SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution

Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, Ten H. Lai
Department of Computer Science and Engineering
The Ohio State University
{*chen.4329, chen.4825, xiao.465*}*@osu.edu*
{*yinqian, zlin, lai*}*@cse.ohio-state.edu*

Abstract

This paper presents SGXPECTRE Attacks that exploit the recently disclosed CPU bugs to subvert the confidentiality of SGX enclaves. Particularly, we show that when branch prediction of the enclave code can be influenced by programs outside the enclave, the control flow of the enclave program can be temporarily altered to execute instructions that lead to observable cache-state changes. An adversary observing such changes can learn secrets inside the enclave memory or its internal registers, thus completely defeating the confidentiality guarantee offered by SGX. To demonstrate the practicality of our SGXPECTRE Attacks, we have systematically explored the possible attack vectors of branch target injection, approaches to win the race condition during enclave's speculative execution, and techniques to automatically search for code patterns required for launching the attacks. Our study suggests that *any* enclave program could be vulnerable to SGXPECTRE Attacks since the desired code patterns are available in most SGX runtimes (*e.g.*, Intel SGX SDK, Rust-SGX, and Graphene-SGX).

1 Introduction

Software Guard eXtensions (SGX) is a hardware extension available in recent Intel processors. It is designed to improve the application security by removing the privileged code from the trusted computing base (TCB). At a high level, SGX provides software applications shielded execution environments, called *enclaves*, to run private code and operate sensitive data, where both the code and data are isolated from the rest of the software systems. Even privileged software such as the operating systems and hypervisors are not allowed to directly inspect or manipulate the memory inside the enclaves. Software applications adopting Intel SGX are partitioned into sensitive and non-sensitive components. The sensitive components run inside the SGX enclaves (hence called *enclave programs*) to harness the SGX protection, while non-sensitive components run outside the enclaves and interact with the system software.

Although SGX is still in its infancy, the promise of shielded execution has encouraged researchers and practitioners to develop various new applications to utilize these features (*e.g.*, [8, 49, 28, 57, 83, 67, 52, 66, 86]), and new software tools or frameworks (*e.g.*, [11, 10, 65, 30, 15, 41, 78, 70, 63, 59, 48]) to help developers adopt this emerging programming paradigm. Most recently, SGX has been adopted by commercial public clouds, such as Azure confidential computing [56], aiming to protect cloud data security even with compromised operating systems or hypervisors, or even "malicious insiders with administrative privilege".

In SGX, the CPU itself, as part of the TCB, plays a crucial role in the security promises. However, the recently disclosed CPU vulnerabilities due to the out-of-order and speculative execution [29] have raised many questions and concerns about the security of SGX. Particularly, the so-called Meltdown [43] and Spectre attacks [40] have demonstrated that an unprivileged application may exploit these vulnerabilities to extract memory content that is only accessible to privileged software. The developers have been wondering whether SGX will hold its original security promises after the disclosure of these hardware bugs [5]. It is therefore imperative to answer this important question and understand its implications to SGX.

As such, we set off our study with the goal of comprehensively understanding the security impact of these CPU vulnerabilities on SGX. Our study leads to the SGXPECTRE Attacks, a new breed of the Spectre attacks on SGX. At a high level, SGXPECTRE exploits the race condition between the injected, speculatively executed memory references, which lead to side-channel observable cache traces, and the latency of the branch resolution. We coin a new name for our SGX version of the Spectre attacks not only for the convenience of our dis-

cussion, but also to highlight the important differences between them, including the threat model, the attack vectors, the techniques to win the race conditions, and the consequences of the attacks. We will detail these differences in later sections.

SGXPECTRE Attacks are a new type of side-channel attacks against SGX enclaves. Although it has already been demonstrated that by observing execution traces of an enclave program left in the CPU caches [58, 13, 27, 25], branch target buffers [42], DRAM's row buffer contention [75], page-table entries [72, 75], and page-fault exception handlers [80, 62], a side-channel adversary with system privileges may *infer* sensitive data from the enclaves, these traditional side-channel attacks are only feasible if the enclave program already has secret-dependent memory access patterns.

The consequences of SGXPECTRE Attacks are far more concerning. We show that SGXPECTRE Attacks can completely compromise the confidentiality of SGX enclaves. In particular, because vulnerable code patterns exist in most SGX runtime libraries (*e.g.*, Intel SGX SDK, Rust-SGX, Graphene-SGX) and are difficult to be eliminated, the adversary could perform SGXPECTRE Attacks against *any* enclave programs. We demonstrate end-to-end attacks to show that the adversary could learn the content of the enclave memory, as well as its register values in such attacks.

Responsible disclosure. We have disclosed our study to the security team at Intel.

Contributions. This paper makes the following contributions.

- *Systematic studies of a timely issue.* We provide the first comprehensive exploration of the impacts of the recent micro-architectural vulnerabilities on the security of SGX.
- *New techniques to enable SGX attacks.* We develop several new techniques that enable attacks against any enclave programs, including symbolic execution of SDK runtime binaries for vulnerability detection and combination of various side-channel techniques for winning the race conditions.
- *Security implications for SGX.* Our study concludes that SGX processors with these hardware vulnerabilities are no longer trustworthy, urging the enclave developers to add vulnerability verification into their development.

Roadmap. Sec. 2 introduces key concepts of Intel processor micro-architectures to set the stage of our discussion. Sec. 3 discusses the threat model. Sec. 4 presents a systematic exploration of attack vectors in enclaves and techniques that enable practical attacks. Sec. 5 presents a symbolic execution tool for automatically searching instruction gadgets in enclave programs. Sec. 6 shows end-to-end SGXPECTRE Attacks against enclave runtimes that lead to a complete breach of enclave confidentiality. Sec. 7 discusses and evaluates countermeasures against the attacks. Sec. 8 discusses related work and Sec. 9 concludes the paper.

2 Background

2.1 Intel processor internals

Out-of-order execution. Modern processors implement deep pipelines, so that multiple instructions can be executed at the same time. Because instructions do not take equal time to complete, the order of the instructions' execution and their order in the program may differ. This form of out-of-order execution requires taking special care of instructions whose operands have inter-dependencies, as these instructions may access memory in orders constrained by the program logic. To handle the potential data hazards, instructions are retired in order, resolving any inaccuracy due to the out-of-order execution at the time of retirement.

Speculative execution. Speculative execution shares the same goal as out-of-order execution, but differs in that speculation is made to speed up the program's execution when the control flow or data dependency of the future execution is uncertain. One of the most important examples of speculative execution is branch prediction. When a conditional or indirect branch instruction is met, because checking the branch condition or resolving branch targets may take time, predictions are made, based on its history, to prefetch instructions first. If the prediction is true, speculatively executed instructions may retire; otherwise mis-predicted execution will be re-winded. The micro-architectural component that enables speculative execution is the branch prediction unit (BPU), which consists of several hardware components that help predict conditional branches, indirect jumps and calls, and function returns. For example, branch target buffers (BTB) are typically used to predict indirect jumps and calls, and return stack buffers (RSB) are used to predict near returns. These micro-architectural components, however, are shared between software running on different security domains (*e.g.*, user space vs. kernel space, enclave mode vs. non-enclave mode), thus leading to the security issues that we present in this paper.

Implicit caching. Implicit caching refers to the caching of memory elements, either data or instructions, that are not due to direct instruction fetching or data accessing. Implicit caching may be caused in modern processors by "aggressive prefetching, branch prediction, and TLB miss handling" [2]. For example, mis-predicted branches

will lead to the fetching and execution of instructions, as well as data memory reads or writes from these instructions, that are not intended by the program. Implicit caching is one of the root causes of the CPU vulnerabilities studied in this paper.

2.2 Intel SGX

Intel SGX is a hardware extension in recent Intel processors offering stronger application security by providing primitives such as memory isolation, memory encryption, sealed storage, and remote attestation. An important concept in SGX is the secure enclave. An enclave is an execution environment created and maintained by the processor so that only applications running in it have a dedicated memory region that is protected from all other software components. Both confidentiality and integrity of the memory inside enclaves are protected from the untrusted system software.

Entering and exiting enclaves. To enter the enclave mode, the software executes the `EENTER` leaf function by specifying the address of Thread Control Structure (TCS) inside the enclave. TCS holds the location of the first instruction to execute inside the enclave. Multiple TCSs can be defined to support multi-threading inside the same enclave. Registers used by the untrusted program may be preserved after `EENTER`. The enclave runtime needs to determine the proper control flow depending on the register values (*e.g.*, differentiating `ECall` from `ORet`).

Asynchronous Enclave eXit (AEX). When interrupts, exceptions, and VM exits happen during the enclave mode, the processor will securely save the execution state in the State Save Area (SSA) of the current enclave thread, and replace it with a synthetic state to prevent information leakage. After the interrupts or exceptions are handled, the execution will be returned (through `IRET`) from the kernel to an address external to enclaves, which is known as Asynchronous Exit Pointer (AEP). The `ERESUME` leaf function will be executed to transfer control back to the enclave by filling the `RIP` with the copy saved in the SSA.

