave security is broken into attestation at bootup and privacy/integrity guarantees at runtime [53]. The runtime protections give enclave code access to a dedicated region of virtual memory which cannot be read or written except by that enclave code. Intel SGX implements these memory protections using virtual memory isolation for on-chip data and cryptographic mechanisms for off-chip data [21, 27]. For ease of use and information passing, SGX's design also allows enclave code to access user-level memory, owned by the host process, outside of the private enclave memory region.

For MicroScope to attack an enclave-based TEE, the only requirement is that the OS handles page faults during enclave execution, when trying to access either private enclave pages or insecure user-level pages. Intel SGX uses the OS for both of these cases. When a page fault occurs during enclave execution in SGX, the enclave signals an AEX (asynchronous exit), and the OS receives the VPN of the faulting page. To service the fault, the OS has complete control over the translation pages (PGD, PUD, etc.). If the faulting page is in the enclave's private memory region, additional checks are performed when the OS loads the page, e.g., to make sure it corresponds to the correct VPN [14]. MicroScope does not rely on the OS changing page mappings maliciously, and thus is not impacted by these defenses. If loading a new page requires displacing another page, the OS is responsible for TLB invalidations.

2.4 Side Channel Attacks

While enclave-based TEEs provide strong memory isolation mechanisms, they do not explicitly mitigate microarchitectural side channel attacks. Here, we review known side channel attacks that can apply to enclaves in Intel SGX. These attacks differ in their spatial granularity, temporal resolution, and noise level. We classify these attacks according to their capabilities in Table 1.

We classify an attack as providing fine-grain spatial granularity if the attack can be used to monitor victim access patterns at the granularity of cache lines or finer. We classify an attack as providing coarse-grain spatial granularity if it can only observe victim access patterns at coarser granularity, such as pages.

Coarse spatial granularity. Xu et al. [60] proposed controlled side channels to observe a victim's page-level access patterns by monitoring its page faults. Further, Wang et al. [58] proposed several new attack vectors, called Sneaky Page Monitoring (SPM). Instead of leveraging page faults to monitor the accesses that trigger many AEXs, SPM monitors the Access and Dirty bits in the page tables. Both attacks target page tables, and can only achieve page-level granularity, i.e., 4KB. In terms of noise, these attacks can construct noiseless channels, since the OS can manipulate the status of pages and can observe every page access.

Spatial	Coarse Grain	Fine Grain	
Temporal	–	Low Resolution	Medium/High Resolution
No Noise	Controlled side-channel [60] Sneaky Page Monitoring [58]		MicroScope (this work)
With Noise	TLBleed [20] TLB contention [25] DRAMA [46]	SGX Prime+Probe [18], Software Grand Exposure [9] Cache Bleed [64], MemJam [39], PortSmash [5] FPU subnormal attack [7], Execution unit contention [3, 59] BTB contention [1, 2], BTB collision [16], Leaky Cauldron [58]	Cache Games [22] CacheZoom [40] Hahnel et al. [23] SGX-Step [57]

Table 1: Characterization of side channel attacks on Intel SGX.

Gras et al. [20] and Hund et al. [25] proposed side channel attacks targeting TLB states. They create contention on the L1 DTLB and L2 TLB, which are shared across logical cores in an SMT core, to recover secret keys in cryptography algorithms and defeat ASLR. Similar to page table attacks, they can only achieve page-level granularity. Moreover, these two attacks suffer medium noise due to the races between attacker and victim TLB accesses. DRAMA [46] is another coarse-grain side channel attack that exploits DRAM row buffer reuse and contention. It can provide a granularity equal to the row buffer size (e.g., 2KB or 4KB).

Fine spatial granularity. There have been a number of works that exploit SGX to create fine spatial granularity side channel attacks that target the cache states or execution units (see Table 1). However, they all have sources of noise. Therefore, the victim must be run multiple times to obtain multiple traces, and intelligent post-processing techniques are required to minimize attack errors.

We further classify fine spatial granularity attacks according to the level of temporal resolution that they can achieve. We consider an attack to have high temporal resolution if it is able to monitor the execution of every single instruction. These attacks almost always require the attacker to have the ability to single-step the victim program. We define an attack to have low temporal resolution if it is only able to monitor the aggregated effects of multiple instructions.

Low temporal resolution. Several cache attacks on SGX [9, 18] use the Prime+Probe attack strategy and the PMU (performance monitoring unit) to observe a victim's access patterns at the cache line level. Leaky Cauldron [58] proposed combining cache attacks and DRAMA attacks to achieve fine-grain spatial granularity. These attacks cannot attain high resolution, since the attacker does not have a reliable way to synchronize with the victim, and the prime and probe steps generally take multiple hundreds of cycles. Moreover, these attacks suffer from high noise, due to cache pollution and coarse-grain PMU statistics. Generally, they require hundreds of traces to get modestly reliable results—e.g., 300 traces in the SGX Software Grand Exposure attack [9].

CacheBleed [64] and MemJam [39] can distinguish a victim's access patterns at even finer spatial granularity, i.e., sub-cache line granularity. Specifically, CacheBleed exploits L1 cache bank contention, while MemJam exploits false aliasing between load and store addresses from two threads in two different SMT contexts. However, in these attacks, the attacker analyzes the bank contention or load-store forwarding effects by measuring the total execution time of the victim. Thus, these attacks have low temporal resolution, as such information can only be used to analyze the accumulated effects of many data accesses. Moreover, such attacks are high noise, and require thousands of traces or thousands of events per trace.

There are several attacks that exploit contention on execution units [3, 5, 59], including through subnormal floating-point numbers [7], and collisions and contention on the BTB (branch target buffer) [1, 2, 16]. As they exploit contention in the system, they have similar challenges as CacheBleed. Even though these attacks can achieve fine spatial granularity, they have low temporal resolution and suffer from high noise.

Medium/high temporal resolution. Very few attacks can achieve both fine spatial granularity and high temporal resolution. Cache Games [22] exploits a vulnerability in the Completely Fair Scheduler (CFS) of Linux to slow victim execution, and achieve high temporal resolution. CacheZoom [40] and Hahnel et al. [23] and SGX-Step [57] use high-resolution timer interrupts to frequently stop the victim process, at the granularity of a few memory accesses, and collect L1 access information using Prime+Probe. Although these techniques encounter relatively low noise, they still require multiple runs of the application to denoise the exfiltrated information.

In summary, none of the prior works can simultaneously achieve fine spatial granularity, high temporal resolution, and no noise. We propose MicroScope to boost the effectiveness of almost all of the above attacks by de-noising them while, importantly, requiring only one run of the victim application. MicroScope is sufficiently general to be applicable to both cache attacks and contention-based attacks on various hardware components, such as execution units [7], cache banks [64], and load-store units [39].

