SGXPECTRE: Stealing Intel Secrets from SGX Enclaves 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*—Speculative execution side-channel vulnerabilities in micro-architecture processors have raised concerns about the security of Intel SGX. To understand clearly the security impact of this vulnerability against SGX, this paper makes the following studies: First, to demonstrate the feasibility of the attacks, we present SGXPECTRE Attacks (the SGX-variants of Spectre attacks) that exploit speculative execution side-channel vulnerabilities to subvert the confidentiality of SGX enclaves. We show that when the 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. Second, to determine whether real-world enclave programs are impacted by the attacks, we develop techniques to automate the search of vulnerable code patterns in enclave binaries using symbolic execution. Our study suggests that nearly *any* enclave program could be vulnerable to SGXPECTRE Attacks since vulnerable code patterns are available in most SGX runtimes (*e.g.*, Intel SGX SDK, Rust-SGX, and Graphene-SGX). Third, we apply SGXPECTRE Attacks to steal seal keys and attestation keys from Intel signed quoting enclaves. The seal key can be used to decrypt sealed storage outside the enclaves and forge valid sealed data; the attestation key can be used to forge attestation signatures. For these reasons, SGXPECTRE Attacks practically defeat SGX's security protection. Finally, we evaluate Intel's existing countermeasures against SGXPECTRE Attacks and discusses the security implications.**

I. INTRODUCTION

Intel's Software Guard eXtensions (SGX) improves application security by removing 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 operating systems and hypervisors are not allowed to directly inspect or manipulate the memory inside the enclaves. 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.*, [3], [55], [22], [64], [93], [75], [58], [74], [96]), and new software tools or frameworks (*e.g.*, [6], [5], [73], [25], [77], [46], [88], [79], [71], [67], [54]) to help developers adopt this emerging programming paradigm. Most recently, SGX has been adopted by commercial public clouds, such as Azure confidential computing [63], [2], aiming to protect cloud data against compromised operating systems or hypervisors, or even "malicious insiders with administrative privilege" [63].

However, the recently disclosed CPU vulnerabilities due to the out-of-order and speculative execution [23] have raised many questions and concerns about the security of SGX. Particularly, the so-called Meltdown [48] and Spectre attacks [44] 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 given these hardware bugs [35].

In this paper, we particularly study Spectre-like attacks against SGX enclaves. We aim to answer the following research questions: (1) Is SGX vulnerable to Spectre attacks? (2) As Spectre attacks require vulnerable code patterns in the target software, do such code patterns exist in real-world enclave programs? (3) What are the consequences of the attacks? (4) Is SGX completely broken due to these hardware bugs? The answers to these questions are critically important to the adoption of the SGX technology and commercialization of SGX-based applications in the future; they are also valuable to the research community in understanding SGX's threat model.

In this paper, we first explore techniques to conduct SGXPECTRE Attacks. SGXPECTRE is the term we coined for the SGX-variants of the Spectre attacks. This is to differentiate from other variants of Spectre attacks. At a high level, SGXPECTRE exploits the race condition between the speculatively executed memory references and the latency of the branch resolution, in order to generate side-channel observable cache traces and consequently read memory content. Specifically, we explore how branch targets can be injected into SGX enclaves, how registers inside enclaves can be controlled by the outside world, how information can be leaked through side channels, and how the adversary could increase the probability of winning the race condition. These techniques are the key components for successfully performing SGXPECTRE Attacks. *To the best of our knowledge, they have never been studied in previous works.*

Second, we develop techniques to automate the search of vulnerabilities in enclave binaries. We observe SGXPECTRE Attacks are enabled by two types of code gadgets in the enclave binary. To help the enclave developers detect vulnerabilities in their code, we develop binary analysis tools to symbolically execute enclave code and automatically identify such gadgets from enclave binaries. As a result, we found both types of gadgets exist in widely used SGX runtimes, such as Intel SGX SDK, Rust-SGX SDK, and Graphene-SGX library OS. Therefore any enclave program built with these runtimes would be vulnerable to SGXPECTRE Attacks. *To our knowledge, our tool is the first to perform symbolic execution on enclave binary (which we have open-sourced on GitHub). It is also the first tool to automatically detect software vulnerabilities that enable Spectre-like attacks. We expect our study will inspire future research.*

Third, we demonstrate end-to-end attacks to validate the fidelity of SGXPECTRE Attacks and extract Intel's secrets. Particularly, we show that the adversary could learn the content of the enclave memory as well as its register values from a victim enclave. An even more alarming consequence is that SGXPECTRE Attacks can be leveraged to steal secrets belonging to Intel SGX platforms, such as *provisioning keys*, *seal keys*, and *attestation keys*. For example, we have demonstrated that SGXPECTRE Attacks are able to read memory from the quoting enclave developed by Intel and extract Intel's seal key, which can be used to decrypt the sealed EPID blob to extract the attestation key (*i.e.*, EPID private key). With an attestation key, the adversary could compromise a large group of SGX platforms that share the same EPID public key. *Our work was one of the first to demonstrate the extraction of Intel's secrets.*

Fourth, we investigate the security implication of SGXPECTRE Attacks on the SGX ecosystem. We enumerate all derived keys and in-memory secrets of Intel's SGX platforms, and study how Intel mitigate the threats to these in-memory secrets by having them depending on the version of the microcode of the SGX platform. *This paper contributes to the overall understanding of the security implications of* SGXPECTRE *Attacks and similar attacks targeting the confidentiality of SGX platforms.*

The rest of the paper is organized as follows: Sec. II introduces key concepts of Intel processor micro-architectures to set the stage of our discussion. Sec. III discusses the threat model. Sec. IV presents a systematic exploration of attack vectors in enclaves and techniques that enable practical attacks. Sec. V presents a symbolic execution tool for searching instruction gadgets in enclave programs in an automated manner. Sec. VI shows end-to-end SGXPECTRE Attacks against enclave runtimes that lead to a complete breach of enclave confidentiality. Sec. VII discusses and evaluates countermeasures against the attacks. Sec. VIII discusses the security implications of side channels on SGX platforms. Sec. IX discusses related work and Sec. X concludes the paper.

II. BACKGROUND

A. Intel Processor Internals

Out-of-order execution. Modern CPUs 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 the branch target may take time, a prediction is made, based on its history, to prefetch instructions first. If the prediction is correct, speculatively executed instructions may retire; otherwise, mis-predicted execution will be rewinded. 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 softwares 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" [31]. 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.