2.3 Cache side channels

Cache side channels leverage the timing difference between cache hits and cache misses to infer the victim's memory access patterns. Typical examples of cache side-channel attacks are PRIME-PROBE and FLUSH-RELOAD attacks. In PRIME-PROBE attacks [54, 53, 84, 51, 7, 68, 46, 36], by pre-loading cache lines in a cache set, the adversary expects that her future memory accesses (to the same memory) will be served by the cache, unless evicted by the victim program. Therefore, cache misses will reveal the victim's cache usage of the target cache set. In FLUSH-RELOAD attacks [26, 82, 81, 12, 85, 9], the adversary shares some physical memory pages (*e.g.*, through dynamic shared libraries) with the victim. By issuing `clflush` on certain virtual address that are mapped to the shared pages, the adversary can flush the shared cache lines out of the entire cache hierarchy. Therefore, RELOADs of these cache lines will be slower because of cache misses, unless they have been loaded by the victim into the cache. In these ways, the victim's memory access patterns can be revealed to the adversary.

3 Threat Model

In this paper, we consider an adversary with the system privilege of the machine that runs on the processor with SGX support. Specifically, we assume the adversary has the following capabilities.

- *Complete OS Control:* We assume the adversary has complete control of the entire OS, including recompiling of the OS kernel and rebooting of the OS with arbitrary argument as needed.
- *Interacting with the targeted enclave:* We assume the adversary is able to launch the targeted enclave program with a software program under her control. This means the arguments of `ECall`s and return values of `OCall`s are both controlled by the adversary.
- *Launching and controlling another enclave:* we assume the adversary is able to run another enclave that she completely controls in the same process or another process. This implies that the enclave can poison any BTB entries used by the targeted enclave.

The goal of the attack is to learn the memory content inside the enclave. We assume the binary code of the targeted enclave program is already known to the adversary and does not change during the execution. Therefore, we assume that the adversary is primarily interested in learning the secret data inside the enclaves after the enclave has been initialized (*e.g.*, generating secrets from random values or downloading secrets from the enclave owners.)

4 SGXPECTRE Attacks

4.1 A simple example

The basic idea of an SGXPECTRE Attack is illustrated in Figure 1. There are 5 steps in an SGXPECTRE Attack:

Step ❶ is to poison the branch target buffer, such that when the enclave program executes a branch instruction at a specific address, the predicted branch target is the address of enclave instructions that may leak secrets. For

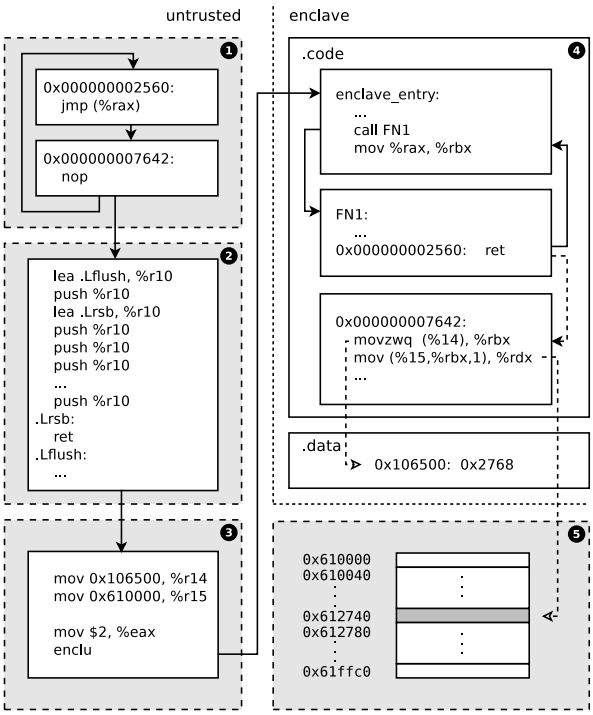

Figure 1: A simple example of SGXPECTRE Attacks. The gray blocks represent code or data outside the enclave. The white blocks represent enclave code or data.

example, in Fig. 1, to trick the `ret` instruction at address 0x02560 in the enclave to speculatively return to the secret-leaking instructions located at address 0x07642, the code to poison the branch prediction executes an indirect jump from the source address 0x02560 to the target address 0x07642 multiple times. We will discuss branch target injection in more details in Sec. 4.2.

Step ❷ is to prepare for a CPU environment to increase the chance of speculatively executing the secret-leaking instructions before the processor detects the misprediction and flushes the pipeline. Such preparation includes flushing the victim's branch target address (to delay the retirement of the targeted branch instruction or return) and depleting the RSB (to force the CPU to predict return address using the BTB). Flushing branch targets cannot use the `clflush` instruction, as the enclave memory is not accessible from outside (We will discuss alternative approaches in Sec. 4.5). The code for depleting the RSB (shown in Figure 1) pushes the address of a `ret` instructions 16 times and returns to itself repeatedly to drain all RSB entries.

Step ❸ is to set the values of registers used by the speculatively executed secret-leaking instructions, such that they will read enclave memory targeted by the adversary and leave cache traces that the adversary could monitor. In this simple example, the adversary sets `r14` to 0x106500, the address of a 2-byte secret inside the enclave, and sets `r15` to 0x610000, the base address of a monitored array outside the enclave. The `enclu` instruction with `rax=2` is executed to enter the enclave. We will discuss methods to pass values into the enclaves in Sec. 4.3.

Step ❹ is to actually run the enclave code. Because of the BTB poisoning, instructions at address 0x07642 will be executed speculatively when the target of the `ret` instruction at address 0x02560 is being resolved. The instruction "`movzwq (%r14), %rbx`" loads the 2-byte secret data into `rbx`, and "`mov (%r15, %rbx, 1), %rdx`" touches one entry of the monitored array dictated by the value of `rbx`.

Step ❺ is to examine the monitored array using a FLUSH-RELOAD side channel and extract the secret values. Techniques to do so are discussed in details in Sec. 4.4.

4.2 Injecting branch targets into enclaves

The branch prediction units in modern processors typically consists of:

- *Branch target buffer:* When an indirect jump/call or a conditional jump is executed, the target address will be cached in the BTB. The next time the same indirect jump/call is executed, the target address in the BTB will be fetched for speculative execution. Modern x86-64 architectures typically support 48-bit virtual address and 40-bit physical address [2, 37]. For space efficiency, many Intel processors, such as Skylake, uses the lower 32-bit of a virtual address as the index and tag of a BTB entry.
- *Return stack buffer:* When a near `Call` instruction with non-zero displacement[1] is executed, an entry with the address of the instruction sequentially following it will be created in the return stack buffer (RSB). The RSB is not affected by far `Call`, far `Ret`, or `Iret` instructions. Most processors that implement RSB have 16 entries [23]. On Intel Skylake or later processors, when RSB underflows, BTBs will be used instead.

Poisoning BTBs from outside. To temporarily alter the control-flow of the enclave code by injecting branch targets, the adversary needs to run BTB poisoning code outside the targeted enclave, which could be done in one of the following ways (as illustrated in Figure 2).

- *Branch target injection from the same process.* The adversary could poison the BTB by using code outside the enclave but in the same process. Since the

[1] `Call` instructions with zero displacement will not affect the RSB, because they are common code constructions for obtaining the current `RIP` value. These zero displacement calls do not have matching returns.

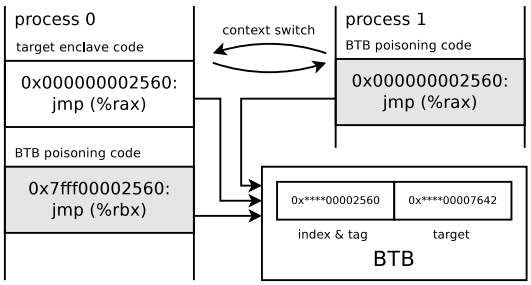

Figure 2: Poisoning BTB from the Same Process or A Different Process

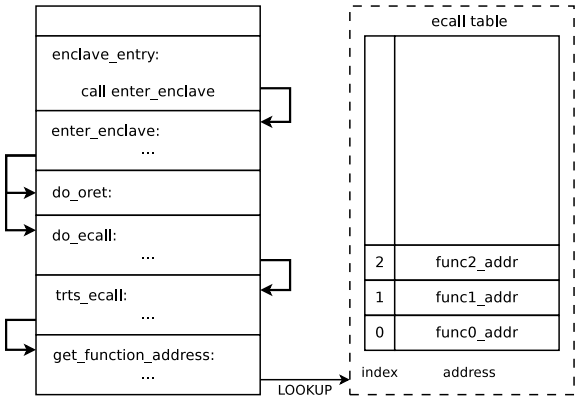

Figure 3: EENTER and ECall Table Lookup

BTB uses only the lower 32 bits of the source address as BTB indices and tags, the adversary could reserve a $2^{32} = 4$GB memory buffer, and execute an indirect jump instruction (within the buffer) whose source address (*i.e.*, 0x7fff00002560) is the same as the branch instruction in the target enclave (*i.e.*, 0x02560) in the lower 32 bits, and target address (*i.e.*, 0x7fff00007642) is the same as the secret-leaking instructions (*i.e.*, 0x07642) inside the target enclave in the lower 32 bits.