3 THREAT MODEL

We adopt a standard threat model used when evaluating Intel SGX [9, 19, 24, 40, 43, 47, 48, 58, 65], namely, a victim program running within an SGX enclave alongside malicious supervisor software (i.e., the OS or a hypervisor). This gives the adversary complete control over the platform, except for the ability to directly introspect or tamper enclave private memory as described in Section 2.3. The adversary's goal is to break privacy, and learn as much about the secret enclave data as possible. For this purpose, the adversary may monitor any microarchitectural side channel (e.g., those in Section 2.4) while the enclave runs.

We restrict the adversary to run victim enclave code only one time per sensitive input. This follows the intent of many applications, such as tax filings and personalized health care. The victim can defend against the adversary replaying the entire enclave code

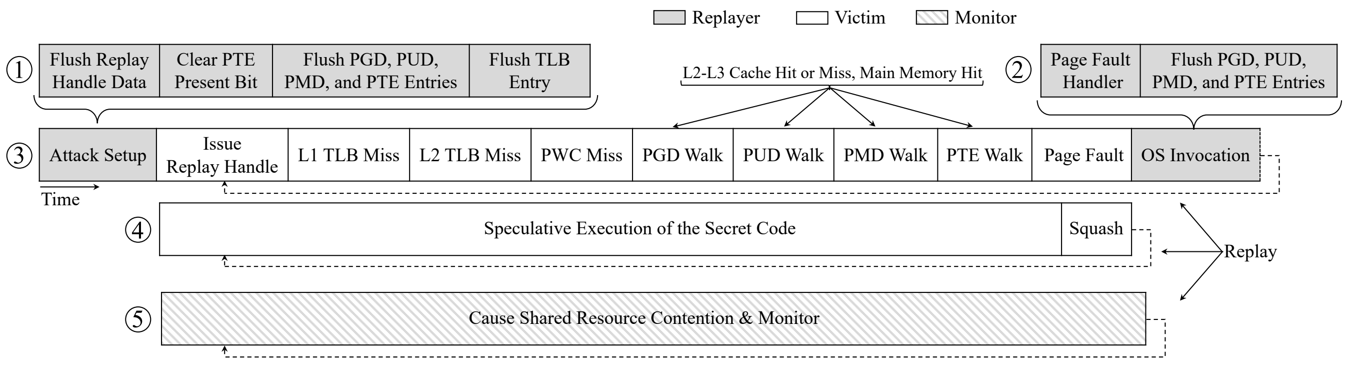


Figure 3: Timeline of a MicroScope attack. The *Replayer* is an untrusted OS or hypervisor process that forces the *Victim* code to replay, enabling the *Monitor* to denoise and extract the secret information.

by using a combination of secure channels and SGX attestation mechanisms, or through more advanced techniques [37].

Integrity of computation and physical side channels. Our main presentation is focused on breaking privacy over microarchitectural (digital) side channels. While we do not focus on program integrity, or physical attacks such as power/EM [34, 42], we discuss how microarchitectural replay attacks can be extended to these threat models in Section 7.

4 THE MICROSCOPE ATTACK

MicroScope is based on the key observation that modern hardware allows recently executed, but not retired, instructions to be rolled back and replayed if certain conditions are met. This behavior can be easily exploited by an untrusted OS to denoise side channels.

4.1 Overview

A MicroScope attack has three actors: Replayer, Victim, and Monitor. The *Replayer* is a malicious OS or hypervisor that is responsible for page table management. The *Victim* is an application process that executes some secret code that we wish to exfiltrate. The *Monitor* is a process that performs auxiliary operations, such as causing contention and monitoring shared resources.

4.1.1 Attack Setup: The Replay Handle. MicroScope is enabled by what we call a *Replay Handle*. A replay handle can be any memory access instruction that occurs shortly before a sensitive instruction in program order, and that satisfies two conditions. First, it accesses data from a different page than the sensitive instruction. Second, the sensitive instruction is not data dependent on the replay handle. Programs have many potential replay handles, including accesses to the program stack or heap, or memory access instructions that are unrelated to the sensitive instruction.

In MicroScope, the OS forces the replay handle to perform a page walk and incur a minor page fault. In the meantime, instructions that are younger than the replay handle, such as the sensitive instruction, can execute. More precisely, they can be inserted in the ROB and execute until the page fault is identified and the replay handle is at the head of the ROB, or until the ROB is full. Of course, instructions that are dependent on the replay handle do not execute.

Figure 3 shows the timeline of the interleaved execution of the Replayer, Victim, and Monitor. To initiate an attack, the adversary first identifies a replay handle close to the sensitive instruction. The adversary then needs to know the approximate time at which the replay handle will be executed, e.g., by single-stepping the Victim at page-fault [60] or close-to-instruction [40] granularity. The Replayer then pauses the Victim program before this point, and sets up the attack that triggers a page fault on the replay handle.

The Replayer sets up the attack by locating the page table entries required for virtual-to-physical translation of the replay handle—i.e., its pgd_t, pud_t, pmd_t, pte_t in Figure 2. The Replayer can easily do so by using the replay handle's virtual address. Then, the Replayer performs the following steps, shown in the timeline ① of Figure 3. First, it flushes from the caches the data to be accessed by the replay handle. This can be done by priming the caches. Second, it clears the present bit in the leaf page table entry (pte_t). Next, it flushes from the cache subsystem the four page table entries in the PDG, PUD, PMD, and PTE required for translation. Finally, it flushes the TLB entry that stores the {VPN, PPN} translation for the replay handle access. Together, these steps will cause the replay handle to miss in the TLB, and induce a hardware page walk to locate the translation, which will miss in the Page Walk Cache (PWC) and eventually result in a minor page fault.

Sometimes, it is also possible for the Replayer to use an instruction with a naturally occurring page fault as the replay handle.

4.1.2 Page Walk and Speculative Execution. After the attack is set-up, the Replayer allows the Victim to resume execution and issue the replay handle. The operation is shown in timeline ③ of Figure 3. The replay handle access misses in the L1 TLB, L2 TLB, and PWC, and initiates a page walk. The hardware page walker fetches the necessary page table entries sequentially, starting from PGD, then PUD, PMD, and finally PTE. The Replayer can tune the duration of the page walk time to take from a few cycles to over one thousand cycles, by ensuring that the desired page table entries are either present or absent from the cache hierarchy (shown in the arrows above timeline ③ of Figure 3).

In the shadow of the page walk, due to speculative execution, the Victim continues executing subsequent instructions, which perform the secret code computation. Such speculative instructions

```
1 //public address
2 handle(pub_addr);
3 ...
4 transmit(secret);
5 ...
```

(a) Single secret.

```
1 for i in ...
2       handle(pub_addrA);
3       ...
4       transmit(secret[i]);
5       ...
6       pivot(pub_addrB);
7       ...
```

(b) Loop secret.

```
1 handle(pub_addrA);
2 if (secret)
3     transmit(pub_addrB)
4 else
5     transmit(pub_addrC)
```

(c) Control flow secret.