B. Intel SGX

Intel SGX is an architecture extension in recent Intel processors aiming to offer strong 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.

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 determines 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 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.

Remote Attestation. SGX remote attestation is used by enclaves to prove to the ISV (*i.e.*, the enclave developer) that a claimed enclave is running inside an SGX enabled processor. An anonymous signature scheme, called Intel *Enhanced Privacy ID* (EPID) [40], is used to produce the attestation signature. The attestation key (*i.e.*, EPID private key) cannot be directly accessed by an attested enclave, otherwise a malicious enclave could generate any valid attestation signature to deceive the remote party. Hence, Intel issues two privileged enclaves, called the *provisioning enclave* and the *quoting enclave* to manage the attestation key and sign attestation data.

Sealed storage. Enclaves can encrypt and integrity-protect some secrets, *e.g.*, the attestation key, via *sealing* to store the secrets outside the enclave, *e.g.*, on a non-volatile memory. The encryption key used during the sealing process is called the *seal key*, which is derived via `EGETKEY` instruction.

C. 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 [61], [60], [94], [57], [1], [76], [50], [39], 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 [20], [91], [92], [7], [95], [4], the adversary shares some physical memory pages (*e.g.*, through dynamic shared libraries) with the victim. By issuing `clflush` on certain virtual addresses 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.

```
1 if (x < array1_size)
2     y = array2[array1[x] * 4096];
```

Listing 1. An example of bounds check bypass [44]

D. Spectre Attacks

Spectre attacks [44], [23] leverage hardware vulnerabilities due to speculative execution to extract memory content that should not be accessible by the adversary. Originally there were two variants of Spectre attacks: bounds check bypass and branch target injection. The first variant targets the conditional branch prediction. An example of this variant is shown in Listing 1: A conditional branch is used to check whether input `x` is within the bounds of the array (line 1 in Listing 1). However, when the value of `x` is out-of-bounds, due to the misprediction of the conditional branch (*i.e.*, by the hardware branch prediction unit), speculative out-of-bounds memory access may happen before the bounds check is resolved, which triggers implicit caching (loading a particular memory address of `array2` to CPU cache) that reflects the out-of-bounds memory content of `array1`. The adversary could then leverage cache side channels to learn the state of the implicit caching and infer the data values.

The second variant targets the indirect branch prediction. Particularly, the adversary first manipulates the branch target buffer (BTB) such that when the victim process executes a indirect branch instruction, the BTB will mispredict the target address to speculatively execute code that could never be executed by normal control flows. Similar to the first variant of Spectre attacks, sensitive data can be extracted using cache side channels. As the code patterns in Listing 1 rarely exist in real-world code, this paper explores the second variant of Spectre attacks on SGX.

III. 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 over the entire OS, including re-compiling the kernel and rebooting the OS with arbitrary arguments.
- *Interacting with the targeted enclave:* We assume the adversary is able to launch the targeted enclave with a software program under her control and to enter the enclaves with parameters under her control.
- *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

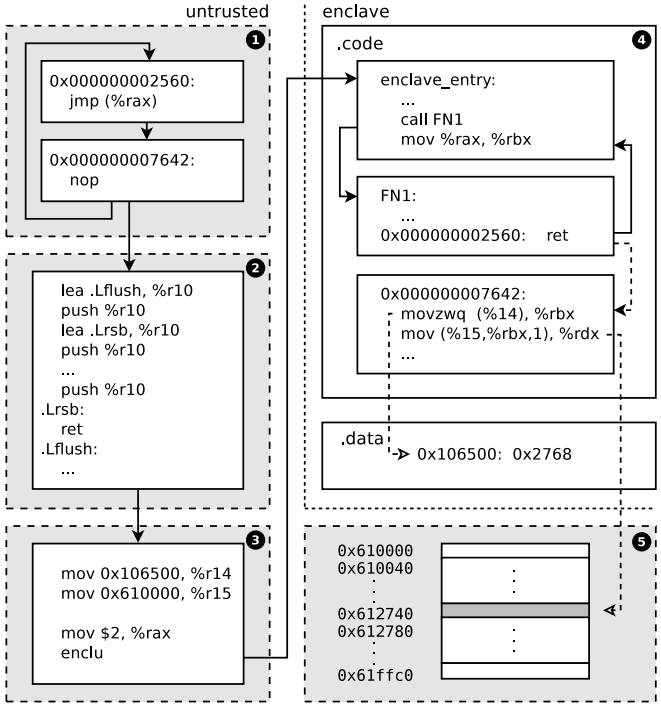

Fig. 1. A simple example of SGXPECTRE Attacks. The gray blocks represent code or data outside the enclave. The white blocks represent enclave code or data.

implies that the enclave can poison any BTB entries used by the targeted enclave.

We assume the binary code of the targeted enclave is already known to the adversary and does not change during execution. Therefore, we assume that the adversary is primarily interested in extracting the secrets that have been provisioned into the enclaves, either by *Intel* or by *regular enclave developers*.

IV. SGXPECTRE ATTACKS

A. A Simple Example

Steps of an SGXPECTRE Attack are illustrated in Fig. 1. **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 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. IV-B.

Step ❷ is to prepare a CPU environment to increase the chance of speculatively executing the secret-leaking instructions before the processor detects the mis-prediction 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 instruction) 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. IV-E). The code for depleting the RSB (shown in Fig. 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 register values 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. IV-C.

Step ❹ is to trigger the enclave code. Because of the BTB poisoning, instructions at address 0x07642 will be executed speculatively while 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. IV-D.

B. Injecting Branch Targets into Enclaves

The branch prediction units in modern processors typically consist 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 [31], [41]. For space efficiency, many Intel processors, such as Skylake, use only 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 [17]. On Intel Skylake or later processors, when RSB underflows, BTBs will be used for prediction 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 Fig. 2).

[1] `Call` instructions with zero displacements 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.