- *Branch target injection from a different process.* The adversary could inject the branch targets from a different process. Although this attack method requires a context switch in between of the execution of the BTB poisoning code and targeted enclave program, the advantage of this method is that the adversary could encapsulate the BTB poisoning coding into another enclave that is under his control. This allows the adversary to perfectly shadow the branch instructions of the targeted enclave program (*i.e.*, matching all bits in the virtual addresses).

It is worth noting that address space layout randomization can be disabled by adversary to facilitate the BTB poisoning attacks. On a Lenovo Thinkpad X1 Carbon (4th Gen) laptop with an Intel Core i5-6200U processor (Skylake), we have verified that for indirect jump/call, the BTB could be poisoned either from the same process, or a different process. For the return instructions, we only observed successful poisoning using a different process (*i.e.*, perfect branch target matching). The poisoning code for return instructions is the same as for indirect jumps/calls. To force return instructions to use BTB, the RSB needs to be depleted before executing the target enclave code. Interestingly, as shown in Fig. 1, a near call is made in `enclave_entry`, which could have filled the RSB, but we still could inject the return target of the return instruction at 0x02560 with BTB.

4.3 Controlling registers in enclaves

Because all registers are restored by hardware after `ERESUME`, the adversary is not able to control any registers inside the enclave when the control returns back to the enclave after an AEX. In contrast, most registers can be set before the `EENTER` leaf function and remain controlled by the adversary after entering the enclave mode until modified by the enclave code. Therefore, the adversary might have a chance to control some registers in the enclave after an `EENTER`.

The SGX developer guide [3] defines `ECall` and `OCall` to specify the interaction between the enclave and external software. An `ECall`, or "Enclave Call", is a function call to enter enclave mode; an `OCall`, or "Out Call", is a function call to exit the enclave mode. Returning from an `OCall` is called an `ORet`. Both `ECalls` and `ORets` are implemented through `EENTER` by the SGX SDK. As shown in Fig. 3, the function `enter_enclave` is called by the enclave entry point, `enclave_entry`. Then depending on the value of the `edi` register, `do_ecall` or `do_oret` will be called. The `do_ecall` function is triggered to call `trts_ecall` and `get_function_address` in a sequence and eventually look up the `Ecall` table. Both `Ecall` and `ORet` can be exploited to control registers in enclaves.

4.4 Leaking secrets via side channels

The key to the success of SGXPECTRE Attacks lies in the artifact that speculatively executed instructions trigger implicit caching, which is not properly rewinded when these incorrectly issued instructions are discarded by the processor. Therefore, these side effects of speculative execution on the CPU caches can be leveraged to leak information from inside the enclave.

Cache side-channel attacks against enclave programs have been studied recently [58, 13, 27, 25], all of which

demonstrated that a program runs outside the enclave may use PRIME-PROBE techniques [68] to extract secrets from the enclave code, only if the enclave code has secret-dependent memory access patterns. Though more fine-grained and less noisy, FLUSH-RELOAD techniques [82] cannot be used in SGX attacks because enclaves do not share memory with the external world.

Different from these studies, however, SGXPECTRE Attacks may leverage these less noisy FLUSH-RELOAD side channels to leak information. Because the enclave code can access data outside the enclave directly, an SGXPECTRE Attack may force the speculatively executed memory references inside enclaves to touch memory location outside the enclave, as shown in Figure 1. The adversary can flush an array of memory before the attack, such as the array from address 0x610000 to 0x61ffff, and then reload each entry and measure the reload time to determine if the entry has been touched by the enclave code during the speculative execution.

Other than cache side-channel attacks, previous work has demonstrated BTB side-channel attacks, TLB side-channel attacks, DRAM-cache side-channel attacks, and page-fault attacks against enclaves. In theory, some of these venues may also be leveraged by SGXPECTRE Attacks. For instance, although TLB entries used by the enclave code will be flushed when exiting the enclave mode, a PRIME-PROBE-based TLB attack may learn that a TLB entry has been created in a particular TLB set when the program runs in the enclave mode. Similarly, BTB and DRAM-cache side-channel attacks may also be exploitable in this scenario. However, page-fault side channels cannot be used in SGXPECTRE Attacks because the speculatively executed instructions will not raise exceptions.

4.5 Winning a race condition

At the core of an SGXPECTRE Attack is a race between the execution of the branch instruction and the speculative execution: data leakage will only happen when the branch instruction retires later than the speculative execution of the secret-leaking code. Fig. 4 shows a desired scenario for wining such a race condition in an SGXPECTRE Attack: The branch instruction has one data access `D1`, while the speculative execution of the secret-leaking code has one instruction fetch `I1` and two data accesses `D2` and `D3`. To win the race condition, the adversary should ensure that the memory accesses of `I1`, `D2` and `D3` are fast enough. However, because `I1` and `D2` fetch memory inside the enclave, and as TLBs and paging structures used inside the enclaves are flushed at AEX or `EEXIT`, the adversary could at best perform the address translation of the corresponding pages from caches (*i.e.*, use cached copies of the page table). Fortunately, it can be

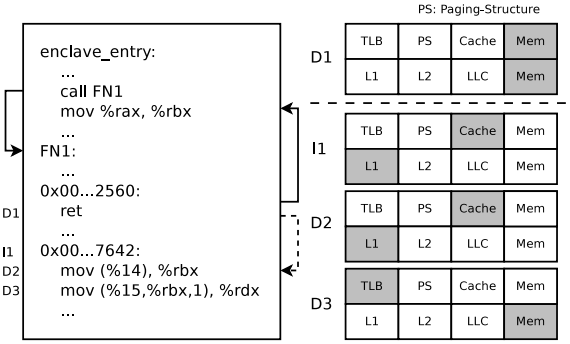

Figure 4: Best scenarios for wining a race condition. Memory accesses `D1`, `I1`, `D2`, `D3` are labeled next to the related instructions. The address translation and data accesses are illustrated on the right: The 4 blocks on top denote the units holding the address translation information, including TLBs, paging structures, caches (for PTEs), and the memory; the 4 blocks at the bottom denote the units holding data/instruction. The shadow blocks represent the units from which the address translation or data/instruction access are served.

achieved by performing the attack **Step ❹** in Fig. 1 multiple times. It is also possible to preload the instructions and data used in `I1` and `D2` into the L1 cache to further speed up the speculative execution. As `D3` accesses memory outside the enclave, it is possible to preload the TLB entry of the corresponding page. However, data of `D3` must be loaded from the memory.

Meanwhile, the adversary should slow down `D1` by forcing its address translation and data fetch to happen in the memory. However, this step has been proven technically challenging. First, it is difficult to effectively flush the branch target (and the address translation data) to memory without using `clflush` instruction. Second, because the return address is stored in the stack frames, which is very frequently used during the execution, evicting return addresses must be done frequently. In the attack described in Sec. 6, we leveraged an additional page fault to suspend the enclave execution right before the branch instruction and flush the return target by evicting all cache lines in the same cache set.

5 Attack Gadgets Identification

In this section, we show that any enclave programs developed with existing SGX SDKs are vulnerable to SGXPECTRE Attacks. In particular, we have developed an automated program analysis tool[2] that symbolically executes the enclave code to examine code patterns in the SGX runtimes, and have identified those code pat-

[2]The source code of our gadget scanning tool will be made public available when this manuscript is published.

terns in every runtime library we have examined, including Intel's SGX SDK [1], Graphene-SGX [15], Rust-SGX [20]. In this section, we present how we search these gadgets in greater detail.

5.1 Types of gadgets

In order to launch SGXPECTRE Attacks, two types of code patterns are needed. The first type of code patterns consists of a branch instruction that can be influenced by the adversary and several registers that are under the adversary's control when the branch instruction is executed. The second type of code patterns consists of two memory references sequentially close to each other and collectively reveals some enclave memory content through cache side channels. Borrowing the term used in return-oriented programming [60] and Spectre attacks [40], we use *gadgets* to refer to these patterns. More specifically, we name them *Type-I gadgets* and *Type-II gadgets*, respectively.

5.1.1 Type-I gadgets: branch target injection

Unlike the typical ROP gadget, we consider a gadget to be just a sequence of instructions that are executed sequentially during one run of the enclave program and they may not always be consecutive in the memory layout. A Type-I gadget is such an instruction sequence that starts from the entry point of `EENTER` (dubbed `enclave_entry`) and ends with one of the following instructions: (1) near indirect jump, (2) near indirect call, or (3) near return. `EENTER` is the only method for the adversary to take control of registers inside enclaves. During an `EENTER`, most registers are preserved by the hardware; they are left to be sanitized by the enclave software. If any of these registers are not overwritten by the software before one of the three types of branch instructions are met, a Type-I gadget is found.

An example of a Type-I gadget is shown in Listing 1, which is excerpted from `libsgx_trts.a` of Intel SGX SDK. In particular, line 44 in Listing 1 is the first return instruction encountered by an enclave program after `EENTER`. When this near return instruction is executed, several registers can still be controlled by the adversary, including `r8`, `r9`, `r10`, `r11`, `r14`, and `r15`.