Figure 4: Simple examples of codes that present opportunities for microarchitectural replay attacks.

execute but will not retire. They leave some state in the cache subsystem and/or create contention for hardware structures in the core. When the hardware page walker locates the leaf PTE that contains the translation, it finds that the present bit is clear. This finding eventually causes the hardware to raise a page fault exception and squash all of the speculative state in the pipeline.

The Replayer is then invoked to execute the page fault handler and handle the page fault. The Replayer could now set the present bit and allow the Victim to make forward progress. Alternatively, as shown in timeline ② of Figure 3, MicroScope's Replayer keeps the present bit clear and re-flushes the PGD, PUD, PMD, and PTE page table entries from the cache subsystem. As a result, as the Victim resumes and re-issues the replay handle, the whole process repeats. Timeline ④ of Figure 3 shows the actions of the Victim. This process can be repeated as many times as desired to denoise and extract the secret information.

4.1.3 *Monitoring Execution.*

The *Monitor* is responsible for extracting the secret information of the *Victim*. Depending on the Victim application and the side channel being exploited, we distinguish two configurations. In the first one, shown in timeline ⑤ of Figure 3, the Monitor executes in parallel with the Victim's speculative execution. The Monitor can cause contention on shared hardware resources and monitor the behavior of the hardware. For example, an attack that monitors contention in the execution units uses this configuration.

In the second configuration, the Monitor is part of the Replayer. After the Victim has yielded control back to the Replayer, the latter inspects the result of the speculative execution, such as the state of specific cache sets. A cache-based side channel attack could use this configuration.

4.1.4 *Summary of a MicroScope Attack.*

The attack consists of the following steps:

(1) The Replayer identifies the replay handle and prepares the attack.
(2) When the Victim executes the replay handle, it suffers a TLB miss followed by a page walk. The time taken by this step can be over one thousand cycles. It can be tuned as per the requirements of the attack.
(3) In the shadow of the page walk and until the page fault is serviced, the Victim continues executing speculatively past the replay handle into the sensitive region, potentially until the ROB is full.
(4) The Monitor can cause and measure contention on shared hardware resources during the Victim's speculative execution, or inspect the hardware state at the end of the Victim's speculative execution.
(5) When the Replayer gains control after the replay handle causes a page fault, it can optionally leave the present bit cleared in the PTE entry. This will induce another replay cycle that the Monitor can leverage to collect more information. Before the replay, the adversary may also prime the processor state for the next measurement. For example, if it uses a Prime+Probe cache-based attack, it can re-prime the cache.
(6) When sufficient traces have been gathered, the Replayer sets the present bit in the PTE entry. This enables the Victim to make forward progress.

With these steps, MicroScope can generate a large number of execution traces for one "logical" execution trace. It can denoise a side channel formed by, potentially, any instruction(s)—even ones that expose a secret only once in straight-line code.

4.2 Simple Attack Examples

Figure 4 shows several examples of codes that present opportunities for MicroScope attacks. Each example showcases a different use case.

4.2.1 *Single-Secret Attack.*

Figure 4a shows a simple code that has a single secret. Line 2 accesses a public address (i.e., known to the OS). This access is the replay handle. After a few other instructions, sensitive code at Line 4 processes some secret data. We call this computation the *transmit* computation of the Victim, using terminology from [32]. The transmit computation may leave some state in the cache or may use specific functional units that create observable contention. The goal of the adversary is to extract the secret information. The adversary can obtain it by using MicroScope to repeatedly perform steps (2)–(5) from Section 4.1.4.

To gain more insight, consider a more detailed example of the single-secret code (Figure 5). The figure shows function `getSecret` in C source code (Figure 5a) and in assembly (Figure 5b). In Figure 5a, we see that the function increments `count` and returns `secrets[id]/key`.

With MicroScope, the adversary can leverage the read to `count` as a replay handle. In Figure 5b, the replay handle is the `mov` instruction at Line 6. Then, MicroScope can be used to monitor the port

```
static uint64_t count;
static float secrets[512];

float getSecret(int id,
          float key){
  //replay handle
  count++;
  //measurement access
  return secrets[id]/key;
}
```

(a) Single-secret source.

```
1  _getSecret:
2  push   %rbp
3  mov    %rsp,%rbp
4  mov    %edi,-0x4(%rbp)
5  movss  %xmm0,-0x8(%rbp)
6  mov    0x200b27(%rip),%rax
7  add    $0x1,%rax
8  mov    %rax,0x200b1c(%rip)
9  mov    -0x4(%rbp),%eax
10 cltq
11 movss  0x601080(,%rax,4),%xmm0
12 divss  -0x8(%rbp),%xmm0
13 pop    %rbp
14 retq
```

(b) Single-secret assembly.

Figure 5: Single-secret detailed code.

contention in the floating-point division functional unit that executes `secrets[id]/key`. In Figure 5b, the division instruction is at Line 12. This is the transmit instruction. With this support, the adversary can determine whether `secrets[id]/key` is a subnormal floating-point operation, which has a different latency.

Alternatively, MicroScope can be used to monitor the cache access made by `secrets[id]`. In Figure 5b, the access `secrets[id]` is at Line 11. With MicroScope, the adversary can extract the cache line address of `secrets[id]`.

4.2.2 Loop-Secret Attack. We now consider the scenario where we want to monitor a given instruction in different iterations of a loop. We call this case `Loop Secret`, and show an example in Figure 4b. In the code, the loop body has a replay handle and a transmit operation. In each iteration, the transmit operation accesses a different secret. The adversary wants to obtain the secrets of all the iterations. The challenging case is when the address of the replay handle maps to the same physical data page in all the iterations.

This scenario highlights a common problem in side channel attacks: `secret[i]` and `secret[i+1]` may induce similar effects, making it hard to disambiguate between the two. For example, both secrets may co-locate in the same cache line, or induce similar pressure on the execution units. This fact severely impedes the ability to distinguish the two accesses.

MicroScope addresses this challenge through two capabilities. The first one, discussed in Section 4.1.2, is that the Replayer can dynamically tune the duration of the speculative execution, by controlling the page walk duration. In particular, the speculative execution window can be tuned to be short enough to allow the execution of only a single secret transmission per replay. This allows the Replayer to extract `secret[i]` without any noise.

The second capability is that the Replayer can use a *second* memory instruction to move between the replay handles in different iterations. This second instruction is located after the transmit instruction in program order, and we call it the *Pivot* instruction. For example, in Figure 4b, the instruction at Line 6 can act as the pivot. The only condition that the pivot has to satisfy is that its address should map to a different physical page than the replay handle.

MicroScope uses the pivot as follows. After the adversary infers `secret[i]` and is ready to proceed to extract `secret[i+1]`, the adversary performs one additional action during step 6 in Section 4.1.4. Specifically, after setting the present bit in the PTE entry for the replay handle, it clears the present bit in the PTE entry for the pivot, and resumes the Victim's execution. As a result, all the Victim instructions before the pivot are retired, and a new page fault is incurred for the pivot.