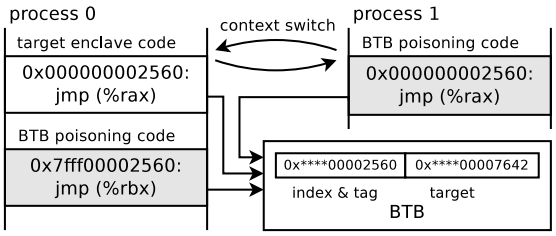

Fig. 2. Poisoning BTB from the same process or a different process

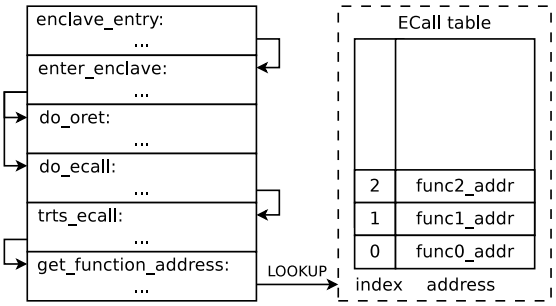

Fig. 3. `EENTER` and `ECall` table lookup

- *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 BTB uses only the lower 32 bits of the source address to index a BTB entry, the adversary could reserve a $2^{32} = 4$GB memory buffer, and execute an indirect jump instruction (within the buffer) whose source address (*e.g.*, 0x7fff00002560) is the same as the branch instruction in the target enclave (*i.e.*, 0x02560) in the lower 32 bits, and target address (*e.g.*, 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 the 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). 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. We speculate that this is an architecture-specific implementation. A more reliable way to deplete the RSB is through the use of AEX as described in Sec. VI-A.

C. Controlling Registers in Enclaves

Because all registers are restored by hardware after `ERESUME`, the adversary is not able to control any register 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 [32] 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 "Outside Call", is a function call to exit the enclave mode. Returning from an `OCall` is called an `ORet`. Both `ECall`s and `ORet`s 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.

D. Leaking Secrets via Side Channels

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

Cache side-channel attacks against enclave programs have been studied recently [66], [8], [21], [19], all of which demonstrated that a program runs outside the enclave may use PRIME+PROBE techniques [76] to infer 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 [91] cannot be used in SGX attacks since 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 locations outside the enclave, as shown in Fig. 1. The adversary can flush an array of memory before the attack, such as the array from address 0x610000 to 0x61ffff,

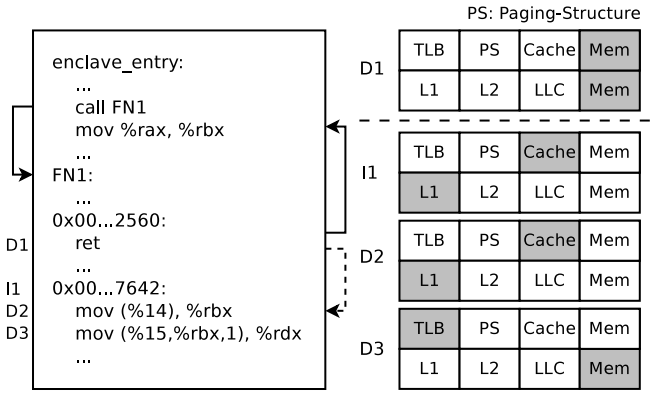

Fig. 4. Best scenarios for winning a race condition. Memory accesses `D1`, `I1`, `D2`, `D3` are labeled next to the related instructions. The address translation and data accesses are illustrated on the right: The 4 blocks on top denote the units holding the address translation information, including 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.

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.

E. 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 winning 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 achieved by performing **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` should 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. VI, 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.

V. 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 that symbolically executes the enclave code to examine code patterns in SGX runtimes, and have identified those code patterns in every runtime library we have examined, including Intel's SGX SDK [36], Rust-SGX [14], Graphene-SGX [77]. In this section, we present how we search these gadgets in greater detail.

A. 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 close to each other and collectively reveal some enclave memory content through cache side channels. Borrowing the term used in return-oriented programming [68] and Spectre attacks [44], we use *gadgets* to refer to these patterns. More specifically, we name them *Type-I gadgets* and *Type-II gadgets*, respectively.

Type-I gadgets: branch target injection. A gadget is a sequence of instructions that are executed sequentially during one run of the enclave program (but not necessarily 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 is not overwritten by the software before one of the three types of branch instructions is met, a Type-I gadget is found. An example of a Type-I gadget is shown in Listing 2, which is excerpted from `libsgx_trts.a` of Intel SGX SDK. In particular, line 49 in Listing 2 is the first return

```
0000000000003662 <enclave_entry>:
3662: cmp     $0x0,%rax
3666: jne     3709 <enclave_entry+0xa7>
366c: xor     %rdx,%rdx
366f: mov     %gs:0x8,%rax
3676: 00 00
3678: cmp     $0x0,%rax
367c: jne     368d <enclave_entry+0x2b>
367e: mov     %rbx,%rax
3681: sub     $0x10000,%rax
3687: sub     $0x2b0,%rax
368d: xchg    %rax,%rsp
368f: push    %rcx
3690: push    %rbp
3691: mov     %rsp,%rbp
3694: sub     $0x30,%rsp
3698: mov     %rax,-0x8(%rbp)
369c: mov     %rdx,-0x18(%rbp)
36a0: mov     %rbx,-0x20(%rbp)
36a4: mov     %rsi,-0x28(%rbp)
36a8: mov     %rdi,-0x30(%rbp)
36ac: mov     %rdx,%rcx
36af: mov     %rbx,%rdx
36b2: callq   1f20 <enter_enclave>
...

0000000000001f20 <enter_enclave>:
1f20: push    %r13
1f22: push    %r12
1f24: mov     %rsi,%r13
1f27: push    %rbp
1f28: push    %rbx
1f29: mov     %rdx,%r12
1f2c: mov     %edi,%ebx
1f2e: mov     %ecx,%ebp
1f30: sub     $0x8,%rsp
1f34: callq   b60 <sgx_is_enclave_crashed>
...

0000000000000b60 <sgx_is_enclave_crashed>:
b60: sub     $0x8,%rsp
b64: callq   361b <get_enclave_state>
...

000000000000361b <get_enclave_state>:
361b: lea     0x213886(%rip),%rcx      # 216ea8 <
      g_enclave_state>
3622: xor     %rax,%rax
3625: mov     (%rcx),%eax
3627: retq
```

Listing 2. An example of a Type-I gadget

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 `rbx`, `rdi`, `rsi`, `r8`, `r9`, `r10`, `r11`, `r14`, and `r15`.