Gadget exploitability. The exploitability of a Type-I gadget is determined by the number of registers that are controlled (both directly or indirectly) by the adversary at the time of the execution of the branch instruction. The more registers that are under control of the adversary, the higher the exploitability of the gadget. Highly exploitable Type-I gadgets mean less restriction on the Type-II gadgets in the exploits.

```
1  00000000000049a2 <enclave_entry>:
2  49a2: cmp    $0x0,%rax
3  49a6: jne    4a49 <enclave_entry+0xa7>
4  49ac: xor    %rdx,%rdx
5  49af: mov    %gs:0x8,%rax
6  49b6: 00 00
7  49b8: cmp    $0x0,%rax
8  49bc: jne    49cd <enclave_entry+0x2b>
9  49be: mov    %rbx,%rax
10 49c1: sub    $0x10000,%rax
11 49c7: sub    $0x2b0,%rax
12 49cd: xchg   %rax,%rsp
13 49cf: push   %rcx
14 49d0: push   %rbp
15 49d1: mov    %rsp,%rbp
16 49d4: sub    $0x30,%rsp
17 49d8: mov    %rax,-0x8(%rbp)
18 49dc: mov    %rdx,-0x18(%rbp)
19 49e0: mov    %rbx,-0x20(%rbp)
20 49e4: mov    %rsi,-0x28(%rbp)
21 49e8: mov    %rdi,-0x30(%rbp)
22 49ec: mov    %rdx,%rcx
23 49ef: mov    %rbx,%rdx
24 49f2: callq  3270 <enter_enclave>
25 ...
26
27 0000000000003270 <enter_enclave>:
28 3270: push   %r13
29 3272: push   %r12
30 3274: mov    %rsi,%r13
31 3277: push   %rbp
32 3278: push   %rbx
33 3279: mov    %rdx,%r12
34 327c: mov    %edi,%ebx
35 327e: mov    %ecx,%ebp
36 3280: sub    $0x8,%rsp
37 3284: callq  495b <get_enclave_state>
38 ...
39
40 000000000000495b <get_enclave_state>:
41 495b: lea    0x220566(%rip),%rcx
42 4962: xor    %rax,%rax
43 4965: mov    (%rcx),%eax
44 4967: retq
```

Listing 1: An Example of a Type-I Gadget

5.1.2 Type-II gadgets: secret extraction

A Type-II gadget is a sequence of instructions that starts from a memory reference instruction that loads data in the memory pointed to by register `regA` into register `regB`, and ends with another memory reference instruction whose target address is determined by the value of `regB`. When the control flow is redirected to a Type-II gadget, if `regA` is controlled by the adversary, the first memory reference instruction will load `regB` with the value of the enclave memory chosen by the adversary. Because the entire Type-II gadget is speculatively executed and eventually discarded when the branch instruction in the Type-I gadget retires, the secret value stored in `regB` will not be learned by the adversary directly. However, as the second memory reference will trigger the implicit caching, the adversary can use a FLUSH-RELOAD side channel to extract the value of `regB`.

An example of a Type-II gadget is illustrated in Listing 2, which is excerpted from the `libsgx_tstdc.a` library of Intel SGX SDK. Assuming `rdi` is a register con-

```
1 0000000000019460 <__b2d_D2A>:
2 ...
3 1947a: movslq 0x14(%rdi),%rax
4 1947e: lea    (%r10,%rax,4),%r9
5 19482: mov    -0x4(%r9),%r8d
6 ...
```

Listing 2: An Example of a Type-II Gadget

trolled by the adversary, the first instruction (line 3) reads the content of memory address pointed to by `rdi+0x14` to `rax`. Then the value of `r10+rax`$\times$`4` is stored in `r9` (line 4). Finally, the memory address at `r9-0x4` is loaded to `r8d` (line 5). To narrow down the range of `r9`, it is desired that `r10` is also controlled by the adversary. We use `regC` to represent these base registers like `r10`.

Gadget exploitability. The exploitability of a Type-II gadget is determined by two factors: First, whether there exists a register `regC` that serves as the base address of the second memory reference. Having such a register makes the attack much easier, because the range of the second memory references can be controlled by the adversary. Second, the number of instructions between the two memory references. Because speculative execution only lasts for a very short time, only a few instructions can be executed. The fewer instructions there are in the gadget, the higher its exploitability is.

5.2 Symbolically executing SGX code

Although a skillful attacker can manually read the source code or even the disassembled binary code of the enclave program, SGX SDKs, or the runtime libraries to identify usable gadgets for exploitation, such an effort is very tedious and error-prone. It is highly desirable to leverage automated software tools to scan an enclave binary to detect any exploitable gadgets, and eliminate the gadgets before deploying them to the untrusted SGX machines.

To this end, we devise a dynamic symbolic execution technique to enable automated identification of SGXPECTRE Attack gadgets. Symbolic execution [39] is a program testing and debugging technique in which symbolic inputs are supplied instead of concrete inputs. Symbolic execution abstractly executes a program and concurrently explores multiple execution paths. The abstract execution of each execution path is associated with a path constraint that represents multiple concrete runs of the same program that satisfy the path conditions. Using symbolic execution techniques, we can explore multiple execution paths in the enclave programs to find gadgets of SGXPECTRE Attacks.

More specifically, we leverage `angr` [64], a popular binary analysis framework to perform the symbolic execution. During the simulated execution of a program, machine states are maintained internally in `angr` to represent the status of registers, stacks, and the memory; instructions update the machine states represented with symbolic values while the execution makes forward progress. We leverage this symbolic execution feature of `angr` to enumerate execution paths and explore each machine state to identify the gadgets.

Symbolic execution of an enclave function. To avoid the path explosion problem during the symbolic execution of a large enclave program (or a large SGX runtime such as Graphene-SGX), we design a tool built atop the `angr` framework, which allows the user to specify an arbitrary enclave function to start the symbolic execution. The exploration of an execution path terminates when the execution returns to this entry function or detects a gadget. To symbolically execute an SGX enclave binary, we have extended `angr` to handle: (1) the `EEXIT` instruction, by putting the address of the enclave entry point, `enclave_entry`, in the `rip` register of its successor states; (2) dealing with instructions that are not already supported by `angr`, such as `xsave`, `xrstore`, `repz`, and `rdrand`.

5.3 Gadget identification

Identifying Type-I gadgets. The key requirement of a Type-I gadget is that before the execution of the indirect jump/call or near return instruction, the values of some registers are controlled (directly or indirectly) by the adversary, which can only be achieved via `EENTER`. We consider two types of Type-I gadget separately: `ECall` gadgets and `ORet` gadgets.

To detect `ECall` gadgets, the symbolic execution starts from the `enclave_entry` function and stops when a Type-I Gadget is found. During the path exploration, `edi` register is set to a value that leads to an `ECall`.

To detect `ORet` gadgets, the symbolic execution starts from a user-specified function inside the enclave. Once an `OCall` is encountered, the control flow is transfered to `enclave_entry` and the `edi` register is set to a value that leads to an `ORet`. At this point, all other registers are considered controlled by the adversary and thus are assigned symbolic values. An `ORet` gadget is found if an indirect jump/call or near return instruction is encountered and some of the registers still have symbolic values. The symbolic execution continues if no gadgets are found until the user-specified function finishes.

Identifying Type-II gadgets. To identify Type-II gadgets, our tool scans the entire enclave binary and looks for memory reference instructions (*i.e.*, `mov` and its variants, such as `movd` and `moveq`) that load register `regB` with data from the memory location pointed to by `regA`. Both `regA` and `regB` are general registers, such as `rax`,

	Category	End Address	Controlled Registers
Intel SGX SDK	indirect jump	<do_ecall>:0x118	rdi, r8, r9, r10, r11, r14, r15
	indirect call	—	—
	return	<do_ecall>:0x21	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<do_ecall>:0x63	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<_ZL16init_stack_guardPv>:0x21	rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<enter_enclave>:0x55	rbx, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<restore_xregs>:0x2b	r8, r9, r10, r11, r12, r14, r15
		<get_enclave_state>:0xc	rdx, rdi, r8, r9, r10, r11, r12, r14, r15
		<get_thread_data>:0x9	rbx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<__morestack>:0xe	r8, r9, r10, r11, r12, r13, r14, r15
		<asm_oret>:0x64	r8, r9, r10, r11, r12, r13, r14, r15
Graphene-SGX	indirect jump	—	—
	indirect call	<_DkGenericEventTrigger>:0x20	r9, r10, r11, r13, r14, r15
	return	<_DkGetExceptionHandler>:0x30	rdi, r8, r9, r10, r11, r12, r13, r14, r15
		<get_frame>:0x84	r8, r9, r10, r11, r12, r13, r14, r15
		<_DkHandleExternelEvent>:0x55	rdi, r8, r9, r10, r11, r12, r13, r14, r15
		<_DkSpinLock>:0x27	rbx, rdi, r8, r9, r10, r11, r12, r13, r14, r15
		<sgx_is_within_enclave>:0x23	rdi, rsi, r8, r12, r13, r14
		<handle_ecall>:0xcd	rdi, rsi, r8
		<handle_ecall>:0xd5	rdx, rdi, rsi, r8
Rust SGX SDK	indirect jump	<do_ecall>:0x118	rdi, r9, r10, r11, r12, r13, r14, r15
	indirect call	—	—
	return	<_ZL14do_init_threadPv>:0x109	rdi, r9, r10, r11, r12, r13, r14, r15
		<do_ecall>:0x21	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<do_ecall>:0x63	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<_ZL16init_stack_guardPv>:0x21	rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<_ZL16init_stack_guardPv>:0x69	rdi, r8, r9, r10, r11, r12, r13, r14, r15
		<enter_enclave>:0x55	rbx, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<restore_xregs>:0x2b	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<elf_tls_info>:0xa0	rbx, rdx, rsi, r9, r10, r11, r14, r15
		<get_enclave_state>:0xc	rdx, rdi, r8, r9, r10, r11, r12, r14, r15
		<get_thread_data>:0x9	rbx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<__morestack>:0xe	r8, r9, r10, r11
		<asm_oret>:0x64	r8, r9, r10, r11
		<__memcpy>:0xa3	rax, rbx, rdi, r9, r10, r11, r14, r15
		<__memset>:0x1d	rax, rbx, rdx, rdi, r9, r10, r11, r14, r15
		<__intel_cpu_features_init_body>:0x42b	rbx, rdx, rdi, r9, r10, r11, r14, r15

Table 1: SGXPECTRE Attack Type-I Gadgets in Popular SGX Runtime Libraries.