When the Replayer is invoked to handle the pivot's page fault, it sets the present bit for the pivot and clears the present bit for the replay handle. This is possible because we choose the pivot from a different page than the replay handle. When the Victim resumes execution, it retires all the instructions of the current iteration and proceeds to the next iteration, suffering a page fault in the replay handle. Steps 2- 5 repeat again, enabling the monitoring of `secret[i+1]`. The process is repeated for all the iterations.

As a special case of this attack scenario, when the transmit instruction (Line 4) is itself a memory instruction, MicroScope can simply use the transmit instruction as the pivot. This eliminates the need for a third instruction to act as pivot.

4.2.3 Control Flow Secret Attack. A final scenario that is commonly exploited using side channels is a secret-dependent branch condition. We call this case `Control Flow Secret`, and show an example in Figure 4c. In the code, the direction of the branch is determined by a secret, which the adversary wants to extract.

As shown in the figure, the adversary uses a replay handle before the branch, and a transmit operation in both paths out of the branch. The adversary can extract the direction taken by the branch using at least two different types of side channels. First, if Lines 3 and 5 in Figure 4c access different cache lines, then the Monitor can perform a cache based side-channel attack to identify the cache line accessed, and deduce the branch direction.

A second case is when the two paths out of the branch access the same addresses but perform different computations—e.g., one path performs a multiplication and the other performs a division. In this case, the transmit instructions are instructions that use the functional units. The Monitor can apply pressure on the functional units and, by monitoring the contention observed, deduce the operation that the code performs.

Prediction. The above situation is slightly more complicated in the presence of control-flow prediction such as branch prediction. With a branch predictor, the branch direction will initially be a function of the predictor state, *not* the secret. If the secret does not match the prediction, execution will squash. In this case both sides of the branch will execute, complicating the adversary's measurement.

MicroScope deals with this situation using the following insight: If the branch predictor state is *public*, whether there is a misprediction (re-execution) leaks the secret value, i.e., reveals `secret==predictor state`. The adversary can measure whether there is a misprediction by monitoring side channels for both sides of the branch in different replays.

The branch predictor state will likely be public in our setting. For example, the adversary can prime the predictor to a known state as in [33]. Likewise, if the predictor is flushed at enclave entry [12] the very act of flushing it puts it into a known state.

4.3 Exploiting Port Contention

To show the capabilities of MicroScope, we implement two popular attacks: in this section, we perform a port contention attack similar to PortSmash [5] without noise; in the next section, we use a cache-based side channel to attack AES.

In a port contention attack, the attacker tries to infer a few arithmetic operations performed by the Victim. Typically, the Monitor executes different types of instructions on the same core as the Victim, to create contention on the functional units, and observes the resulting contention. These attacks can have very high resolution [5], since they can potentially leak secrets at instruction granularity—even if the Victim code is fully contained in a single instruction and data cache line. However, they suffer from high noise due to the difficulty of perfectly aligning the timing of the execution of Victim and Monitor instructions.

We build the attack using the `Control Flow Secret` code of Figure 4c. One side of the branch performs two integer multiplications, while the other side performs two floating-point divisions. Importantly, there is *no loop* in the code; each side of the branch simply performs the two operations. The assembly code for the two sides of the branch is shown in Figure 6a (multiplication) and 6b (division). For clarity, each code snippet also includes the replay handle instruction in Line 1. Such instruction is executed before the branch. We can see that, in Lines 13 and 15, one code performs two integer multiplications and the other two floating-point divisions.

```
1  addq   $0x1,0x20(%rbp)
2  ...
3  __victim_mul
4  mov    0x2014b1(%rip),%rax
5  mov    %rax,0x20(%rsp)
6  mov    0x201498(%rip),%rax
7  mov    %rax,0x28(%rsp)
8  mov    0x20(%rsp),%rsi
9  mov    0x28(%rsp),%rdi
10 mov    (%rsi),%rbx
11 mov    (%rdi),%rcx
12 mov    %rcx,%rax
13 mul    %rbx
14 mov    %rcx,%rax
15 mul    %rbx
```

(a) Multiplication side.

```
1  addq   $0x1,0x20 (%rbp)
2  ...
3  __victim_div
4  mov    0x201548(%rip),%rax
5  mov    %rax,0x10(%rsp)
6  mov    0x20153f(%rip),%rax
7  mov    %rax,0x18(%rsp)
8  mov    0x10(%rsp),%rax
9  mov    0x18(%rsp),%rbx
10 movsd  (%rax),%xmm0
11 movsd  (%rbx),%xmm1
12 movsd  %xmm1,%xmm2
13 divsd  %xmm0,%xmm2
14 movsd  %xmm1,%xmm3
15 divsd  %xmm0,%xmm3
```

(b) Division side.

Figure 6: Victim code executing two multiplications (a) or two divisions (b). Note that code is not in a loop.

The goal of the adversary is to extract the secret that decides the direction of the branch.[2] To do so, the Monitor executes the simple port contention monitor code of Figure 7a. The code is a loop where each iteration repeatedly invokes `unit_div_contention()`, which performs a single floating-point division operation. The code measures the time taken by these operations and stores the time in an array. Figure 7b shows the assembly code of `unit_div_contention()`. Line 11 performs the single division, which creates port contention in the division functional unit.

```
1 for (j = 0; j < buff; j++){
2   t1 = read_timer();
3   for (i = 0; i < cont; i++){
4   // cause contention
5     unit_div_contention();
6   }
7   t2 = read_timer();
8   buffer[j] = t2 - t1;
9 }
```

(a) Monitor source code.

```
1  __unit_div_contention
2  mov    0x2012f1(%rip),%rax
3  mov    %rax,-0xc0(%rbp)
4  mov    0x2012eb(%rip),%rax
5  mov    %rax,-0xb8(%rbp)
6  mov    -0xc0(%rbp),%rax
7  mov    -0xb8(%rbp),%rbx
8  movsd  (%rax),%xmm0
9  movsd  (%rbx),%xmm1
10 movsd  %xmm1,%xmm2
11 divsd  %xmm0,%xmm2
```

(b) Assembly code for the division operation.

Figure 7: Monitor code that creates and measures port contention in the division functional unit.

The attack begins with the Replayer causing a page fault at Line 1 of Figure 6. MicroScope forces the victim to keep replaying the code that follows, which is either 6a or 6b, depending on the value of the secret. On a different SMT context of the same physical core, the Monitor concurrently executes the code in Figure 7a. The Monitor's `divsd` instruction at Line 11 of Figure 7b may or may not experience contention depending on the execution path of the Victim. If the Victim takes the path with `mul` (Figure 6a), the Monitor does not experience any contention and executes fast. If it takes the path with `divsd` (Figure 6b), the Monitor experiences contention and executes slower. Based on the execution time of the Monitor code, MicroScope reliably identifies the operation executed by the Victim, and thus the secret, after a few replay iterations.

This attack can also be used to discover fine-grain properties about an instruction's execution. As indicated before, one example is whether an individual floating-point operation receives a subnormal input. Prior attacks in this space are very course-grained, and can only measure whole-program timing [7].