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

```
0000000000005c10 <dlfree>:
...
607f: mov     0x38(%rsi),%edi
6082: mov     %rdi,%rcx
6085: lea     (%rbx,%rdi,8),%rdi
6089: cmp     0x258(%rdi),%rsi
...
```

Listing 3. An example of a Type-II gadget

FLUSH+RELOAD side channel to extract the value of `regB`. An example of a Type-II gadget is illustrated in Listing 3, which is excerpted from the `libsgx_tstdc.a` library of Intel SGX SDK. Assuming `rsi` is a register controlled by the adversary, the first instruction (line 3) reads the content of memory address pointed to by `rsi+0x38` to `edi`. Then the value of `rbx+rdi×8` is stored in `rdi` (line 5). Finally, the memory address at `rdi+0x258` is loaded to be compared with `rsi` (line 6). To narrow down the range of `rdi+0x258`, it is desired that `rbx` is also controlled by the adversary. We use `regC` to represent these base registers like `rbx`.

B. Symbolically Executing SGX Code

Although a skillful developer can manually read the source code or even the disassembled binary code of an enclave program and runtime libraries to identify exploitable gadgets, 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 gadgets, and eliminate them before deploying them to untrusted SGX machines.

To this end, we devise a dynamic symbolic execution technique to enable automated identification of SGXPECTRE Attack gadgets. Symbolic execution [43] 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 enclave programs to find gadgets of SGXPECTRE Attacks.

Symbolic execution of an enclave function. We design a tool built atop the `angr` [72], a popular binary analysis framework to perform the symbolic execution. To avoid the path explosion problem in symbolically executing a large enclave program (or a large SGX runtime such as Graphene-SGX), our tool allows the user to specify an arbitrary enclave function to start the symbolic execution. During the symbolic execution, machine states are maintained internally 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. 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`.

C. 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 transferred 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 `movq`) that load register `regB` with data from the memory location pointed to by `regA`. Both `regA` and `regB` are general registers, such as `rax`, `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 is another memory reference instruction (*e.g.*, `mov`, `cmp`) 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. A 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.

D. Experimental Results of Gadget Detection

We run our symbolic execution tool on three well-known SGX runtimes: the official Intel Linux SGX SDK (version 2.1.102.43402), Rust-SGX SDK (version 0.9.1), and Graphene-SGX (commit bf90323). In all cases, a minimal enclave with a single empty `ECall` was developed for analysis, because gadgets detected in a minimal enclave binary will appear in any enclave code developed using these SDKs. When the enclave binary becomes more complex, the size of the resulting enclave binary will grow to include more components of the SDK libraries, and the number of available gadgets will also increase. For example, a simple `OCall` implementation of `printf()` introduces three more Type-II gadgets. In addition, the code written by the enclave author might also introduce extra exploitable gadgets.

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 `OCall`s. In contrast, Graphene-SGX has more diverse `OCall` 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 `OCall` sites consumes more time: the median running time of analyzing one `OCall` site 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 I. In Table I, column 2 shows the type of the gadget, whether it being an *indirect jump*, *indirect call*, or *return*; column 3 shows only the gadget's end address (because Type-I gadgets always start at the `enclave_entry()`), which is the address of an branch instruction, 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 I 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 the adversary are `rdi`, `r8`, `r9`, `r10`, `r11`, `r14` and `r15`.

Due to space limit, Type-II gadgets are not listed in the paper. The results are highlighted as follows: For 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), we have found 6 gadgets in Intel's SGX SDK, 6 gadgets in Rust-SGX, and 18 gadgets in Graphene-SGX. For Type-II gadgets of the form [`regA`, `regB`], we have found 6, 86, and 180 such gadgets in these three runtime libraries, respectively.

VI. Stealing Enclave Secrets

In this section, we demonstrate two end-to-end SGXPECTRE Attacks against SGX enclave programs. In the first example, we show how SGXPECTRE Attacks could read register values from arbitrary enclave program developed using Intel SGX SDK [36]. In the second example, we demonstrate the extraction of Intel's secrets (*e.g.*, attestation keys) using SGXPECTRE Attacks. Both experiments were conducted on a

	Category	End Address	Controlled Registers
Intel SGX SDK	indirect jump	<do_ecall>:0x118	rdi, r8, r9, r10, r11, r14, r15
	indirect call	—	—
	return	<get_enclave_state>:0xc	rbx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<sgx_is_enclave_crashed>:0x16	rbx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<get_thread_data>:0x9	rbx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<_ZL16init_stack_guardPv>:0x21	rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<do_ecall>:0x21	rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<enter_enclave>:0x62	rbx, rsi, r8, r9, r10, r11, r12, r13, r14, r15
		<restore_xregs>:0x2b	rsi, r8, r9, r10, r11, r12, r14, r15
		<do_rdrand>:0x11	r8, r9, r10, r11, r12, r14, r15
		<sgx_read_rand>:0x46	rbx, r8, r9, r10, r11, r12, r14, r15
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
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

TABLE I
SGXPECTRE ATTACK TYPE-I GADGETS IN POPULAR SGX RUNTIME LIBRARIES.

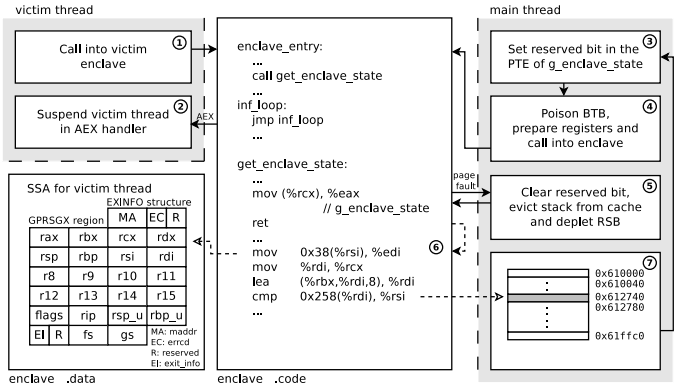

Fig. 5. Exploiting Intel SGX SDK. The blocks with dark shadows represent instructions or data located in untrusted memory. Blocks without shadows are instructions inside the target enclave or the `.data` segment of the enclave memory.

Lenovo Thinkpad X1 Carbon (4th Gen) laptop with an Intel Core i5-6200U processor and 8GB memory.