`rbx`, `rcx`, `rdx`, `r8` - `r15`. Once one of such instructions is found, the following N instructions (*e.g.*, $N = 10$) are examined to see if there exists another memory reference instruction (*e.g.*, `mov`, `cmp`, `add`) that accesses a memory location pointed to by register `regD`. If so, the instruction sequence is a potential Type-II gadget. It is desired to have a register `regC` used as the base address for the second memory reference. However, we also consider gadgets that do not involve `regC`, because they are also exploitable.

Once we have identified a potential gadget, it is executed symbolically using `angr`. The symbolic execution starts from the first instruction of a potential Type-II gadget, and `regB` and `regC` are both assigned symbolic values. At the end of the symbolic execution of the potential gadget, the tool checks whether `regD` contains a derivative value of `regB`, and when `regC` is used as the base address of the second memory reference, whether `regC` still holds its original symbolic values. The potential gadget is a true gadget if the checks pass. We use either [`regA`, `regB`, `regC`] or [`regA`, `regB`] to represent a Type-II gadget.

5.4 Case studies

We run our symbolic execution tool on three well-known SGX runtimes: the official Intel Linux SGX SDK (version 2.0.40950), Graphene-SGX (commit bf90323), and Rust-SGX SDK (version 0.9.1). To detect `ECall` Type-I Gadgets, the symbolic execution starts from the `enclave_entry` function in all three runtime libraries. To detect `ORet` Type-I gadgets, in Intel SGX SDK and Rust-SGX SDK, we started our analysis from the `sgx_ocall` function, which is the interface defined to serve all `OCalls`. In contrast, Graphene-SGX has more diverse `OCalls` sites. In total, there are 37 such sites as defined in `enclave_ocalls.c`. Unlike in other cases where the symbolic analysis completes instantly due to small function sizes, analyzing these 37 `OCalls` sites consumes more time: the median running time of analyzing one `OCalls` sites was 39 seconds; the minimum analysis time was 8 seconds; and the maximum was 340 seconds.

The results for Type-I gadgets are summarized in Table 1 and those for Type-II gadgets are listed in Table 2. More specifically, in Table 1, column 2 shows the type of the gadget, whether it being *indirect jump*, *indirect call*, or *return*; column 3 shows the address of

the branch instruction (basically the gadget's end address. Note that the Type-I gadget always starts at the `enclave_entry`.) represented using the function name the instruction is located and its offset; column 4 shows the registers that are under the control of the adversary when the branch instructions are executed. For example, the first entry in Table 1 shows an indirect jump gadget, which is located in `do_ecall` (with an offset of 0x118). By the time of the indirect jump, the registers that are still under the control of adversary are `rdi`, `r8`, `r9`, `r10`, `r11`, `r14` and `r15`.

Table 2 (in Appendix) lists Type-II gadgets of the form [`regA`, `regB`, `regC`], which means at the time of memory reference, two registers, `regB` and `regC`, are controlled by the adversary. This type of gadgets is easier to exploit. Column 2 shows the beginning address of the gadgets, represented using the function name and offset within the function; column 3 lists the entire gadgets. In most these examples, the number of instructions in the gadgets is less than 5; the shorter the gadgets are, the easier they can be exploited. The Type-II gadgets of the form [`regA`, `regB`] were not listed in the table, because they are too many. In total, we have identified 18, 86, and 180 such gadgets in these three runtimes, respectively.

6 Exploiting Intel SGX SDK

In this section, we demonstrate end-to-end SGXPECTRE Attacks against an arbitrary enclave program written with Intel SGX SDK [1], because this is Intel's official SDK. Rust-SGX was developed based on the official SDK and thus can be exploited in the same way. For demonstration purposes, the enclave program we developed has only one `ECall` function that runs in a busy loop. We verified that our own code does not contain any Type-I or Type-II gadgets in itself. The exploited gadgets, however, are located in the runtime libraries of SDK version 2.0.40950, which are listed in Listing 1 and Listing 2. Experiments were conducted on a Lenovo Thinkpad X1 Carbon (4th Gen) laptop with an Intel Core i5-6200U processor and 8GB memory.

6.1 Reading register values

We first demonstrate an attack that enable the adversary to read arbitrary register values inside the enclave. This attack is possible because during AEX, the values of registers are stored in the SSA before exiting the enclave. As the SSA is also a memory region inside the enclave, the adversary could leverage the SGXPECTRE Attacks to read the register values in the SSA during an AEX. This attack is especially powerful as it allows the adversary to frequently interrupt the enclave execution with AEX [71] and take snapshots of its SSAs to single-step trace its register values during its execution.

In particular, the attack is shown in Figure 5. In **Step** ①, the targeted enclave code is loaded into the enclave that is created by the program controlled by the adversary. After `EINIT`, the malicious program starts a new thread (denoted as the victim thread) to issue `EENTER` to execute the enclave code. Our enclave code only runs in a busy loop. But in reality, the enclave program might complete a remote attestation and establish trusted communication with its remote owner. In **Step** ②, the adversary triggers frequent interrupts to cause AEX from the targeted enclave. The processor stores the register values into the SSA during an AEX, and then exits the enclave and invokes the system software's interrupt handler. Before the control is returned to the enclave program via `ERESUME`, the adversary pauses the victim thread's execution at the AEP, a piece of instructions in the untrusted runtime library that takes control after `IRet`.

In **Step** ③, the main thread of the adversary-controlled program sets (through a kernel module) the reserved bit in the PTE of an enclave memory page that holds a global variable, `g_enclave_state`. As shown in Listing 1, this global variable is accessed right before the `ret` instruction of the Type-I gadget (*i.e.*, the memory referenced by `rcx` in the instruction "`mov (%rcx),%eax`". In **Step** ④, the main thread poison the BTB, deplete the RSB, prepare registers (*i.e.*, `rdi` and `r10`), and executes `EENTER` to trigger the attack. To poison the BTB, the adversary creates an auxiliary enclave program in another process containing an indirect jump with the source address equals the address of the return instruction in the Type-I gadget, and the target address the same as the start address of the Type-II gadget in the victim enclave. The process that runs in the auxiliary enclave is pinned onto the same logical core as the main thread. To trigger the BTB poisoning code, the main thread calls `sched_yield()` to relinquish the logical core to the auxiliary enclave program.

In **Step** ⑤, after the main thread issues `EENTER` to get into the enclave mode, the Type-I gadget will be executed immediately. Because a reserved bit in the PTE is set, a page fault is triggered when the enclave code accesses the `g_enclave_state` global variable. In the page fault handler, the adversary clears the reserved bit in the PTE, evicts the stack frame that holds the return address of the `ret` instruction from cache by accessing $2,000$ memory blocks whose virtual addresses have the same lower 12-bits as the stack address. In **Step** ⑥, due to the extended delay of reading the return address from memory, the processor speculatively executes the Type-II gadget (as a result of the BTB poisoning). After the processor flushes the speculatively executed instructions from the pipeline, the enclave code continues to execute.

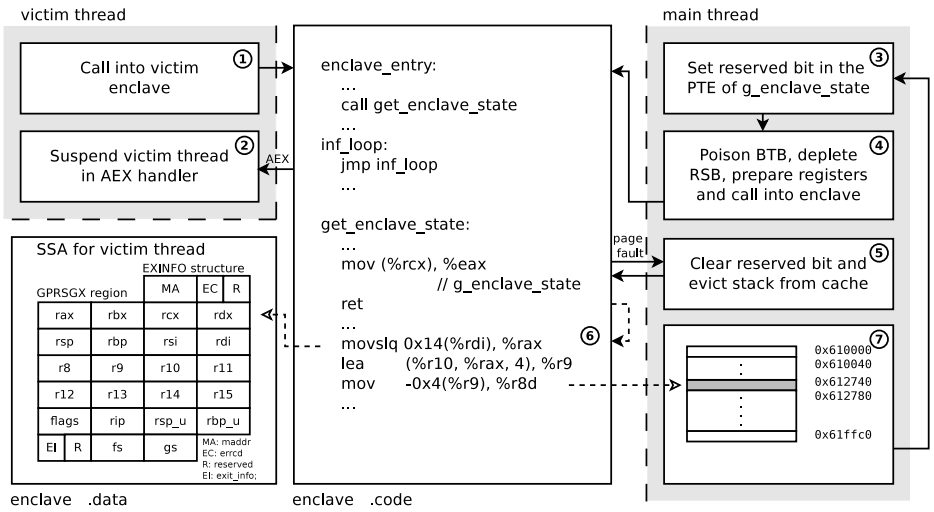

Figure 5: Exploiting Intel SGX SDK. The blocks with dark shadows represent instructions or data located in untrusted memory. Blocks without shadows are instructions inside the target enclave or the `.data` segment of the enclave memory.