4.4 Attacking AES

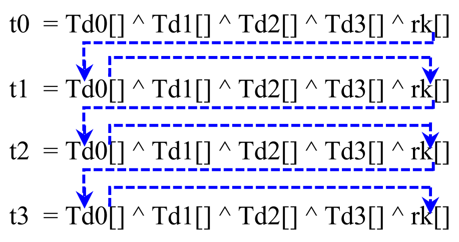
This section shows how MicroScope is used to attack AES decryption. We consider the AES decryption implementation from OpenSSL 0.9.8. [44]. For key sizes equal to 128, 192, and 256 bits, the algorithm performs 10, 12, and 14 rounds of computation, respectively. During each round, a set of pre-computed tables are used to generate the substitution and permutation values. Figure 8a shows the upper part of a computation round. For simplicity, we only focus on the upper part of a computation round; the lower part is similar. In the code shown in Figure 8a, each of the tables `Td0`, `Td1`, `Td2`, and `Td3` stores 256 unsigned integer values, and `rk` is an array of 60 unsigned integer elements. Our goal in this attack is to extract which entries of the `Td0-Td3` tables are read in each assignment in Figure 8a.

MicroScope attacks the AES decryption function using two main observations. First, the `Td0-Td3` tables and the `rk` array are stored in different physical pages. Therefore, MicroScope uses an access to

[2]We note that prior work demonstrated how to infer branch directions in SGX. Their approach, however, is no longer effective with today's countermeasures that flush branch predictor state at the enclave boundary [12].

```
1 for (;;) {
2   t0 = Td0[(s0 >> 24)] ^ Td1[(s3 >> 16) & 0xff] ^
3        Td2[(s2 >> 8) & 0xff] ^ Td3[(s1)&0xff] ^ rk[4];
4   t1 = Td0[(s1 >> 24)] ^ Td1[(s0 >> 16) & 0xff] ^
5        Td2[(s3 >> 8) & 0xff] ^ Td3[(s2)&0xff] ^ rk[5];
6   t2 = Td0[(s2 >> 24)] ^ Td1[(s1 >> 16) & 0xff] ^
7        Td2[(s0 >> 8) & 0xff] ^ Td3[(s3)&0xff] ^ rk[6];
8   t3 = Td0[(s3 >> 24)] ^ Td1[(s2 >> 16) & 0xff] ^
9        Td2[(s1 >> 8) & 0xff] ^ Td3[(s0)&0xff] ^ rk[7];
10
11  rk += 8;
12  if (--r == 0) {
13    break;
14  }
15  ...
16  ...
17 }
```

(a) AES decryption code from OpenSSL.


t0 = Td0[] ^ Td1[] ^ Td2[] ^ Td3[] ^ rk[]
t1 = Td0[] ^ Td1[] ^ Td2[] ^ Td3[] ^ rk[]
t2 = Td0[] ^ Td1[] ^ Td2[] ^ Td3[] ^ rk[]
t3 = Td0[] ^ Td1[] ^ Td2[] ^ Td3[] ^ rk[]

(b) MicroScope's replay handle and pivot path.

Figure 8: Using MicroScope to attack AES decryption.

rk as a replay handle, and an access to one of the Td tables as a pivot. This approach was described in Section 4.2.2. Second, the Replayer can fine-tune the page walk duration so that a replay covers only a small number of instructions. Hence, with such a small replay window, MicroScope can extract the desired information without noise. Overall, with these two strategies, we mount an attack where the adversary single steps the decryption function, extracting all the information without noise.

Specifically, the Replayer starts by utilizing the access to `rk[4]` in Line 3 of Figure 8a as the replay handle, and tunes the page walk duration so that the replay covers the instructions from Line 4 to Line 9. After each page fault is triggered, the Replayer acts as the Monitor, and accesses all the cache lines of all the Td tables. Based on the access times, after several replays, the Replayer can reliably deduce the lines accessed speculatively by the Victim. However, it does not know if a given line was accessed in the assignment to `t1`, `t2`, or `t3`.

After extracting this information, the Replayer sets the present bit for the `rk[4]` page, and clears the present bit for the `Td0` page. As explained in the `Loop Secret` attack of Section 4.2.2, `Td0` in Line 4 is a pivot. When the Victim resumes execution, the `rk[4]` access in Line 3 commits, and a page fault is triggered at the `Td0` access in Line 4. Now, the Replayer sets the present bit for the `Td0` page, and clears the present bit for the `rk[5]` page. As execution resumes, the Replayer now measures the accesses in Lines 6 to 9 of this iteration, and in Lines 2 to 3 of the next iteration. Hence, it can disambiguate any overlapping accesses from the instructions in Lines 4 to 5, since these instructions are no longer replayed.

The process repeats for the next lines and across loop iterations. Figure 8b shows a graphical representation of the path followed by the replay handle and pivot in one iteration. Note that, as described, this algorithm misses the accesses to tables `Td0-Td3` in Lines 2 and 3 for the first iteration. However, such values are obtained by using a replay handle before the loop.

Overall, with this approach, MicroScope reliably extracts all the cache accesses performed during the decryption. Importantly, MicroScope does it with only a single execution of AES decryption.

5 MICROSCOPE IMPLEMENTATION

In this section, we present the MicroScope framework that we implemented in the Linux kernel.

5.1 Attack Execution Path

Figure 9 shows a high-level view of the execution path of a MicroScope attack. The figure shows the user space, the kernel space, and the MicroScope module that we implemented in the kernel.

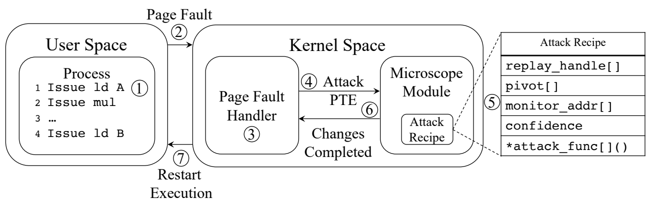


Figure 9: Execution path of a MicroScope attack.

Applications issue memory accesses using virtual addresses, which are later translated to physical ones (①). When the MMU identifies a page fault, it raises an exception and yields control to the OS to handle it (②). The page fault handler in the OS identifies what type of page fault it is, and calls the appropriate routine to service it (③). If it is a fault due to the present bit being clear, our modified page fault handler compares the faulting PTE entry to the ones currently marked as under attack. On a match, trampoline code redirects the page fault handler to the MicroScope module that we implemented (④). The MicroScope module may change the present bits of the PTE entries under attack (⑤), and prevents the OS page handler from interfering with them. After the MicroScope module completes its operations, the page fault handler is allowed to continue (⑥). Finally, control returns to the application (⑦).

5.2 MicroScope Module

The MicroScope module in the kernel uses a set of structures described below.