A. Reading Register Values from Arbitrary Enclaves

We first demonstrate an attack that enables the adversary to read arbitrary register values inside an *arbitrary* enclave program written with Intel SGX SDK [36], 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 2.1.102.43402 (compiled with `gcc` version 5.4.020160609), which are listed in Listing 2 and Listing 3.

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 and its address is fixed when loading the enclave, the privileged adversary could leverage SGXPECTRE Attacks to read 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 [81] and take snapshots of its SSAs to single-step trace its register values during its

execution.

In particular, the attack is shown in Fig. 5. In **Step** ①, the targeted enclave code is loaded into an enclave that is created by a malicious 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. The 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 AEXs from the targeted enclave. During an AEX, the processor stores its register values into the SSA, 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 the enclave memory page that holds `g_enclave_state`, a global variable used by Intel SGX SDK to track the state of the enclave, *e.g.*, initialized or crashed states. As shown in Listing 2, 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 poisons the BTB, prepares registers (*i.e.*, `rsi` and `rdi`), and executes `EENTER` to trigger the attack. Note that `rbx` will be set to `rdi` by the time the `ret` instruction is executed (line 34 in Listing 2), in such a way we can control `rsi` and `rbx` when speculatively executing Type-II gadget. 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 `ret` 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 global variable `g_enclave_state`. 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. The RSB is depleted right before `ERESUME` from the fault handling, so that it will remain empty until the `ret` instruction of Type-I gadget is executed. 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 and RSB depletion). After the processor detects the mis-prediction and flushes speculatively executed instructions from the pipeline, the enclave code continues to execute. However, because `rdi` is set as a memory address in our attack, 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 enclave. 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 is very short as it contains only 4 instructions. Since calling into this 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 using FLUSH+RELOAD. However, if we know the value of lower 24 bits, we can adjust the base of the second memory access (*i.e.*, `rbx`) 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 some initial consecutively known bytes. For example, the unused bytes in an enclave data page will be initialized as 0x00, 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.

Although the demonstrated attack only targets the register values, we note 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 begin with a relatively small address which is likely unused and thus is known to

be initialized with 0xcc. In this way, the adversary can start reading the stack frames from these known bytes.

B. Stealing Intel Secrets

Next, we show how to steal Intel secrets, such as seal keys and attestation keys, from Intel's prebuilt and signed quoting enclave, *i.e.*, `libsgx_qe.signed.so` (version 2.1.2). All the attacks described below have been empirically validated on a Lenovo Thinkpad X1 Carbon (4th Gen) laptop with an Intel Core i5-6200U processor.

The demonstrated attack involves first extracting seal keys of the quoting enclave and then decrypting sealed storage blob for the attestation keys. More particularly, the adversary could use SGXPECTRE Attacks to read the seal keys from the enclave memory when it is being used during sealing or unsealing operations. In our demonstration, we targeted Intel SDK API `sgx_unseal_data()` used for unsealing a sealed blob. The `sgx_unseal_data()` API works as follows: firstly, it calls `sgx_get_key()` function to derive the seal key and then store it temporarily on the stack in the enclave memory. Secondly, with the seal key, it calls `sgx_rijndael128GCM_decrypt()` function to decrypt the sealed blob. Finally, it clears the seal key (by setting the memory range storing the seal key on the stack to 0s) and returns. Hence, to read the seal key, the adversary suspends the execution of the victim enclave when function `sgx_rijndael128GCM_decrypt()` is being called, by setting the reserved bit of the PTE of the enclave code page containing `sgx_rijndael128GCM_decrypt()`. The adversary then launches SGXPECTRE Attacks to read the stack and extract the seal key.

To decrypt the sealed blob, the adversary could export the seal key and then use an AES-128-GCM decryption algorithm implemented by herself. This may happen outside the enclave or on a different machine, because the SGX hardware is no longer involved in the process. We have validated the attacks on our testbed.

Extracting the seal key of the quoting enclave. The quoting enclave has two `ECall` functions: `verify_blob()`, which is used to verify the sealed EPID blob, and `get_quote()`, which is used to generate a quote on behalf of an attested enclave for remote attestation. Particularly, `verify_blob()` calls an internal function `verify_blob_internal()`, which further calls the `sgx_unseal_data()` API to unseal the EPID blob. So we targeted the `verify_blob()` `ECall` function, suspended its execution when `sgx_rijndael128GCM_decrypt()` was being called, and read the stack to obtain the quoting enclave's seal key. These steps have been described in the previous paragraphs.

Extracting attestation key. After running the provisioning protocol with Intel's provisioning service, an attestation key (*i.e.*, EPID private key) is created and then sealed in an EPID blob by the provisioning enclave and stored on a non-volatile memory. Though the location of the non-volatile memory is not documented, during remote attestation, SGX still relies on the untrusted OS to pass the sealed EPID blob into the quoting enclave. This offers the adversary a chance to obtain the sealed EPID blob. With the extracted seal key of the quoting enclave, we could decrypt the sealed EPID blob to extract the EPID private key.

After obtaining the attestation key, the adversary could use this EPID private key to generate an anonymous group signature and pass the remote attestation. This means the adversary can now impersonate any machine in the EPID group. Moreover, the adversary could also use the attestation key completely outside the enclave and trick the ISVs to believe their code runs inside an enclave. This attack has been validated in our experiments, by generating a valid signature of a quote from an ISV's enclave without running it on SGX.

We note here that one challenge we have addressed in attacking the Intel signed quoting enclave is that the TCS number of the quoting enclave is set to 1, which means the adversary has to use the same TCS to enter the enclaves. SGXPECTRE Attacks are still possible as the number of SSAs per TCS is 2, which is designed to allow the victim to run exception handlers within the enclave when the exception could not be resolved outside the enclave during AEXs. However, this also enables the adversary to `EENTER` into the enclave during an AEX, thus launching the SGXPECTRE Attack to steal the secrets being used by the quoting enclave.

VII. Evaluating Existing Countermeasures

Hardware patches. To mitigate branch target injection attacks, Intel has released microcode updates to support the following three features [34].

- *Indirect Branch Restricted Speculation (IBRS):* IBRS restricts the speculation of indirect branches [38]. Software running in a more privileged mode can set a 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 the `IA32_SPEC_CTRL.STIBP` MSR to 1.
- *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 code before the barrier. The barrier can be established by setting the `IA32_PRED_CMD.IBPB` MSR.