However, because `rdi` points to the memory address of interest (*e.g.*, the address of SSA) inside the enclave, it is an invalid value for the SDK as `rdi` is used as the index of the `ecall_table`. The enclave execution will return with an error quickly after the speculative execution. This artifact allows the adversary to repeatedly probe into the enclaves. In **Step** ⑦, the adversary uses FLUSH-RELOAD techniques to infer the memory location accessed inside the Type-II gadget. One byte of SSA can thus be leaked. The main thread then repeats **Step** ③ to **Step** ⑦ to extract the remaining bytes of the SSA.

In our Type-I gadget, the `get_enclave_state` function only contains 4 instructions, which is very short. As executing the function will load the stack into the L1 cache, it is very difficult to flush the return address out of the cache to win the race condition. In fact, our initial attempts to flush the return address all failed. Triggering page faults to flush the return address resolves the issue. However, directly introducing page faults in every stack access could greatly increase the amount of time to carry out the attack. Therefore, instead of triggering page faults on the stack memory, the page fault is enforced on the global variable `g_enclave_state` which is located on another page. In this way, we can flush the return address with only one page fault in each run.

In our Type-II gadget, the first memory access reads 4 bytes (32 bits). It is unrealistic to monitor 2^{32} possible values in the FLUSH-RELOAD. However, if we know the value of lower 24 bits, we can adjust the base of the second memory access (*i.e.*, `r10`) to map the 256 possible values of the highest 8 bits to the cache lines monitored by the FLUSH-RELOAD code. Once all 32 bits of the targeted memory are learned, the adversary shifts the target address by one byte to learn the value of a new byte. We found in practice that it is not hard to find the initial consecutively known bytes. For example, the unused bytes in an enclave data page will be initialized as 0s, as they are used to calculate the measurement hash. Particularly, we found that there are 4 reserved bytes (in the EXINFO structure) in the SSA right before the GPRSGX region (which stores registers). Therefore, we can start from the reserved bytes (all 0s), and extract the GPRSGX region from the first byte to the last. As shown in Fig. 5, all register values, including `rax`, `rbx`, `rcx`, `rdx`, `r8` to `r15`, `rip`, *etc*, can be read from the SSA very accurately. To read all registers in the GPRSGX region (184 bytes in total), our current implementation takes 414 to 3677 seconds to finish. On average, each byte can be read in 6.6 seconds. We believe our code can be further improved.

6.2 Read other enclave memory

Reading other enclave memory follows exactly the same steps. The primary constraint is that the attack is much more convenient if three consecutive bytes are known. To read the `.data` segments, due to data alignment, some bytes are reserved and initialized as 0s, which can be used to bootstrap the attack. In addition, some global variables have limited data ranges, rendering most bytes known. To read the stack frames, the adversary could single-step trace the execution of the victim enclave (using SGXPECTRE Attacks) and learn the value of `rbp` from SSA to infer the the location of the return address. The return address in the stack frame should contain the address of the instruction that follows the `call` instruction that calls of the current function, which can be learned using simple program analysis. In this way, the adversary can start reading the stack frames from these known bytes.

7 Countermeasures

Hardware patches. To mitigate branch target injection attacks, Intel has released microcode updates to support the following three features [4].

- *Indirect Branch Restricted Speculation (IBRS):* IBRS restricts the speculation of indirect branches [6]. Software running in a more privileged mode can set an architectural model-specific register (MSR), `IA32_SPEC_CTRL.IBRS`, to 1 by using the `WRMSR` instruction, so that indirect branches will not be controlled by software that was executed in a less privileged mode or by a program running on the other logical core of the physical core. By default, on machines that support IBRS, branch prediction inside the SGX enclave cannot be controlled by software running in the non-enclave mode.
- *Single Thread Indirect Branch Predictors (STIBP):* STIBP prevents branch target injection from software running on the neighboring logical core, which can be enabled by setting `IA32_SPEC_CTRL.STIBP` to 1 by using the `WRMSR` instruction.
- *Indirect Branch Predictor Barrier (IBPB):* IBPB is an indirect branch control command that establishes a barrier to prevent the branch targets after the barrier from being controlled by software executed before the barrier. The barrier can be established by setting the `IA32_PRED_CMD.IBPB` MSR using the `WRMSR` instruction.

Particularly, IBPS provides a default mechanism that prevents branch target injection. To validate the claim, we developed the following tests: First, to check if the BTB is cleansed during `EENTER` or `EEXIT`, we developed a dummy enclave code that trains the BTB to predict address *A* for an indirect jump. After training the BTB, the enclave code uses `EEXIT` and a subsequent `EENTER` to switch the execute mode once and then executes the same indirect jump but with address *B* as the target. Without the IBRS patch, the later indirect jump will speculatively execute instructions in address *A*. However, with the hardware patch, instructions in address *A* will not be executed.

Second, to test if the BTB is cleansed during `ERESUME`, we developed another dummy enclave code that will always encounter an AEX (executing a memory access to a specific address that will trigger a page fault) right before an indirect call. In the AEP, another BTB poisoning enclave code will be executed before `ERESUME`. Without the patch, the indirect call speculatively executed the secret-leaking gadget. The attack failed after patching.

Third, to test the effectiveness of the hardware patch under Hyper-Threading, we tried poisoning the BTB using a program running on the logical core sharing the same physical core. The experiment setup was similar to our end-to-end case study in Sec. 6, but instead of pinning the BTB poisoning enclave code onto the same logical core, we pinned it onto the sibling logical core. We observed some secret bytes leaked before the patch, but no leakage after applying the patch.

Therefore, from these tests, we can conclude that SGX machines with microcode patch will cleanse the BTB during `EENTER`/`EEXIT` and during `ERESUME`, and also prevent branch injection via Hyper-Threading, thus they are immune to SGXPECTRE Attacks. Note that the CPU security version number (CPUSVN), which is used in generating keys for local and remote attestation, does reflects the processor's microcode update version, specifying the CPUSVN of the released microcode patch as the minimum CPUSVN in key generation processes could prevent SGXPECTRE Attacks.

Retpoline. Retpoline is a pure software-based solution to Spectre attacks [69], which has been developed for major compilers, such as GCC [79] and LLVM [14]. The name "retpoline" comes from "return" and "trampoline". Because modern processors have implemented separate predictors for function returns, such as Intel's return stack buffer [31, 32, 33, 34, 35] and AMD's return-address stack [37], it is believed that these return predictors are not vulnerable to Spectre attacks. Therefore, the key idea of retpoline is to replace indirect jump or indirect calls with returns to prevent branch target injection.

However, in recent Intel Skylake/Kabylake processors, on which SGX is supported, when the RSB is depleted, the BPU will fall back to generic BTBs to predict a function return. This allows poisoning of return instructions. Therefore, Retpoline is useless by itself in preventing SGXPECTRE Attacks.

Our recommendations. Due to the severity of SGXPECTRE Attacks, we urge the enclave authors to specify the minimum CPUSVN during their development. Moreover, we also suggest developers of runtime libraries (such as SGX SDKs) to scrutinize their code to remove exploitable gadgets in prevention of other potential ways of poisoning the BTB in the future. The symbolic execution tool presented in this paper can be used to look for these gadgets. Type-II gadgets can be removed by adding `lfense` in between of the two memory references. But the performance loss needs to be evaluated as [`regA`, `regB`] Type-II gadgets are very common in the runtimes. Type-I gadgets are harder to be eliminated, as it requires almost all registers to be sanitized after `EENTER` and before the control flows reach any indirect branch instructions or near returns.

8 Related Work

Meltdown and Spectre attacks. Our work is closely related to the recently demonstrated Spectre attacks [40,

29]. There are two variants of Spectre attacks: bounds check bypass and branch target injection. The first variant targets the conditional branch prediction and the second targets the indirect jump target prediction. A variety of attack scenarios have been demonstrated, including cross-process memory read [40], kernel memory read from user process, and host memory read from KVM guests [29]. However, their security implications on SGX enclaves have not been studied. In contrast, in this paper we have systematically investigated the enclave security on vulnerable SGX machines, devised new techniques to enable attacks against any enclave programs developed with Intel SGX SDK, and examined the effectiveness of various countermeasures.

Meltdown attacks [43] are another micro-architectural side-channel attacks that exploit implicit caching to extract secret memory content that is not directly readable by the attack code. Different from Spectre attacks, Meltdown attacks leverage the feature of out-of-order execution to execute instructions that should have not been executed. An example given by Lipp *et al.* [43] showed that an unprivileged user program could access an arbitrary kernel memory element and then visit a specific offset in an attacker-controlled data array, in accordance with the value of the kernel memory element, to load data into the cache. Because of the out-of-order execution, instructions after the illegal kernel memory access can be executed and then discarded when the kernel memory access instruction triggers an exception. However, due to implicit caching, the access to the attacker-controlled data array will leave traces in the cache, which will be captured by subsequent FLUSH-RELOAD measurements. Similar attacks can be performed to attack Xen hypervisor when the guest VM runs in paravirtualization mode [43]. However, we are not aware of any demonstrated Meltdown attacks against SGX enclaves.