5.2.1 *Attack Recipes.*

The `Attack Recipes` is a structure in the MicroScope module that stores all the required information for specific microarchitectural replay attacks. For a given attack, it stores the replay handle, the pivot, and addresses to monitor for cache based side-channel attacks. It also includes a confidence threshold that is used to decide when the noise is low enough to stop the replays. Finally, each recipe defines a set of attack functions that are used to perform the attack.

This modular design allows an attacker to use a variety of approaches to perform an attack, and to dynamically change the attack recipe depending on the victim behavior. For example, if a side-channel attack is unsuccessful for a number of replays, the attacker can switch from a long page walk to a short one.

5.2.2 Attack Operations. MicroScope performs a number of operations to successfully carry-out microarchitectural replay attacks. The MicroScope module contains the code needed to execute such operations. The operations are as follows. First, MicroScope can identify the page table entries required for a virtual memory translation. This is achieved by performing a software page walk through the page table entries. Second, MicroScope can flush specific page table entries from the PWC and from the cache hierarchy. Third, it can invalidate TLB entries. Fourth, it can communicate through shared memory or signals with the Monitor that runs concurrently with the Victim; it sends stop and start signals to the Monitor when the Victim pauses and resumes, respectively, as well as other information based on the attack recipe. Finally, in cache based side-channel attacks, MicroScope can prime the cache system.

5.2.3 Interface to the User for Attack Exploration. To enable microarchitectural replay attack exploration, MicroScope provides an interface for the user to pass information to the MicroScope module. This interface enables the operations in Table 2. Some operations allow a user to provide a replay handle, a pivot, and addresses to monitor for cache based side-channel attacks. In addition, the user can force a specific address to initiate a page walk of *length* page-table levels, where length can vary from 1 to 4. Finally, the user can force a specific address to suffer a page fault.

Function	Operands	Semantics
provide_replay_handle	addr	Provide a replay handle
provide_pivot	addr	Provide a pivot
provide_monitor_addr	addr	Provide address to monitor
initiate_page_walk	addr, length	Initiate a walk of *length*
initiate_page_fault	addr	Initiate a page fault

Table 2: API used by a user process to access MicroScope.

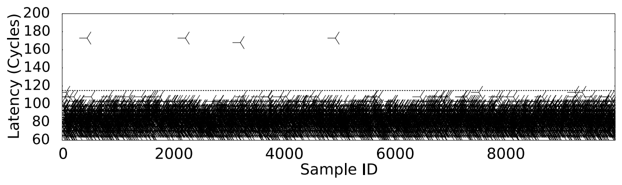
6 EVALUATION

We evaluate MicroScope on a Dell Precision Tower 5810 with an Intel Xeon E5-1630 v3 processor running Ubuntu with the 4.4 Linux kernel. We note that while our current prototype is not on an SGX-equipped machine, our attack framework uses an abstraction equivalent to the one defined by the SGX enclaves, as discussed in Section 3. Related work makes similar assumptions [60]. In this section, we evaluate two attacks: the port contention attack of Section 4.3, and the AES attack of Section 4.4.

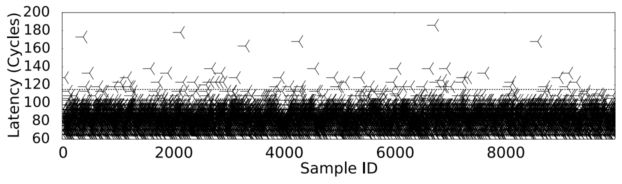
6.1 Port Contention Attack

In this attack, the Monitor performs the computation shown in Figure 7a. Concurrently, MicroScope forces the Victim to replay either the code in Figure 6a or the one in Figure 6b. The Monitor performs 10, 000 measurements. They measure a single logical run of the Victim, as the Victim code snippet is replayed many times.

Figure 10 shows the latency in cycles of each of the 10, 000 Monitor measurements while the Victim runs the code with the two multiplications (Figure 6a), or the one with the two divisions (Figure 6b). When the victim executes the code with the two multiplications, the latency measurements in Figure 10a show that all but 4 of the samples take less than 120 cycles. Hence, we set the contention threshold to slightly less than 120 cycles, as shown by the horizontal line.



(a) Victim executes two multiply operations as shown in Figure 6a.



(b) Victim executes two division operations as shown in Figure 6b.

Figure 10: Latencies measured by performing a port contention attack.

When the victim executes the code with the two divisions, the latency measurements in Figure 10b show that 64 measurements are above the threshold of 120 cycles. To understand this result, note that most Monitor samples are taken while the page fault handling code is running, rather than when the Victim code is running. The reason is that the page fault handling code executes for considerably longer than the Victim code in each replay iteration, and we use a simple free-running Monitor. For this reason, many measurements are below the threshold for both figures.

However, there is substantial difference between Figure 10b and Figure 10a. The former has 16x more samples over the threshold. This makes the two cases clearly distinguishable.

Overall, MicroScope is able to detect the presence or absence of two divide instructions, without any loop. It does so by denoising a notoriously noisy side channel through replay.

6.2 AES Attack

We use MicroScope to perform the cache side-channel attack on AES described in Section 4.4. We focus on one iteration of the loop in Figure 8a, and replay three times to obtain the addresses of the cache lines accessed by the Td0-Td3 tables. Each of these tables uses 16 cache lines.

Before the first replay (Replay 0), the Replayer does not prime the cache hierarchy. Hence, when the Replayer probes the cache after the replay, it finds the cache lines of the tables in different levels of the cache hierarchy. Before each of the next two replays (Replay 1 and Replay 2), the Replayer primes the cache hierarchy, evicting all the lines of the tables to main memory. Therefore, when the Replayer probes the cache after each replay, it finds the lines of the tables accessed by the Victim in the L1 cache, and the rest of the lines in main memory. As a result, the Replayer is able to identify the lines accessed by the victim.

Figure 11 shows the latency in cycles (Y axis) observed by the Replayer as it accesses each of the 16 lines of table Td1 (X axis) after each replay. We see that, after Replay 0, some lines have a latency lower than 60 cycles, others between 100 and 200 cycles, and one over 300. They correspond to hits in the L1, hits in L2/L3, and misses in L3, respectively. After Replay 1 and Replay 2, however, the picture is very clear and consistent. Only lines 4, 5, 7, and 9 hit in the L1, and all the other lines miss in L3. This experiment shows that MicroScope is able to extract the lines accessed in the AES tables without noise in a single logical run.

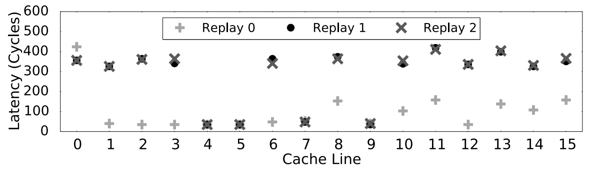


Figure 11: Latency of the accesses to the Td1 table after each of three replays of one iteration of AES.