Particularly, IBRS 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 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 (by introducing page faults) 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. VI, 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 and during ERESUME, and also prevent branch injection via Hyper-Threading, thus they are immune to SGXPECTRE Attacks.

Retpoline. Retpoline is a pure software-based solution to Spectre attacks [78], which has been developed for major compilers, such as GCC [89] and LLVM [9]. Because modern processors have implemented separate predictors for function returns, such as Intel's return stack buffer [26], [27], [28], [29], [30] and AMD's return-address stack [41], it is believed that these return predictors are not vulnerable to Spectre attacks. Therefore, the key idea of retpoline is to replace indirect jumps 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.

VIII. Is SGX Broken?

In the previous sections, we have shown that SGXPECTRE Attacks lead to confidentiality breaches for both Intel's enclaves and developers' enclaves. In this section, we aim to understand the security implications of SGXPECTRE Attacks (as well as other similar attacks due to speculative or out-of-order execution [80]): Is SGX completely broken under these threats?

A. Intel's Secrets

As demonstrated in this paper, all secrets in the memory (or registers saved in the SSA during AEX) can be extracted by SGXPECTRE Attacks. We believe all secrets that are exposed in the enclave memory (even only temporarily) can be exfiltrated by these attacks. While all secrets in developers' enclaves are exposed, however, not all Intel's secrets can be stolen in the same manner. Specifically, Intel's secrets for its SGX platforms can be found in Fig. 6. Next, we explain in details how these secrets are affected by SGXPECTRE Attacks.

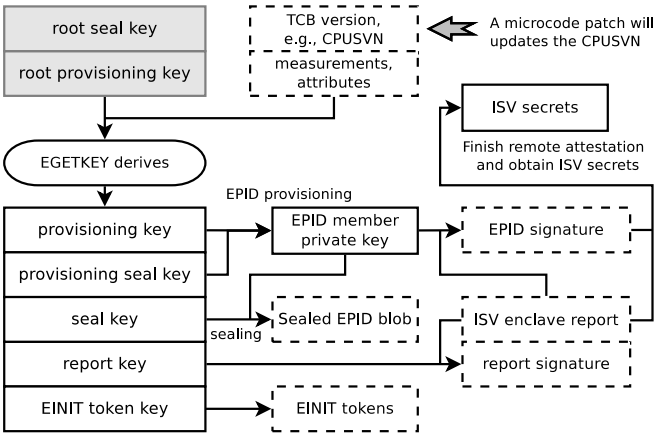

Fig. 6. Intel's secrets and key derivation.

Intel's root secrets. For Intel's infrastructure services to trust an SGX machine, during the manufacturing process, Intel generates a *root provisioning key* at its internal key generation facility, and burns it into the e-fuse of an SGX machine. The root provisioning key is also stored in an Intel's database to be referenced by Intel's provisioning service. As such, the root provisioning key serves as a shared secret that is only known by Intel and the underlying SGX machine [40]. A 128-bit *root seal key* is generated inside the processor chip during the manufacturing process [31]. This root seal key is not known by Intel. For improving security, these two keys can only be accessed through the EGETKEY instruction and EREPORT instruction, but never exported to enclaves' protected memory.

Derived keys. Derived secrets of an SGX platform include the *provisioning key*, the *provisioning seal key*, the *report keys*, the *seal keys*, the EINIT token key. The provisioning key is the secret used to establish trust during the provisioning protocol; the provisioning seal key is a symmetric key used to generate an encrypted backup copy of the attestation key to be stored in Intel's provisioning service; the EINIT token key is used only by the launch enclave to sign the EINIT token of a legitimate enclave. A report key is a symmetric key possessed by each enclave. The EREPORT instruction is used by an enclave to generate a report of its execution context and produce a CMAC tag of the report using the report key of a specified enclave (*e.g.*, the quoting enclave). However, the report key is not exported to the memory in this process as it is kept secret to the specified enclave; it can only be exported to the memory by the owner enclave using the EGETKEY instruction. A seal key is used by an enclave to encrypt and decrypt the sealed storage; therefore, there could be multiple seal keys for each enclave, which can be identified using their KEYID. It is also possible for enclaves from the same ISV to share the seal keys. *All of these derived secrets*

*can be exposed in the enclave memory, either by developers' enclaves or Intel's enclaves (*e.g.*, quoting enclaves and launch enclaves). Therefore,* SGXPECTRE *Attacks are able to extract all of them.*

EPID signing keys for attestation. In the EPID provisioning protocol, the SGX platform first sends a message to Intel's provisioning service that contains the platform identifier (`PPID`) and the trusted computing base (TCB) version. Upon receiving the message, Intel's provisioning service verifies the `PPID` and *selects* an EPID group for the SGX platform. Intel assigns SGX platforms with the same CPU type and the same TCB version the same EPID group, which contains millions of machines [40]. Intel's provisioning service then sends the EPID group public key to the SGX platform. With the group public key, the provisioning enclave runs an EPID join protocol with Intel's provisioning service to generate an EPID private key. The EPID private key is sealed using Intel's seal key for later use. Each TCB version only requires running the provisioning protocol once. *Breach of one EPID private key might invalidate the entire EPID group. Unfortunately, the EPID private key is also an in-memory secret that can be extracted by* SGXPECTRE *Attacks, as demonstrated in this paper.*

Summary. The relationship among these keys are illustrated in Fig. 6. The gray boxes represent secrets only in the hardware and firmware, and the white boxes represent secrets exposed to memory. Dashed boxes represent values that are known to the platform, some of which are used to derive secrets. We can see that all Intel's secrets, except the root seal key and root provisioning key, can be exposed by SGXPECTRE Attacks.

B. Defense via Centralized Attestation Services

Defenses by Intel's attestation service. Although the IBRS microcode patch defeats SGXPECTRE Attacks, unpatched processors remain vulnerable. The key to the security of the SGX ecosystem is whether attestation measurements and signatures from processors without the IBRS patch can be detected during remote attestation. Indeed, Intel's attestation service arbitrates every attestation request from the ISV, detects attestation signatures generated from unpatched CPUs, and responses to the ISV with an error message indicating outdated `CPUSVN` (see Table II). Therefore, the combination of the microcode patches and defenses by Intel's attestation service has been an effective defense against SGXPECTRE Attacks (and also the Foreshadow attack [80]).