Micro-architectural side channels in SGX. The SGXPECTRE Attacks are variants of micro-architectural side-channel attacks. Previously, various micro-architectural side-channel attacks have been demonstrated on SGX, which CPU cache attacks [58, 13, 27, 25], BTB attacks [42], page-table attacks [80, 62, 72], cache-DRAM attacks [75], *etc*. SGXPECTRE Attacks are different because they target memory content inside enclaves, while previous attacks aim to learn secret-dependent memory access patterns. However, SGXPECTRE Attacks leverage techniques used in these side-channel attacks to learn "side effects" of speculatively executed enclave code.

Side-channel defenses. Existing countermeasures to side-channel attacks can be categorized into three classes: hardware solutions, system solutions, and application solutions. Hardware solutions [76, 77, 21, 47, 45, 18] require modification of the processors, which are typically effective, but are limited in that the time window required to have major processor vendors to incorporate them in commercial hardware is very long. System solutions only modify system software [38, 73, 44, 87]. However, these solutions are not directly applicable to SGX enclaves as the system software is not trusted.

Application solutions are potentially applicable to SGX. Previous work generally falls into three categories: First, using compiler-assisted approaches to eliminate secret-dependent control flows and data flows [50, 17, 62], or to diversify or randomize memory access patterns at runtime to conceal the true execution traces [19, 55]. However, as the vulnerabilities in the enclave programs that enable SGXPECTRE Attacks are not caused by secret-dependent control or data flows, these approaches are not applicable. Second, using static analysis or symbolic execution to detect cache side-channel vulnerabilities in commodity software [22, 74]. However, these approaches model secret-dependent memory accesses in a program; they are not applicable in the detection of the gadgets used in our attacks. Third, detecting page-fault attacks or interrupt-based attacks against SGX enclave using Intel's hardware transactional memory [61, 16, 24]. These approaches can be used to detect frequent AEX. But they do not prevent SGXPECTRE Attacks.

9 Conclusion

We presented SGXPECTRE Attacks that extract secrets from the SGX enclaves. To demonstrate their practicality, we systematically explored the possible vectors of branch target injection, approaches to win the race condition during enclave's speculative execution, and techniques to automatically search for code patterns required for launching the attacks. We also demonstrated practical attacks against an arbitrary enclave program written with Intel SGX SDK, which not only extracts secrets in the enclave memory, but also the registers used only in the enclave mode.

Acknowledgments

The work was supported in part by the NSF grants 1564112, 1566444, and 1718084.

References

[1] Intel SGX SDK. `https://github.com/intel/linux-sgx`.

[2] Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2017. Order Number: 325462-065US, December 2017.

[3] Intel software guard extensions developer guide. `https://download.01.org/intel-sgx/linux-2.0/docs/`

Intel_SGX_Developer_Guide.pdf, 2017. Intel SGX Linux 2.0 Release.

[4] Intel analysis of speculative execution side channels, 2018. Revision 1.0, January 2018.

[5] Intel developer zone: Forums. https://software.intel.com/en-us/forum, 2018.

[6] Speculative execution side channel mitigations. http://kib.kiev.ua/x86docs/SDMs/336996-001.pdf, 2018. Revision 1.0, January 2018.

[7] ACIIÇMEZ, O. Yet another microarchitectural attack: exploiting I-Cache. In *2007 ACM workshop on Computer security architecture* (2007), pp. 11–18.

[8] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative technology for cpu based attestation and sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[9] APECECHEA, G. I., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! a fast, cross-vm attack on AES. In *Cryptology ePrint Archive* (2014).

[10] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. Scone: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), USENIX Association.

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

[12] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *Cryptology ePrint Archive* (2014).

[13] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies* (2017).

[14] CARRUTH, C. Retpoline patch for LLVM. https://reviews.llvm.org/D41723, 2018.

[15] CHE TSAI, C., PORTER, D. E., AND VIJ, M. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 645–658.

[16] CHEN, S., ZHANG, X., REITER, M., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with deja vu. In *12th ACM Symposium on Information, Computer and Communications Security* (2017).

[17] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy* (2009).

[18] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium* (2016), USENIX Association.

[19] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *ISOC Network and Distributed System Security Symposium* (2015).

[20] DING, Y., DUAN, R., LI, L., CHENG, Y., ZHANG, Y., CHEN, T., WEI, T., AND WANG, H. Poster: Rust sgx sdk: Towards memory safety in intel sgx enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2491–2493.

[21] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012).

[22] DOYCHEV, G., FELD, D., KÖPF, B., AND MAUBORGNE, L. CacheAudit: A tool for the static analysis of cache side channels. In *22st USENIX Security Symposium* (2013).

[23] FOG, A. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering* (2017).

[24] FU, Y., BAUMAN, E., QUINONEZ, R., AND LIN, Z. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17)* (Atlanta, Georgia. USA, September 2017).

[25] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on intel sgx. In *EUROSEC* (2017).

[26] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy* (2011), pp. 490–505.

[27] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference 17* (2017), USENIX Association.

[28] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[29] HORN, J. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[30] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[31] INTEL. Method and apparatus for implementing a speculative return stack buffer. US5964868, 1999.

[32] INTEL. Method and apparatus for predicting target addresses for return from subroutine instructions utilizing a return address cache. US Patent, Intel Corporation, US6170054, 2001.

[33] INTEL. Return address predictor that uses branch instructions to track a last valid return address. US Patent, Intel Corporation, US6253315, 2001.

[34] INTEL. System and method of maintaining and utilizing multiple return stack buffers. US Patent, Intel Corporation, US6374350, 2002.

[35] INTEL. Return register stack target predictor. US Patent, Intel Corporation, US6560696, 2003.

[36] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *36th IEEE Symposium on Security and Privacy* (May 2015).

[37] KELTCHER, C. N., MCGRATH, K. J., AHMED, A., AND CONWAY, P. The amd opteron processor for multiprocessor servers. *IEEE Micro 23*, 2 (March 2003), 66–76.

[38] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium* (2012).

[39] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[40] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (Jan. 2018).

[41] KUVAISKII, D., OLEKSENKO, O., ARNAUTOV, S., TRACH, B., BHATOTIA, P., FELBER, P., AND FETZER, C. Sgxbounds: Memory safety for shielded execution. In *12th European Conference on Computer Systems* (2017), ACM.

[42] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium* (2017), pp. 557–574.

[43] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (Jan. 2018).

[44] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *22nd IEEE Symposium on High Performance Computer Architecture* (2016).

[45] LIU, F., AND LEE, R. B. Random fill cache architecture. In *47th IEEE/ACM Symposium on Microarchitecture* (2014).

[46] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy* (May 2015).

[47] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture* (2012).

[48] MATETIC, S., KOSTIAINEN, K., DHAR, A., SOMMER, D., AHMED, M., GERVAIS, A., JUELS, A., AND CAPKUN, S. Rote: Rollback protection for trusted execution. Cryptology ePrint Archive, Report 2017/048, 2017. `http://eprint.iacr.org/2017/048.pdf`.

[49] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[50] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *8th international conference on Information Security and Cryptology* (2005).

[51] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on AES. In *13th international conference on Selected areas in cryptography* (2007), pp. 147–162.

[52] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multiparty machine learning on trusted processors. In *25th USENIX Security Symposium* (2016), USENIX Association.

[53] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *6th Cryptographers' track at the RSA conference on Topics in Cryptology* (2006), pp. 1–20.

[54] PERCIVAL, C. Cache missing for fun and profit. In *2005 BSDCan* (2005).

[55] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium* (2015).

[56] RUSSINOVICH, M. Introducing azure confidential computing, 2017. `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`.

[57] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy* (2015).

[58] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. Springer International Publishing, 2017.

[59] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. Sgx-shield: Enabling address space layout randomization for sgx programs. In *The Network and Distributed System Security Symposium* (2017).

[60] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security* (2007).

[61] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[62] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 317–328.

[63] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXENA, P. Panoply: Low-tcb linux applications with SGX enclaves. In *The Network and Distributed System Security Symposium* (2017).

[64] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy* (2016).

[65] STRACKX, R., AND PIESSENS, F. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium* (2016), USENIX Association.

[66] TAMRAKAR, S., LIU, J., PAVERD, A., EKBERG, J.-E., PINKAS, B., AND ASOKAN, N. The circle game: Scalable private membership test using trusted hardware. In *ACM on Asia Conference on Computer and Communications Security* (2017), ACM.

[67] TRAMER, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016. `https://eprint.iacr.org/2016/635`.

[68] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *J. Cryptol. 23*, 2 (Jan. 2010), 37–71.

[69] TURNER, P. Retpoline: a software construct for preventing branch-target-injection. `https://support.google.com/faqs/answer/7625886`, 2018.

[70] TYCHALAS, D., TSOUTSOS, N. G., AND MANIATAKOS, M. SGXCrypter: IP protection for portable executables using intel's SGX technology. In *22nd Asia and South Pacific Design Automation Conference* (2017).

[71] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution* (New York, NY, USA, 2017), SysTEX'17, ACM, pp. 4:1–4:6.

[72] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium* (2017), USENIX Association.

[73] VARADARAJAN, V., RISTENPART, T., AND SWIFT, M. Scheduler-based defenses against cross-VM side-channels. In *23th USENIX Security Symposium* (2014).

[74] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium* (Vancouver, BC, 2017), USENIX Association.

[75] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. 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* (2017).

[76] WANG, Z., AND LEE, R. B. Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference* (2006).

[77] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *34th annual international symposium on Computer architecture* (2007).

[78] WEISER, S., AND WERNER, M. Sgxio: Generic trusted i/o path for intel sgx. arXiv preprint, arXiv:1701.01061, 2017. `https://arxiv.org/abs/1701.01061`.

[79] WOODHOUSE, D. Retpoline patch for GCC. `http://git.infradead.org/users/dwmw2/gcc-retpoline.git`, 2018.

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

[81] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. In *Cryptology ePrint Archive* (2014).

[82] YAROM, Y., AND FALKNER, K. E. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium* (2014), pp. 719–732.

[83] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., , AND SHI, E. Town crier: An authenticated data feed for smart contracts. In *23rd ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM.

[84] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security* (2012).

[85] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security* (2014).

[86] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation* (2017), USENIX Association.

[87] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security* (2016).

10 Appendix

Due to space constraints, we list all [`regA`, `regB`, `regC`] Type-II gadgets of the three SGX runtimes, *e.g.*, Intel SGX SDK, Graphene-SGX, and Rust-SGX SDK, in Table 2. The numbers of [`regA`, `regB`] Type-II gadgets are too large to be included in the paper.

	Start Address	Gadget Instructions
Intel SGX SDK	<dispose_chunk>:0x8a	mov 0x38(%rsi),%r9d; mov %r9,%rcx; lea (%rdi,%r9,8),%r9; cmp 0x258(%r9),%rsi
	<dispose_chunk>:0x299	mov 0x38(%r8),%r9d; mov %r9,%rcx; lea (%rdi,%r9,8),%r9; cmp 0x258(%r9),%r8
	<dlmalloc>:0x180b	mov 0x38(%rdx),%r12d; mov %r12,%rcx; add $0x4a,%r12; cmp 0x8(%rsi,%r12,8),%rdx
	<dlfree>:0x399	mov 0x38(%r8),%edi; mov %rdi,%rcx; lea (%rbx,%rdi,8),%rdi; cmp 0x258(%rdi),%r8
	<dlfree>:0x46f	mov 0x38(%rsi),%edi; mov %rdi,%rcx; lea (%rbx,%rdi,8),%rdi; cmp 0x258(%rdi),%rsi
	<dlrealloc>:0x341	mov 0x38(%rsi),%r10d;mov %r10,%rcx;lea (%rbx,%r10,8),%r10;cmp %rsi,0x258(%r10)
	<__hdtoa>:0x170	movsbl (%rdx),%ecx; sub $0x1,%rdx; movzbl (%r12,%rcx,1),%ecx
	<__hldtoa>:0x158	movsbl (%rdx),%ecx; sub $0x1,%rdx; movzbl (%r12,%rcx,1),%ecx
	<__b2d_D2A>:0x1a	movslq 0x14(%rdi),%rax; lea (%r10,%rax,4),%r9; mov -0x4(%r9),%r8d
Graphene-SGX	<do_lookup_map>:0x97	mov 0x2f0(%r8),%rax; mov (%rax,%rdx,4),%eax
	<do_lookup_map>:0x177	mov 0x2d0(%r8),%rax; mov (%rax,%rdx,4),%r15d
	<do_lookup_map>:0x200	mov 0x2d8(%r8),%rax; mov (%rax,%r15,4),%r15d
	<mbedtls_mpi_safe_cond_assign>:0x98	mov 0x10(%r12),%rcx; movslq %r9d,%rdi; mov %rdi,%rsi; imul (%rcx,%rdx,8),%rsi
	<mbedtls_mpi_get_bit>:0x13	mov 0x10(%rdi),%rax;mov %rsi,%rdx;mov %esi,%ecx;shr $0x6,%rdx;mov (%rax,%rdx,8),%rax
	<mbedtls_mpi_set_bit>:0x32	mov 0x10(%r12),%rax; mov %r13,%rcx; and $0x3f,%ecx; shl %cl,%rbx; lea (%rax,%r14,8),%rdx; mov $0xfffffffffffffffe,%rax; rol %cl,%rax; and (%rdx),%rax
	<mbedtls_mpi_shift_l>:0x4a	mov 0x10(%r13),%rdx; sub %rbx,%rax; lea (%rdx,%rax,8),%rax; mov -0x8(%rax),%rcx
	<mbedtls_mpi_shift_l>:0x8a	mov 0x10(%r13),%rsi; mov $0x40,%edi; mov %r12d,%r8d; sub %r12d,%edi; xor %eax,%eax; mov (%rsi,%rbx,8),%rdx
	<mbedtls_mpi_cmp_abs>:0x7c	mov 0x10(%rdi),%rax; mov -0x8(%rax,%rdx,8),%rdi
	<mpi_montmul.isra.3>:0xa0	mov 0x10(%r15),%rdx; mov (%r14),%rsi; mov -0x58(%rbp),%rdi; mov (%rdx,%r13,8),%r8
	<mbedtls_mpi_cmp_mpi>:0x91	mov 0x10(%rdi),%rcx; mov -0x8(%rcx,%rdx,8),%rsi
	<mbedtls_mpi_mul_mpi>:0x100	mov 0x10(%r13),%rax; mov %r11,%rdx; add 0x10(%rbx),%rdx; mov 0x10(%r12),%rsi; mov %r14,%rdi; mov (%rax,%r11,1),%rcx
	<mbedtls_mpi_mod_int>:0x37	mov 0x10(%rsi),%r11; xor %ecx,%ecx; mov -0x8(%r11,%r10,8),%r9
	<mbedtls_mpi_write_string>:0x129	mov 0x10(%r14),%rax; lea 0x0(,%rdx,8),%ecx; mov (%rax,%r8,1),%rax
	<mbedtls_aes_setkey_enc>:0x108	mov 0xc(%rbx),%edi; add $0x4,%r8; add $0x10,%rbx; mov %rdi,%rdx; movzbl %dh,%edx; movzbl (%rsi,%rdx,1),%ecx
	<mbedtls_aes_setkey_enc>:0x1e8	mov 0x1c(%rbx),%r8d; add $0x20,%rbx; add $0x4,%rdi; mov %r8,%rdx; movzbl %dh,%edx; movzbl (%rsi,%rdx,1),%r9d
	<mbedtls_aes_setkey_enc>:0x238	mov -0x14(%rbx),%edx; mov %ecx,(%rbx); xor -0x1c(%rbx),%ecx; mov %ecx,0x4(%rbx); xor -0x18(%rbx),%ecx; xor %ecx,%edx; mov %ecx,0x8(%rbx); movzbl %dl,%ecx; mov %edx,0xc(%rbx); movzbl (%rsi,%rcx,1),%r9d
	<mbedtls_aes_setkey_enc>:0x2c8	mov 0x14(%rbx),%edi; add $0x18,%rbx; add $0x4,%r8; mov %rdi,%rdx; movzbl %dh,%edx; movzbl (%rsi,%rdx,1),%ecx
Rust-SGX SDK	<dispose_chunk>:0x8a	mov 0x38(%rsi),%r9d; mov %r9,%rcx; lea (%rdi,%r9,8),%r9; cmp 0x258(%r9),%rsi
	<dispose_chunk>:0x299	mov 0x38(%r8),%r9d; mov %r9,%rcx; lea (%rdi,%r9,8),%r9; cmp 0x258(%r9),%r8
	<try_realloc_chunk.isra.2>:0x1eb	mov 0x38(%rsi),%r9d; mov %r9,%rcx; lea (%r12,%r9,8),%r9; cmp 0x258(%r9),%rsi
	<dlmalloc>:0x180b	mov 0x38(%rdx),%r12d; mov %r12,%rcx; add $0x4a,%r12; cmp 0x8(%rsi,%r12,8),%rdx
	<dlfree>:0x391	mov 0x38(%r8),%edi; mov %rdi,%rcx; lea (%rbx,%rdi,8),%rdi; cmp 0x258(%rdi),%r8
	<dlfree>:0x467	mov 0x38(%rsi),%edi; mov %rdi,%rcx; lea (%rbx,%rdi,8),%rdi; cmp 0x258(%rdi),%rsi
	<__hdtoa>:0x170	movsbl (%rdx),%ecx; sub $0x1,%rdx; movzbl (%r12,%rcx,1),%ecx
	<__hldtoa>:0x158	movsbl (%rdx),%ecx; sub $0x1,%rdx; movzbl (%r12,%rcx,1),%ecx
	<__b2d_D2A>:0x1a	movslq 0x14(%rdi),%rax; lea (%r10,%rax,4),%r9; mov -0x4(%r9),%r8d

Table 2: SGXPECTRE Attack Type-II Gadgets in Popular SGX Runtimes.