7 GENERALIZING MICROARCHITECTURAL REPLAY ATTACKS

While this paper focused on a specific family of attacks (denoising microarchitectural side channels using page fault-inducing loads), the notion of replaying snippets in a program's execution is even more general and can be used to mount other privacy- or integrity-based attacks. Figure 12 gives a framework illustrating the different components in a microarchitectural replay attack. In our attack, the *replay handle* is a page fault-inducing load, the *replayed code* contains instructions that leak privacy over microarchitectural side channels, and the attacker's *strategy* is to unconditionally clear the page table present bit until it has high confidence that it has extracted the secret. We now discuss how to create different attacks by changing each of these components.

7.1 Attacks on Program Integrity

Our original goal with microarchitectural replay attacks was to bias non-deterministic instructions such as the Intel true random number generator RDRAND. Suppose the replayed code contains a RDRAND instruction. If the attacker learns the RDRAND return value over a side channel, its strategy is to *selectively* replay the Victim depending on the returned value (e.g., if it is odd, or satisfies some

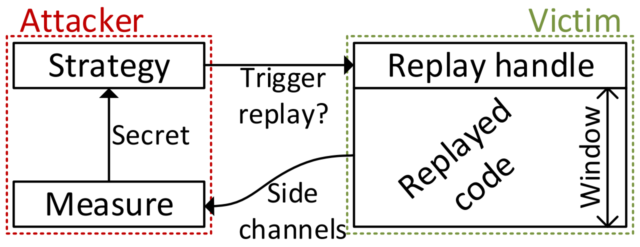


Figure 12: Generalized microarchitectural replay attacks.

other requirement). This is within the attacker's power: to selectively replay the Victim, the OS can access the last level page table (the PTE) directly and set/clear the present bit before the hardware page walker reaches it. The result is that the attacker can prevent the victim from obtaining certain values, effectively biasing the RDRAND from the Victim's perspective.

We managed to get all the components of such an attack to work correctly. However, it turns out that the current implementation of RDRAND on Intel platforms includes a form of fence. This fence prevents speculation after RDRAND, and the attack does not go through. In discussions with Intel, it appears that the reason for including this fence was not related to security. The lesson is that there should be such a fence, for security reasons.

More generally, the above discussion on integrity applies when there is any instruction in the replayed code that is non-deterministic. For example: RDRAND, RDSEED, RDTSC, or memory accesses to shared, writeable variables.

7.2 Attacks Using Different Replay Handles

While this paper uses page fault-inducing loads as replay handles, there are other instructions which can similarly cause a subsequent dynamic instruction to execute multiple times. For example, entering a transaction using transactional memory may cause code within the transaction to replay if the transaction aborts (e.g., if there are write set conflicts). Intel's Transactional Synchronization Extensions (TSX) in particular will abort a transaction if dirty data is evicted from the private cache, which can be easily controlled by an attacker. (We note that prior work has proposed using TSX in conjunction with SGX for defensive purposes [50].) One benefit of using TSX enter as a replay handle is that the window of replayed code is large, i.e., potentially the number of instructions in the transaction as opposed to the ROB size. This changes mitigations strategies. For example, fencing RDRAND (see above) will no longer be effective.

Other instructions can cause a limited number of replays. For example, any instruction which can squash speculative execution [11], e.g., a branch that mispredicts, can cause some subsequent code to be replayed. Since a branch will not mispredict an infinite number of times, the application will eventually make forward progress. However, the number of replays may still be large, e.g., if there are multiple branches in-flight, all of which mispredict. To maximize replays, the adversary can perform setup before the victim runs. For example, it can prime the branch predictor (similar to [33]) to mispredict if there are not already mechanisms to flush the predictors on context switches [12].

7.3 Amplifying Physical Side Channels

While our focus was to amplify microarchitecture side channels, microarchitectural replay attacks may also be an effective tool to amplify physical channels such as power and EM [34, 42]. For example, in the limit, the replayed code may be as little as a single instruction in the Victim, plus the attacker instructions needed to setup the next replay. Unsurprisingly, reducing Victim execution to fewer irrelevant instructions can improve physical attack effectiveness by denoising attack traces [41].

8 POSSIBLE COUNTERMEASURES

We now overview possible defense solutions and discuss their performance and security implications.

The root cause of microarchitectural replay attacks is that each dynamic instruction may execute more than one time. Based on the discussion in Section 7, this can be for a variety of reasons (e.g., a page fault, transaction abort, squash during speculative execution). Thus, it is clear that new, general security properties are required to comprehensively address these vulnerabilities. While we are working to design a comprehensive solution, we review some point mitigation strategies below that can be helpful to prevent specific attacks.

Fences on Pipeline Flushes. The obvious defense against attack variants, whose replayed code is contained within the ROB (see Section 7), is for the hardware or the OS to insert a fence after each pipeline flush. However, there are many corner cases that need to be considered. For example, it is possible that multiple instructions in a row induce a pipeline flush. This can be due to different causes, such as multiple page faults and/or branch mispredictions in close proximity. In these cases, even if a fence is introduced after every pipeline flush, the adversary can extract information from the resulting multiple replays.

Speculative Execution Defenses. MicroScope relies on speculative execution to replay Victim instructions. Therefore, a defense solution that holistically blocks side effects caused by speculative execution can effectively block MicroScope. However, existing defense solutions either have limited defense coverage or introduce substantial performance overhead. For example, using fences [29] or mechanisms such as InvisiSpec [61] or SafeSpec [31] only block specific covert channels such as the cache, and apply protections to all loads, which incurs large overhead. One idea to adapt these works to our attack is to only enable defenses while page faults are outstanding. Even with such an idea, however, these protections do not address side channels on the other shared processor resources, such as port contention [5]. Further, there may be a large gap in time between when an instruction executes and an older load misses in the TLB.

Page Fault Protection Schemes. As MicroScope relies on page faults to trigger replays, we consider whether page fault oriented defense mechanisms could be effective to defeat MicroScope. In particular, T-SGX [50] uses Intel's Transactional Synchronization Extensions (TSX) to capture page faults within an SGX application, and redirect their handling to a user-level code instead of the OS. The goal of T-SGX is to mitigate a controlled side-channel attack that leaks information through the sequence of page faults. However, T-SGX does not mitigate other types of side channels such as cache- or contention-based side channels.

The T-SGX authors are unable to distinguish between page faults and regular interrupts as the cause of transaction aborts. Hence, they use a threshold $N = 10$ of failed transactions to terminate the application. This design decision still provides $N - 1$ replays to MicroScope. Such number can be sufficient in many attacks.

Déjà Vu [13] is a technique that finds out whether a program is compromised by measuring with a clock if it takes an abnormal amount of time to execute. Déjà Vu uses TSX to protect the reference-clock thread. However, Déjà Vu presents two challenges. First, since the time to service an actual page fault is much longer than the time to perform a MicroScope replay, replays can be masked by ordinary application page faults. Second, to update state in Déjà Vu, the clock instructions need to retire. Thus, the attacker can potentially replay indefinitely on a replay handle, while concurrently preventing the clock instructions from retiring until the secret is extracted.