Implications for developer enclaves. Despite the defense, developers should be aware of the security implications of running (or having run) an enclave on unpatched SGX processors. First, any secret provisioned to an unpatched processor can be leaked. This includes secrets in enclaves that are provisioned before the remote attestation, or after the remote attestation if the ISV chooses to ignore the error message returned by the attestation service. Moreover, because the ISV enclave's seal key can be compromised by SGXPECTRE Attacks, any secret sealed by an enclave run on an unpatched processor can be decrypted by the adversary. Furthermore, any legacy sealed secrets become untrustworthy, as they could be forged by the adversary using the stolen seal key.

Result	Description	Trustworthy
OK	EPID signature was verified correctly and the TCB level of the SGX platform is up-to-date.	Yes
SIGNATURE_INVALID	EPID signature was invalid.	No
GROUP_REVOKED	EPID group has been revoked.	No
SIGNATURE_REVOKED	EPID private key used has been revoked by signature.	No
KEY_REVOKED	EPID private key used has been directly revoked (not by signature).	No
SIGRL_VERSION_MISMATCH	SigRL version does not match the most recent version of the SigRL.	No
GROUP_OUT_OF_DATE	EPID signature was verified correctly, but the TCB level of SGX platform is outdated.	Up to ISV
CONFIGURATION_NEEDED	EPID signature was verified correctly, but but additional configuration of SGX platform may be needed	Up to ISV

TABLE II
ATTESTATION RESULTS [33]

Second, as the EPID private key used in the remote attestation can be extracted by the attacker, the attacker can provide a valid signature for any SGX processors in the EPID group [40]. With the attestation key, it is also possible for the attacker to run the enclave code entirely outside the enclave and forge a valid signature to fool the ISV. As shown in Table II, Intel currently rely on ISV to make their own decisions after receiving this error message. An error message during attestation with `GROUP_OUT_OF_DATE` or `CONFIGURATION_NEEDED` implies that the enclave cannot be trusted at all.

IX. RELATED WORK

Spectre. Our work is closely related to the recently demonstrated Spectre attacks [44], [23]. A variety of attack scenarios have been demonstrated, including cross-process memory read [44], kernel memory read from user processes, and host memory read from KVM guests [23]. While these attacks mainly abuse BTB, Spectre-like attacks using other micro-architectural components, such as Return Stack Buffer (RSB) [52] and Store To Load (STL) [24] have also been demonstrated. Further, NetSpectre [65] demonstrated that remote adversaries could launch Spectre-like attacks over network.

In the context of SGX, O'Keeffe *et al.* [59] demonstrated a Spectre-like attack, particularly the bounds check bypass variant, against SGX enclaves, and Koruyeh *et al.* [45] demonstrated a Spectre attack using RSB against SGX enclaves. Both attacks are only proof-of-concept, which target uncommon enclave codes that are specially developed by the authors. In contrast, in this paper we systematically investigate the security of SGX enclaves due to vulnerable speculative execution, by enumerating attack vectors and techniques, developing binary analysis tools for automated gadget identification, and demonstrating end-to-end attacks against arbitrary enclaves. A particular difference in this paper is that we demonstrate successful extraction of Intel's secrets from Intel signed enclaves.

Meltdown. Meltdown attacks [48] 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.* [48] 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 [48].

Foreshadow and L1TF. Concurrent to our work, Van Bulck *et al.* [80] introduced a Meltdown-style attack, called Foreshadow, against SGX enclaves. The Foreshadow attack leverages a new hardware vulnerability called L1 Terminal Fault (L1TF) [37] to read any enclave memory that resided in L1 cache. Both Foreshadow and SGXPECTRE can be used to extracts Intel's secrets from Intel signed enclaves. However, these two attacks exploit different types of hardware vulnerabilities. Though both Foreshadow and SGXPECTRE have been patched by microcode updates, they would motivate researchers to rethink SGX's security model in future research.

Micro-architectural side channels on SGX. SGXPECTRE Attacks are variants of micro-architectural side-channel attacks on SGX, in which the adversary illegally learns secrets inside SGX enclaves. Although it has already been demonstrated that by observing execution traces of an enclave program left in the CPU caches [66], [8], [21], [19], branch target buffers [47], DRAM's row buffer contention [85], page-table entries [82], [85], and page-fault exception handlers [90], [70], 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. In contrast, the consequences of SGXPECTRE Attacks are far more concerning.

Side-channel defenses. Existing countermeasures to side-channel attacks can be categorized into three classes: hardware solutions, system solutions, and application solutions. Hardware solutions [86], [87], [15], [53], [51], [12] 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 [42], [83], [49], [97], but as they require trusted system software, they cannot be directly applied to SGX.

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 [56], [11], [70], or to diversify or randomize memory access patterns at runtime to conceal the true execution traces [13], [62]. 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 [16], [84]. 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 [69], [10], [18]. These approaches can be used to detect frequent AEX, but still allowing secret leaks in SGXPECTRE Attacks.

Randomization SGX-Shield [67] implemented fine-grained ASLR for enclave programs. It uses a secure in-enclave loader to perform randomization. However, the randomization process itself involves accesses to the enclave memory and thus can be learnt by SGXPECTRE Attacks. Hence, such randomization based mitigation has limited effect on SGXPECTRE Attacks.

X. CONCLUSION

In this paper, we studied techniques to perform SGXPECTRE Attacks, developed a symbolic execution tool to automatically detect exploitable gadgets, demonstrated end-to-end attacks to show how secrets (including Intel's secrets) can be extracted, and discussed the security implications on the SGX ecosystems. Our study concludes that SGXPECTRE Attacks are powerful to extract any in-memory secrets from SGX enclaves (including register values that are stored in memory during AEX), but also points out Intel's control of enclave attestation provides a layer of defense that effectively mitigates such vulnerabilities via microcode updates.

ACKNOWLEDGMENTS

The work was supported in part by the NSF grants 1750809, 1718084, 1834213, and 1834215.

REFERENCES

[1] O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-Cache," in *2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.
[2] Alicloud, "ECS bare metal instance," 2018, https://www.alibabacloud.com/product/ebm.
[3] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," in *2nd HASP*, 2013.
[4] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on AES," in *Cryptology ePrint Archive*, 2014.
[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel SGX," in *12th USENIX OSDI*, 2016.
[6] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," *ACM Transactions on Computer Systems*, vol. 33, no. 3, Aug. 2015.
[7] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way," in *Cryptology ePrint Archive*, 2014.
[8] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies*, 2017.
[9] C. Carruth, "Retpoline patch for LLVM," https://reviews.llvm.org/D41723, 2018.

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

[11] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *30th IEEE Symposium on Security and Privacy*, 2009.

[12] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium*, 2016.

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

[14] Y. Ding, R. Duan, L. Li, Y. Cheng, Y. Zhang, T. Chen, T. Wei, and H. Wang, "Poster: Rust SGX SDK: Towards memory safety in Intel SGX enclave," in *ACM Conference on Computer and Communications Security*, 2017.

[15] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012.

[16] G. Doychev, D. Feld, B. Köpf, and L. Mauborgne, "CacheAudit: A tool for the static analysis of cache side channels," in *22st USENIX Security Symposium*, 2013.

[17] A. Fog, "The microarchitecture of Intel, AMD and VIA cpus: An optimization guide for assembly programmers and compiler makers," *Copenhagen University College of Engineering*, 2017.

[18] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults," in *International Symposium on Research in Attacks, Intrusions and Defenses*, 2017.

[19] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *EUROSEC*, 2017.

[20] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *32nd IEEE Symposium on Security and Privacy*, 2011, pp. 490–505.

[21] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX ATC*, 2017.

[22] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *2nd HASP*, 2013.

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

[24] ——, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[25] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX OSDI*, 2016.

[26] Intel, "Method and apparatus for implementing a speculative return stack buffer," US Patent, Intel Corporation, US5964868, 1999.

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

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

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

[30] ——, "Return register stack target predictor," US Patent, Intel Corporation, US6560696, 2003.

[31] ——, "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.

[32] ——, "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.

[33] ——, "Attestation service for Intel software guard extensions (Intel SGX): API documentation," https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf, 2018.

[34] ——, "Intel analysis of speculative execution side channels," 2018, revision 1.0, January 2018.

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

[36] ——, "Intel SGX SDK," https://github.com/intel/linux-sgx, 2018.

[37] ——, "L1 terminal fault," 2018, https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

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

[39] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *36th IEEE Symposium on Security and Privacy*, May 2015.

[40] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," Intel, Tech. Rep, Tech. Rep., 2016.

[41] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, March 2003.

[42] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *21st USENIX Security Symposium*, 2012.

[43] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[44] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[45] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX WOOT*. USENIX Association, 2018.

[46] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *12th European Conference on Computer Systems*. ACM, 2017.

[47] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium*, 2017, pp. 557–574.

[48] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018.

[49] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *22nd IEEE Symposium on High Performance Computer Architecture*, 2016.

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

[51] F. Liu and R. B. Lee, "Random fill cache architecture," in *47th IEEE/ACM Symposium on Microarchitecture*, 2014.

[52] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," *ArXiv e-prints*, Jul. 2018.

[53] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *39th Annual International Symposium on Computer Architecture*, 2012.

[54] S. Matetic, K. Kostiainen, A. Dhar, D. Sommer, M. Ahmed, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution," Cryptology ePrint Archive, Report 2017/048, 2017.

[55] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *2nd HASP*, 2013.

[56] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "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.

[57] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *13th international conference on Selected areas in cryptography*, 2007, pp. 147–162.

[58] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *25th USENIX Security Symposium*, 2016.

[59] D. O'Keeffe, D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch, "Sgxspectre," https://github.com/lsds/spectre-attack-sgx, 2018.

[60] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *6th Cryptographers' track at the RSA conference on Topics in Cryptology*, 2006, pp. 1–20.

[61] C. Percival, "Cache missing for fun and profit," in *2005 BSDCan*, 2005.

[62] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium*, 2015.

[63] M. Russinovich, "Introducing azure confidential computing," 2017, https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[64] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *36th IEEE Symposium on Security and Privacy*, 2015.

[65] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," *CoRR*, vol. abs/1807.10535, 2018. [Online]. Available: http://arxiv.org/abs/1807.10535

[66] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. Springer International Publishing, 2017.

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

[68] H. Shacham, "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.

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

[70] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[71] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB linux applications with SGX enclaves," in *The Network and Distributed System Security Symposium*, 2017.

[72] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[73] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium*, 2016.

[74] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan, "The circle game: Scalable private membership test using trusted hardware," in *ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.

[75] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," Cryptology ePrint Archive, Report 2016/635, 2016.

[76] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 2, pp. 37–71, Jan. 2010.

[77] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX ATC*, 2017.

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

[79] D. Tychalas, N. G. Tsoutsos, and M. Maniatakos, "SGXCrypter: IP protection for portable executables using intel's SGX technology," in *22nd Asia and South Pacific Design Automation Conference*, 2017.

[80] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium*, 2018.

[81] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. New York, NY, USA: ACM, 2017, pp. 4:1–4:6.

[82] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.

[83] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-VM side-channels," in *23th USENIX Security Symposium*, 2014.

[84] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium*. Vancouver, BC: USENIX Association, 2017.

[85] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "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.

[86] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *22nd Annual Computer Security Applications Conference*, 2006.

[87] ——, "New cache designs for thwarting software cache-based side channel attacks," in *34th annual international symposium on Computer architecture*, 2007.

[88] S. Weiser and M. Werner, "SGXIO: Generic trusted i/o path for Intel SGX," arXiv preprint, arXiv:1701.01061, 2017.

[89] D. Woodhouse, "Retpoline patch for GCC," http://git.infradead.org/users/dwmw2/gcc-retpoline.git, 2018.

[90] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[91] Y. Yarom and K. E. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014, pp. 719–732.

[92] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," in *Cryptology ePrint Archive*, 2014.

[93] F. Zhang, E. Cecchetti, K. Croman, A. Juels, , and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[94] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM Conference on Computer and Communications Security*, 2012.

[95] ——, "Cross-tenant side-channel attacks in PaaS clouds," in *ACM Conference on Computer and Communications Security*, 2014.

[96] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX NSDI*, 2017.

[97] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *23rd ACM Conference on Computer and Communications Security*, 2016.