Both of the above defenses rely on Intel TSX. As discussed in Section 7, TSX itself creates a new mechanism with which to create replays, through transaction aborts. Thus, we believe further research is needed before applying either of the above defenses to any variant of microarchitectural replay attack.

Finally, Shinde et. al. [51] proposed a mitigation scheme to obfuscate page-granularity access patterns by providing page-fault obliviousness (or PF-obliviousness). The main idea is to change the program to have the same page access patterns for different input values, by inserting redundant memory accesses. Interestingly, this mechanism makes it easier for MicroScope to perform an attack, as the added memory accesses provide more replay handles.

9 RELATED WORK

We discuss several related works on exploiting speculative execution and improving side-channel attack accuracy.

Transient Execution Attacks. Starting with Meltdown [35] and Spectre [33], out-of-order speculative execution has created a new attack class known as transient execution attacks. These attacks rely on a victim executing code that it would not have executed at program level—e.g., instructions after a faulting load [35] or mispredicted branch [33]. The Foreshadow [10] attack is a Meltdown-style attack on SGX.

MicroScope is fundamentally different from transient execution attacks as it uses out-of-order speculative execution to create a replay engine. Replayed instructions may or may not be transient—e.g., instructions after a page faulting load may retire once the page fault is satisfied. Further, while Foreshadow exploits an implementation defect (L1TF), MicroScope exploits SGX's design, namely how the attacker is allowed to control demand paging.

Improving Side-Channel Attack Accuracy. As side channel attacks are generally noisy, there are several works that try to improve the temporal resolution and decrease the noise of cache attacks. Cache games [22] exploits the OS scheduler to slow down the victim execution and achieve higher attack resolution. CacheZoom [40] inserts frequent interrupts in the victim SGX application to stop the program every several data access instructions. SGX Grand Exposure [9] tries to minimize the noise during an attack by disabling interrupts, and uses performance monitoring counters to detect cache evictions. We provide more background on like attacks in Section 2.4.

All of the mechanisms mentioned can only improve the attack resolution in a limited manner. Also, they are helpful only for cache attacks. Compared to these approaches, MicroScope attains the highest temporal resolution with the minimum noise, since it replays the Victim execution in a fine-grained manner many times. In addition, MicroScope is the first framework that is general enough to be applicable to both cache attacks and other contention-based attacks on various hardware components [5, 39, 64].

10 CONCLUSION

Side-channel attacks are popular approaches to attack applications. However, many modern fine-grained side channels introduce too much noise to reliably leak secrets, even when the victim is run hundreds of times.

In this paper, we introduced Microarchitectural Replay Attacks targeting hardware-based Trusted Execution Environments such as Intel's SGX. We presented a framework, called MicroScope, which can denoise nearly arbitrary microarchitectural side channels in a *single run*, by causing the victim to replay on a page faulting instruction. We used MicroScope to denoise notoriously noisy side channels. In particular, our attack was able to detect the presence or absence of *two divide* instructions in a single run. Finally, we showed that MicroScope is able to single-step and denoise a cache-based attack on the AES implementation of OpenSSL.

ACKNOWLEDGMENTS

This work was funded in part by NSF under grants CCF-1725734 and CNS-1816226, and by an Intel Strategic Research Alliance (ISRA) grant. We greatly thank the anonymous reviewers for their feedback and insights during the review process.

REFERENCES

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, New York, NY, USA, 312–320.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys via Branch Prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'07)*. Springer-Verlag, Berlin, Heidelberg, 225–242.

[3] O. Acıiçmez and J. Seifert. 2007. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. 80–91. https://doi.org/10.1109/FDTC.2007.16

[4] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.*

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. Cryptology ePrint Archive, Report 2018/1060. https://eprint.iacr.org/2018/1060.

[6] T Alves and D Felton. 2004. TrustZone: Integrated Hardware and Software Security. *ARM white paper* (2004).

[7] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy.*

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR* abs/1702.07521 (2017). arXiv:1702.07521 http://arxiv.org/abs/1702.07521

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. [n.d.]. A Systematic Evaluation of Transient Execution Attacks and Defenses. CoRR'18.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085 http://arxiv.org/abs/1802.09085

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 7–18.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086.

[15] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 2 (May 2018), 171–191. https://tches.iacr.org/index.php/TCHES/article/view/879

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[17] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA.

[18] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[21] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204. https://eprint.iacr.org/2016/204.

[22] D. Gullasch, E. Bangerter, and S. Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy.*

[23] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 299–312.

[24] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[25] R. Hund, C. Willems, and T. Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy.* 191–205. https://doi.org/10.1109/SP.2013.23

[26] Intel. 2007. Intel Trusted Execution Technology. http://www.intel.com/technology/security.

[27] Intel. 2013. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/329298-001.pdf.

[28] Intel. 2013. Intel Software Guard Extensions Software Development Kit. https://software.intel.com/en-us/sgx-sdk.

[29] Intel. 2019. 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[30] Mike Johnson. 1991. *Superscalar microprocessor design*. Vol. 77. Prentice Hall Englewood Cliffs, New Jersey.

[31] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2018. SafeSpec:

Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *CoRR* abs/1806.05179 (2018).

[32] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).

[33] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). http://arxiv.org/abs/1801.01203

[34] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)*. Springer-Verlag, 388–397.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD.

[36] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1289–1306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[38] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*. 279–296.

[39] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2017. MemJam: A False Dependency Attack against Constant-Time Crypto Implementations. *CoRR* abs/1711.08002 (2017). arXiv:1711.08002 http://arxiv.org/abs/1711.08002

[40] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *CoRR* abs/1703.06986 (2017). arXiv:1703.06986 http://arxiv.org/abs/1703.06986

[41] Erick Nascimento, Lukasz Chmielewski, David Oswald, and Peter Schwabe. 2016. Attacking embedded ECC implementations through cmov side channels. Cryptology ePrint Archive, Report 2016/923. https://eprint.iacr.org/2016/923.

[42] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. 2017. EDDIE: EM-based detection of deviations in program execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*.

[43] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[44] OpenSSL. 2019. Open source cryptography and SSL/TLS toolkit. https://www.openssl.org.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, David Pointcheval (Ed.). Springer Berlin Heidelberg.

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[47] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[48] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.*

[49] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 18.

[50] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. Network and Distributed System Security Symposium 2017 (NDSS'17).

[51] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2897845.2897885

[52] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.*

[53] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA.

[54] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*. ACM, New York, NY, USA, 22–27. https://doi.org/10.1145/3268935.3268942

[55] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[56] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*. 9:1–9:14.

[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution (SysTEX'17)*. ACM.

[58] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA.

[59] Z. Wang and R. B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 473–482. https://doi.org/10.1109/ACSAC.2006.20

[60] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*.

[61] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).

[62] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA.

[63] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA.

[64] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* (2017).

[65] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. [n.d.]. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS'19*. https://eprint.iacr.org/2018/808.

[66] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA.