Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing

Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, and Hyesoon Kim,
Georgia Institute of Technology;* Marcus Peinado, *Microsoft Research

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

This paper is included in the Proceedings of the
26th USENIX Security Symposium
August 16–18, 2017 • Vancouver, BC, Canada
ISBN 978-1-931971-40-9

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing

Sangho Lee[†] Ming-Wei Shih[†] Prasun Gera[†] Taesoo Kim[†] Hyesoon Kim[†] Marcus Peinado[*]

[†] *Georgia Institute of Technology*
[*] *Microsoft Research*

Abstract

Intel has introduced a hardware-based trusted execution environment, Intel Software Guard Extensions (SGX), that provides a secure, isolated execution environment, or enclave, for a user program without trusting any underlying software (e.g., an operating system) or firmware. Researchers have demonstrated that SGX is vulnerable to a page-fault-based attack. However, the attack only reveals page-level memory accesses within an enclave.

In this paper, we explore a new, yet critical, side-channel attack, *branch shadowing*, that reveals fine-grained control flows (branch granularity) in an enclave. The root cause of this attack is that SGX does not clear branch history when switching from enclave to non-enclave mode, leaving fine-grained traces for the outside world to observe, which gives rise to a branch-prediction side channel. However, exploiting this channel in practice is challenging because 1) measuring branch execution time is too noisy for distinguishing fine-grained control-flow changes and 2) pausing an enclave right after it has executed the code block we target requires sophisticated control. To overcome these challenges, we develop two novel exploitation techniques: 1) a last branch record (LBR)-based history-inferring technique and 2) an advanced programmable interrupt controller (APIC)-based technique to control the execution of an enclave in a fine-grained manner. An evaluation against RSA shows that our attack infers each private key bit with 99.8% accuracy. Finally, we thoroughly study the feasibility of hardware-based solutions (i.e., branch history flushing) and propose a software-based approach that mitigates the attack.

1 Introduction

Establishing a trusted execution environment (TEE) is one of the most important security requirements, especially in a hostile computing platform such as a public cloud or a possibly compromised operating system (OS). When we want to run security-sensitive applications (e.g., processing financial or health data) in the public cloud, we need either to fully trust the operator, which is problematic [16], or encrypt all data before uploading them to the cloud and perform computations directly on the encrypted data. The latter can be based on fully homomorphic encryption, which is still slow [42], or on property-preserving encryption, which is weak [17, 38, 43]. Even when we use a private cloud or personal workstation, similar problems exist because no one can ensure that the underlying OS is robust against attacks given its huge code base and high complexity [2, 18, 23, 28, 36, 54]. Since the OS, in principle, is a part of the trusted computing base of a computing platform, by compromising it, an attacker can fully control any application running on the platform.

Industry has been actively proposing hardware-based techniques, such as the Trusted Platform Module (TPM) [56], ARM TrustZone [4], and Intel Software Guard Extension (SGX) [24], that support TEEs. Specifically, Intel SGX is receiving significant attention because of its recent availability and applicability. All Intel Skylake and Kaby Lake CPUs support Intel SGX, and processes secured by Intel SGX (i.e., processes running inside an *enclave*) can use almost every unprivileged CPU instruction without restrictions. To the extent that we can trust the hardware vendors (i.e., if no hardware backdoor exists [61]), it is believed that hardware-based TEEs are secure.

Unfortunately, recent studies [50, 60] show that Intel SGX has a noise-free side channel—a *controlled-channel attack*. SGX allows an OS to fully control the page table of an enclave process; that is, an OS can map or unmap arbitrary memory pages of the enclave. This ability enables a malicious OS to know exactly which memory pages a victim enclave attempts to access by monitoring page faults. Unlike previous side channels, such as cache-timing channels, the page-fault side channel is deterministic; that is, it has no measurement noise.

The controlled-channel attack has a limitation: It reveals only coarse-grained, page-level access patterns. Fur-

ther, researchers have recently proposed countermeasures against the attack such as balanced-execution-based design [50] and user-space page-fault detection [10, 49, 50]. However, these methods prevent only the page-level attack; hence, a fine-grained side-channel attack, if it exists, would easily bypass them.

We have thoroughly examined Intel SGX to determine whether it has a critical side channel that reveals fine-grained information (i.e., finer than page-level granularity) and is robust against noise. One key observation is that Intel SGX *leaves branch history uncleared during enclave mode switches*. Knowing the branch history (i.e., taken or not-taken branches) is critical because it reveals the fine-grained execution traces of a process in terms of basic blocks. To avoid such problems, Intel SGX hides all performance-related events (e.g., branch history and cache hit/miss) inside an enclave from hardware performance counters, including precise event-based sampling (PEBS), last branch record (LBR), and Intel Processor Trace (PT), which is known as anti side-channel interference (ASCI) [24]. Hence, an OS is unable to directly monitor and manipulate the branch history of enclave processes. However, since Intel SGX does not clear the branch history, an attacker who controls the OS can infer the fine-grained execution traces of the enclave through a branch-prediction side channel [3, 12, 13].

The branch-prediction side-channel attack aims to recognize whether the history of a targeted branch instruction is stored in a CPU-internal branch-prediction buffer, that is, a branch target buffer (BTB). The BTB is shared between an enclave and its underlying OS. Taking advantage of the fact that the BTB uses only the lowest 31 address bits (§2.2), the attacker can introduce set conflicts by positioning a shadow branch instruction that maps to the same BTB entry as a targeted branch instruction (§6.2). After that, the attacker can probe the shared BTB entry by executing the shadow branch instruction and determine whether the targeted branch instruction has been taken based on the execution time (§3). Several researchers exploited this side channel to infer cryptographic keys [3], create a covert channel [12], and break address space layout randomization (ASLR) [13].

This attack, however, is difficult to conduct in practice because of the following reasons. First, an attacker cannot easily guess the address of a branch instruction and manipulate the addresses of its branch targets because of ASLR. Second, since the capacity of a BTB is limited, entries can be easily overwritten by other branch instructions before an attacker probes them. Third, timing measurements of the branch misprediction penalty suffer from high levels of noise (§3.3). In summary, an attacker should have 1) a permission to freely access or manipulate the virtual address space, 2) access to the BTB anytime before it is overwritten, and 3) a method that recognizes branch misprediction with negligible (or no) noise.

In this paper, we present a new branch-prediction side-channel attack, *branch shadowing*, that accurately infers the fine-grained control flows of an enclave without noise (to identify conditional and indirect branches) or with negligible noise (to identify unconditional branches). A malicious OS can easily manipulate the virtual address space of an enclave process, so that it is easy to create shadowed branch instructions colliding with target branch instructions in an enclave. To minimize the measurement noise, we identify alternative approaches, including Intel PT and LBR, that are more precise than using `RDTSC` (§3.3). More important, we find that the LBR in a Skylake CPU allows us to obtain the most accurate information for branch shadowing because it reports whether each conditional or indirect branch instruction is correctly predicted or mispredicted. That is, we can *exactly* know the prediction and misprediction of conditional and indirect branches (§3.3, §3.5). Furthermore, the LBR in a Skylake CPU reports elapsed core cycles between LBR entry updates, which are very stable according to our measurements (§3.3). By using this information, we can precisely infer the execution of an unconditional branch (§3.4).

Precise execution control and frequent branch history probing are other important requirements for branch shadowing. To achieve these goals, we manipulate the frequency of the local advanced programmable interrupt controller (APIC) timer as frequently as possible and make the timer interrupt code perform branch shadowing. Further, we selectively disable the CPU cache when a more precise attack is needed (§3.6).

We evaluated branch shadowing against an RSA implementation in mbed TLS (§4). When attacking sliding-window RSA-1024 decryption, we successfully inferred each bit of an RSA private key with 99.8% accuracy. Further, the attack recovered 66% of the private key bits by running the decryption *only once*, unlike existing cache-timing attacks, which usually demand several hundreds to several tens of thousands of iterations [20, 35, 65].

Finally, we suggest hardware- and software-based countermeasures against branch shadowing that flush branch states during enclave mode switches and utilize indirect branches with multiple targets, respectively (§5).

The contributions of this paper are as follows:

- **Fine-grained attack.** We demonstrate that branch shadowing successfully identifies fine-grained control flow information inside an enclave in terms of basic blocks, unlike the state-of-the-art controlled-channel attack, which reveals only page-level accesses.
- **Precise attack.** We make branch shadowing very precise by 1) exploiting Intel PT and LBR to correctly identify branch history and 2) adjusting the

local APIC timer to precisely control the execution inside an enclave. We can deterministically know whether a target branch was taken without noise for conditional and indirect branches and with negligible noise for unconditional branches.

- **Countermeasures.** We design proof-of-concept hardware- and software-based countermeasures against the attack and evaluate them.

The remainder of this paper is organized as follows. §2 explains SGX and other CPU features our attack exploits. §3 and §4 describe our attack and evaluate it. §5 proposes our countermeasures. §6 discusses our attack's limitations and considers some advanced attacks. §7 introduces related work and §8 concludes this paper.

2 Background

We explain Intel SGX and two other processor features, branch prediction and LBR, closely related to our attack.

2.1 Intel SGX

An Intel CPU supports a hardware-based TEE through a security extension, Intel SGX. SGX provides a set of instructions to allow an application to instantiate an *enclave* that secures the code and data inside it against privileged software such as an OS or a hypervisor, hardware firmware, and even hardware units except for the CPU. To provide such protection, SGX enforces a strict memory access mechanism: allow only enclave code to access memory of the same enclave. In addition, SGX leverages an on-chip memory-encryption engine that encrypts enclave content before writing it into physical memory and decrypts the encrypted content only as it enters the CPU package during enclave execution or enclave mode.

Enclave context switch. To support context switching between enclave and non-enclave mode, SGX provides instructions such as `EENTER`, which starts enclave execution, and `EEXIT`, which terminates enclave execution. Also, `ERESUME` resumes enclave execution after an asynchronous enclave exit (AEX) occurs. The causes of an AEX include exceptions and interrupts. During a context switch, SGX conducts a series of checks and actions to ensure security, e.g., flushing the translation lookaside buffer (TLB). However, we observe that SGX does not clear all cached system state such as branch history (§3).

2.2 Branch Prediction

Branch prediction is one of the most important features of modern pipelined processors. At a high level, an instruction pipeline consists of four major stages: fetch, decode, execute, and write-back. At any given time, there are a number of instructions in-flight in the pipeline. Processors exploit instruction-level parallelism and out-of-order execution to maximize the throughput while still maintaining in-order retirement of instructions. Branch instructions can severely reduce instruction throughput since the processor cannot execute past the branch until the branch's target and outcome are determined. Unless mitigated, branches would lead to pipeline stalls, also known as bubbles. Hence, modern processors use a branch prediction unit (BPU) to *predict* branch outcomes and branch targets. While the BPU increases throughput in general, it is worth noting that in the case of a misprediction, there is a pretty high penalty because the processor needs to clear the pipeline and roll back any speculative execution results. This is why Intel provides a dedicated hardware feature (the LBR) to profile branch execution (§2.3).

Branch and branch target prediction. *Branch prediction* is a procedure to predict the next instruction of a conditional branch by guessing whether it will be taken. *Branch target prediction* is a procedure to predict and fetch the target instruction of a branch before executing it. For branch target prediction, modern processors have the BTB to store the computed target addresses of taken branch instructions and fetch them when the corresponding branch instructions are predicted as taken.

BTB structure and partial tag hit. The BTB is an associative structure that resembles a cache. Address bits are used to compute the set index and tag fields. The number of bits used for set index is determined by the size of the BTB. Unlike a cache that uses all the remaining address bits for the tag, the BTB uses a subset of the remaining bits for the tag (i.e., a partial tag). For example, in a 64-bit address space, if ADDR[11:0] is used for index, instead of using ADDR[63:12] for a tag, only a partial number of bits such as ADDR[31:12] is used as the tag. The reasons for this choice are as follows: First, compared to a data cache, the BTB's size is very small, and the overhead of complete tags can be very high. Second, the higher-order bits typically tend to be the same within a program. Third, unlike a cache, which needs to maintain an accurate microarchitectural state, the BTB is just a predictor. Even if a partial tag hit results in a false BTB hit, the correct target will be computed at the execution stage and the pipeline will roll back if the prediction is wrong (i.e., it affects only performance, not correctness.)

Static and dynamic branch prediction. *Static branch prediction* is a default rule for predicting the next instruction after a branch instruction when there is no history [25]. First, the processor predicts that a forward conditional branch—a conditional branch whose target address is higher than itself—will *not be taken*, which means the next instruction will be directly fetched (i.e., a fall-through path). Second, the processor predicts that a backward conditional branch—a conditional branch whose target address is lower than itself—will *be taken*; that is, the specified target will be fetched. Third, the processor predicts that an indirect branch will *not be taken*, similar to the forward conditional branch case. Fourth,

the processor predicts that an unconditional branch will *be taken*, similar to the backward conditional branch case. In contrast, when a branch has a history in the BTB, the processor will predict the next instruction according to the history. This procedure is known as *dynamic branch prediction*.

In this paper, we exploit these two conditional branch behaviors to infer the control flow of a victim process running inside Intel SGX (§3).

2.3 Last Branch Record

The LBR is a new feature in Intel CPUs that logs information about recently *taken* branches (i.e., omitting information about not-taken branches) without any performance degradation, as it is separated from the instruction pipeline [26, 32, 33]. In Skylake CPUs, the LBR stores the information of up to 32 recent branches, including the address of a branch instruction (from), the target address (to), whether the branch direction or branch target was mispredicted (it does not independently report these two mispredictions), and the elapsed core cycles between LBR entry updates (also known as the timed LBR). Without filtering, the LBR records all kinds of branches, including function calls, function returns, indirect branches, and conditional branches. Also, the LBR can selectively record branches taken in user space, kernel space, or both.

Since the LBR reveals detailed information of recently taken branches, an attacker may be able to know the fine-grained control flows of an enclave process if the attacker can directly use the LBR against it, though he or she still needs mechanisms to handle not-taken branches and the limited capacity of the LBR. Unfortunately for the attacker and fortunately for the victim, an enclave does not report its branch executions to the LBR unless it is in a debug mode [24] to prevent such an attack. However, in §3, we show how an attacker can *indirectly* use the LBR against an enclave process while handling not-taken branches and overcoming the LBR capacity limitation.

3 Branch Shadowing Attacks

We explain the branch shadowing attack, which can infer the fine-grained control flow information of an enclave. We first introduce our threat model and depict how we can attack three types of branches: conditional, unconditional, and indirect branches. Then, we describe our approach to synchronizing the victim and the attack code in terms of execution time and memory address space.

3.1 Threat Model

We explain our threat model, which is based on the original threat model of Intel SGX and the controlled-channel attack [60]: an attacker has compromised the operating system and exploits it to attack a target enclave program.

First, the attacker knows the possible control flows of a target enclave program (i.e., a sequence of branch instructions and their targets) by statically or dynamically analyzing its source code or binary. This is consistent with the important use case of running unmodified legacy code inside enclaves [5,6,51,57]. Unobservable code (e.g., self-modifying code and code from remote servers) is outside the scope of our attack. Also, the attacker can map the target enclave program into specific memory addresses to designate the locations of each branch instruction and its target address. Self-paging [22] and live re-randomization of address-space layout [15] inside an enclave are outside the scope of our attack.

Second, the attacker infers which portion of code the target enclave runs via observable events, e.g., calling functions outside an enclave and page faults. The attacker uses this information to synchronize the execution of the target code with the branch shadow code (§3.8).

Third, the attacker interrupts the execution of the target enclave as frequently as possible to run the branch shadow code. This can be done by manipulating a local APIC timer and/or disabling the CPU cache (§3.6).

Fourth, the attacker recognizes the shadow code's branch predictions and mispredictions by monitoring hardware performance counters (e.g., the LBR) or measuring branch misprediction penalty [3, 12, 13].

Last, the attacker prevents the target enclave from accessing a reliable, high-resolution time source to avoid the detection of attacks because of slowdown. Probing the target enclave for every interrupt or page fault slows the enclave down such that the attacker needs to hide it. SGX version 1 already satisfies such a requirement, as it disallows `RDTSC`. For SGX version 2 (not yet released), the attacker may need to manipulate model-specific registers (MSRs) to hook `RDTSC`. Although the target enclave could rely on an external time source, it is also unreliable because of the network delay and overhead. Further, the attacker can intentionally drop or delay such packets.

3.2 Overview

The branch shadowing attack aims to obtain the fine-grained control flow of an enclave program by 1) knowing whether a branch instruction has been taken and 2) inferring the target address of the taken branch. To achieve this goal, an attacker first needs to analyze the source code and/or binary of a victim enclave program to find all branches and their target addresses. Next, the attacker writes shadow code for a set of branches to probe their branch history, which is similar to Evtyushkin et al.'s attack using the BTB [13]. Since using the BTB and BPU alone suffers from significant noise, branch shadowing exploits the LBR, which allows the attacker to precisely identify the states of all branch types (§3.3, §3.4, §3.5). Because of the size limitations of the BTB, BPU, and LBR, the branch shadowing attack has to synchronize the execution of the victim code and the shadow code in terms of execution time and memory address space. We ma-

```
1  if (a != 0) {
2    ++b;
3    ...
4  }
5  else {
6    --b;
7    ...
8  }
9  a = b;
10 ...
```

(a) Victim code executed inside an enclave. According to the value of `a`, either `if`-block or `else`-block is executed.

```
1  ⋆ if (c != c) {
2      nop; // never executed
3      ...
4    }
5  ⋆ else {
6  ⋆   nop; // execution
7  ⋆   ...
8  ⋆ }
9  ⋆ nop;
10 ⋆ ...
```

(b) Shadow code aligned with **(a)**. The BPU predicts which block will be executed according to the branch history of **(a)**.

Figure 1: Shadow code (b) against a victim's conditional branch (a). The execution time (i.e., running [1, 5-10], marked with ⋆ in (b)) of the shadowing instance depends on the branching result (i.e., taken or not at [1] in (a)) of the victim instance.

nipulate the local APIC timer and the CPU cache (§3.6) to frequently interrupt an enclave process execution for synchronization, adjust virtual address space (§3.7), and run shadow code to find a function the enclave process is currently running or has just finished running (§3.8).

3.3 Conditional Branch Shadowing

We explain how an attacker can know whether a target conditional branch inside an enclave has been taken by shadowing its branch history. For a conditional branch, we focus on recognizing whether the branch prediction is correct because it reveals the result of a condition evaluation for `if` statement or loop. Note that, in this and later sections, we mainly focus on a *forward* conditional branch that will be predicted as not taken by a static branch prediction rule (§2.2). Attacking a backward conditional branch is basically the same such that we skip the explanation of it in this paper.

Inferring through timing (RDTSC). First, we explain how we can infer branch mispredictions with `RDTSC`. Figure 1 shows an example victim code and its shadow code. The victim code's execution depends on the value of `a`: if `a` is not zero, the branch will not be taken such that the `if`-block will be executed; otherwise, the branch will be taken such that the `else`-block will be executed. In contrast, we make the shadow code's branch always be taken (i.e., the `else`-block is always executed). Without the branch history, this branch is always mispredicted because of the static branch prediction rule (§2.2). To make a BTB entry collision [13], we align the lower 31 bits of the shadow code's address (both the branch instruction and its target address) with the address of the victim code.

When the victim code has been executed before the shadow code is executed, the branch prediction or misprediction of the shadow code depends on the execution of the victim code. If the conditional branch of the victim code has been taken, i.e., if `a` was zero, the BPU predicts that the shadow code will also take the conditional branch, which is a correct prediction so that no rollback will occur. If the conditional branch of the victim code either has not been taken, i.e., if `a` was not zero, or has not been executed, the BPU predicts that the shadow code will not take the conditional branch. However, this is an incorrect prediction such that a rollback will occur.

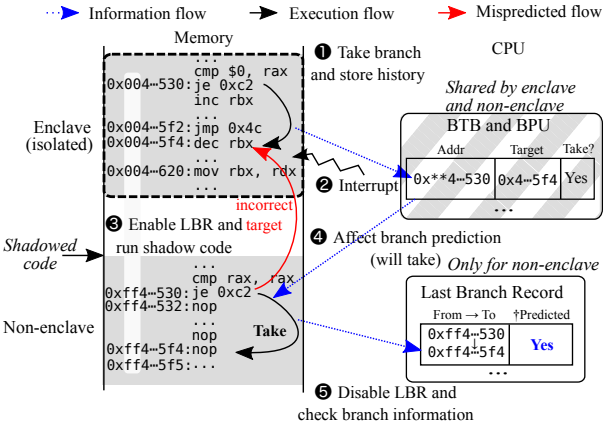
	Correct prediction		Misprediction	
	Mean	σ	**Mean**	σ
RDTSCP	94.21	13.10	120.61	**806.56**
Intel PT CYC packets	59.59	14.44	90.64	**191.48**
LBR elapsed cycle	25.69	9.72	35.04	**10.52**

Table 1: Measuring branch misprediction penalty with `RDTSCP`, Intel PT CYC packet, and LBR elapsed cycle (10,000 times). We put 120 `NOP` instructions at the fall-through path. The LBR elapsed cycle is less noisy than `RDTSCP` and Intel PT. σ stands for standard deviation.

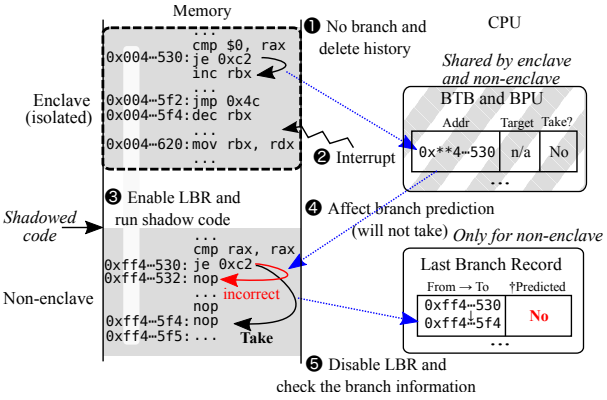
Previous branch-timing attacks try to measure such a rollback penalty with the `RDTSC` or `RDTSCP` instructions. However, our experiments show (Table 1) that branch misprediction timings are quite noisy. Thus, it was difficult to set a clear boundary between correct prediction and misprediction. This is because the number of instructions that would be mistakenly executed because of the branch misprediction is difficult to predict given the highly complicated internal structure of the latest Intel CPUs (e.g., out-of-order execution). Therefore, we think that the `RDTSC`-based inference is difficult to use in practice and thus we aim to use the LBR to realize precise attacks, since it lets us know branch misprediction information, and its elapsed cycle feature has little noise (Table 1).

Inferring from execution traces (Intel PT). Instead of using `RDTSC`, we can use Intel PT to measure a misprediction penalty of a target branch, as it provides precise elapsed cycles (known as a CYC packet) between each PT packet. However, CYC packets cannot be used immediately for our purpose because Intel PT aggregates a series of conditional and unconditional branches into a single packet as an optimization. To avoid this problem, we intentionally insert an indirect branch right after the target branch, making all branches properly record their elapsed time in separate CYC packets. Intel PT's timing information about branch misprediction has a much smaller variance than `RDTSCP`-based measurements (Table 1).

Precise leakage (LBR). Figure 2 shows a procedure for conditional branch shadowing with the BTB, BPU, and LBR. We first explain the case in which a conditional branch has been taken (Case 1). ❶ A conditional branch of the victim code is taken and the corresponding information is stored into the BTB and BPU. This branch taken occurs inside an enclave such that the LBR does not report this information unless we run the enclave process in a debug mode. ❷ Enclave execution is interrupted and



(a) Case 1: The target conditional branch has been taken.



(b) Case 2: The target conditional branch has not been taken (i.e., either not been executed or been executed but not taken).

Figure 2: Branch shadowing attack against a *conditional branch* (i.e., Case 1 for taken and Case 2 for non-taken branches) inside an enclave († LBR records the result of *misprediction*. For clarity, we use the result of *prediction* in this paper.)

the OS takes control. We explain how a malicious OS can frequently interrupt an enclave process in §3.6. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts that the shadowed conditional branch will be taken. At this point, a branch target prediction will fail because the BTB stores a target address inside an enclave. However, this target misprediction is orthogonal to the result of a branch prediction though it will introduce a penalty in CPU cycles (§3.4). ❺ Finally, by disabling and retrieving the LBR, we learn that the shadowed conditional branch has been *correctly predicted*—it has been taken as predicted. We think that this correct prediction is about branch prediction because the target addresses of the two branch instructions are different; that is, the target prediction might be failed. Note that, by default, the LBR reports all the branches (including function calls) that occurred in user and kernel space. Since our shadow code has no function calls and is executed in the kernel, we use the LBR's filtering mechanism to ignore every function call and all branches in user space.

Next, we explain the case in which a conditional branch has not been taken (Case 2). ❶ The conditional branch of the victim code is not taken, so either no information is stored into the BTB and BPU or the corresponding old information might be deleted (if there are conflict missed in the same BTB set.) ❷ Enclave execution is interrupted and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU incorrectly predicts that the shadowed conditional branch will not been taken, so the execution is rolled back to take the branch. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadowed conditional branch has been *mispredicted*—it has been taken unlike the branch prediction.

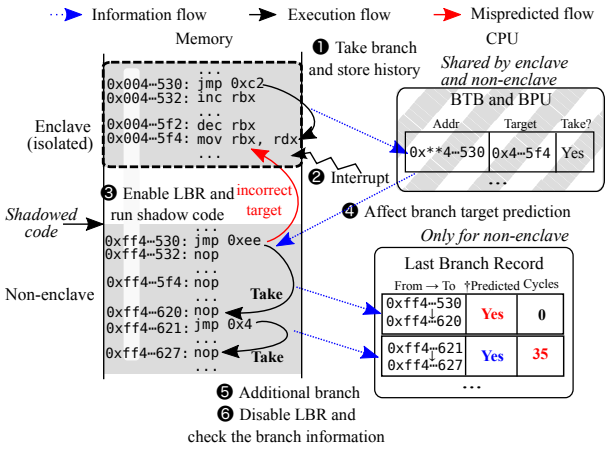
Initializing branch states. When predicting a conditional branch, modern BPUs exploit the branch's several previous executions to improve prediction accuracy. For example, if a branch had been taken several times and then not taken only once, a BPU would predict that its next execution would be taken. This would make the shadow branching infer incorrectly a target branch's execution after it has been executed multiple times (e.g., inside a loop). To solve this problem, after the final step of each attack iteration, we additionally run the shadow code multiple times while varying the condition (i.e., interleaving taken and not-taken branches) to initialize branch states.

3.4 Unconditional Branch Shadowing

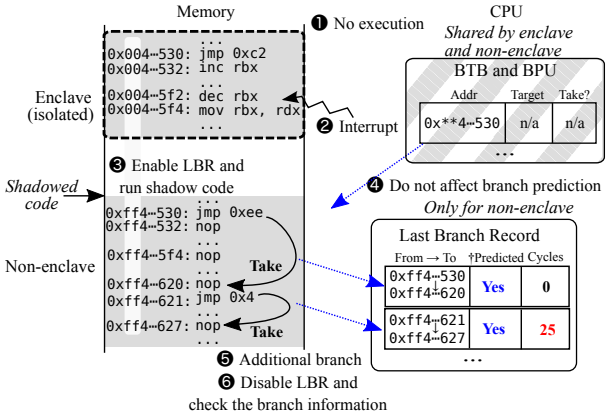
We explain how an attacker can know whether a target unconditional branch inside an enclave has been executed by shadowing its branch history. This gives us two kinds of information. First, an attacker can infer where the instruction pointer (IP) inside an enclave currently points. Second, an attacker can infer the result of the condition evaluation of an `if-else` statement because an `if` block's last instruction is an unconditional branch to skip the corresponding `else` block.

Unlike a conditional branch, an unconditional branch is always taken; i.e., a branch prediction is not needed. Thus, to recognize its behavior, we need to divert its target address to observe branch target mispredictions, not branch mispredictions. Interestingly, we found that the LBR does not report the branch target misprediction of an unconditional branch; it always says that each taken unconditional branch was correctly predicted. Thus, we use the elapsed cycles of a branch that the LBR reports to identify the branch target misprediction penalty, which is less noisy than `RDTSC` (Table 1).

Attack procedure. Figure 3 shows our procedure for unconditional branch shadowing. Unlike the conditional branch shadowing, we make the target of the shadowed unconditional branch differ from that of the victim uncon-



(a) Case 3: The target unconditional branch has been taken. The LBR does not report the misprediction of unconditional branches, but we can infer it by using the elapsed cycles.



(b) Case 4: The target unconditional branch has not been taken.

Figure 3: Branch shadowing attack against an unconditional branch inside an enclave.

ditional branch to recognize a branch target misprediction. We first explain the case in which an unconditional branch has been executed (Case 3). ❶ An unconditional branch of the victim code is executed and the corresponding information is stored into the BTB and BPU. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU mispredicts the branch target of the shadowed unconditional branch because of the mismatched branch history, so execution is rolled back to jump to the correct target. ❺ The shadow code executes an additional branch to measure the elapsed cycles of the mispredicted branch. ❻ Finally, by disabling and retrieving the LBR, we learn that a branch target misprediction occurred because of the large number of elapsed cycles.

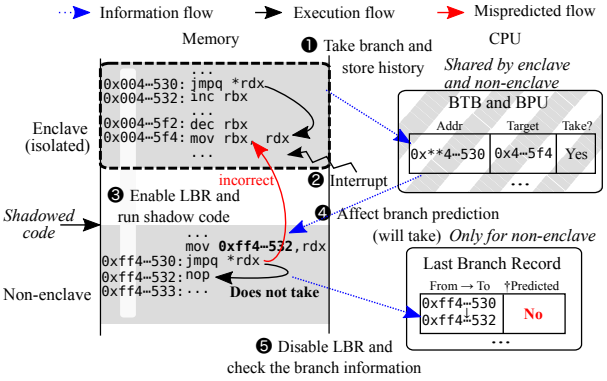
Next, we explain the case in which an unconditional branch has not been taken (Case 4). ❶ The enclave has not yet executed the unconditional branch in the victim code, so the BTB has no information about the branch. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts the shadowed unconditional branch's target, because the target unconditional branch has never been executed. ❺ The shadow code executes an additional branch to measure the elapsed cycles. ❻ By disabling and retrieving the LBR, we learn that no branch target misprediction occurred because of the small number of elapsed cycles.

No misprediction of unconditional branch. We found that the LBR always reports that every taken unconditional branch has been predicted irrespective of whether it mispredicted the target (undocumented behavior). We think that this is because the target of an unconditional branch is fixed such that, typically, target mispredictions should not occur. Also, the LBR was for facilitating branch profiling to reduce mispredictions for optimization. However, programmers have no way to handle mispredicted unconditional branches that result from the execution of the kernel or another process—i.e., it does not help programmers improve their program and just reveals side-channel information. We believe these are the reasons the LBR treats every unconditional branch as correctly predicted.

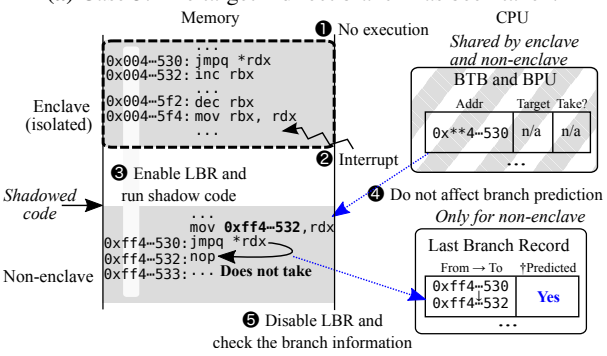
3.5 Indirect Branch Shadowing

We explain how we can infer whether a target indirect branch inside an enclave has been executed by shadowing its branch history. Like an unconditional branch, an indirect branch is always taken when it is executed. However, unlike an unconditional branch, an indirect branch has no fixed branch target. If there is no history, the BPU predicts that the instruction following the indirect branch instruction will be executed; this is the same as the indirect branch not being taken. To recognize its behavior, we make a shadowed indirect branch jump to the instruction immediately following it to monitor a branch target misprediction because of the history. The LBR reports the mispredictions of indirect branches such that we do not need to rely on elapsed cycles to attack indirect branches.

Attack procedure. Figure 4 shows a procedure of indirect branch shadowing. We make the shadowed indirect branch jump to its next instruction to observe whether a branch misprediction occurs because of the branch history. We first explain the case in which an indirect branch has been executed (Case 5). ❶ An indirect branch of the victim code is executed and the corresponding information is stored into the BTB and BPU. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU mispredicts that the shadowed indirect branch will be taken to an incorrect target address, so the execution is rolled back to not take the branch. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadow



(a) Case 5: The target indirect branch has been taken.



(b) Case 6: The target indirect branch has not been taken

Figure 4: Branch shadowing attack against an indirect branch inside an enclave.

code's indirect branch has been incorrectly predicted—it has not been taken, unlike the branch prediction.

Next, we explain the case in which an indirect branch has not been taken (Case 6). ❶ The enclave does not execute the indirect branch of the victim code, so that the BTB has no information about the branch. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts that the shadowed indirect branch will not be taken because there is no branch history. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadow code's indirect branch has been correctly predicted—it has not been taken, as predicted.

Inferring branch targets. Unlike conditional and unconditional branches, an indirect branch can have multiple targets such that just knowing whether it has been executed would be insufficient to know the victim code's execution. Since the indirect branch is mostly used for representing a `switch-case` statement, it is also related to a number of unconditional branches (i.e., `break`) as an `if-else` statement does. This implies that an attacker can identify which case block has been executed by probing the corresponding unconditional branch. Also, if an attacker can repeatedly execute a victim enclave program with the same input, he or she can test the same indirect

Branch	State	BTB/BPU	LBR		Inferred
			Pred.	Elapsed Cycl.	
Cond.	Taken	✓	✓	-	✓
	Not-taken	-	✓	-	✓
Uncond.	Exec.	✓	-	✓	✓
	Not-exec.	-	-	✓	✓
Indirect	Exec.	✓	✓	-	✓
	Not-exec.	-	✓	-	✓

Table 2: Branch types and states the branch shadowing attack can infer by using the information of BTB, BPU, and LBR.

branch multiple times while changing candidate target addresses to eventually know the real target address by observing a correct branch target prediction.

Table 2 summarizes the branch types and states our attack can infer and the necessary information.

3.6 Frequent Interrupting and Probing

The branch shadowing attack needs to consider cases that change (or even remove) BTB entries because they make the attack miss some branch histories. First, the size of the BTB is limited such that a BTB entry could be overwritten by another branch instruction. We empirically identified that the Skylake's BTB has 4,096 entries, where the number of ways is four and the number of sets is 1,024 (§5.1). Because of its well-designed index-hashing algorithm, we observed that conflicts between two branch instructions located at different addresses rarely occurred. But, no matter how, after more than 4,096 different branch instructions have been taken, the BTB will overflow and we will lose some branch histories. Second, a BTB entry for a conditional or an indirect branch can be removed or changed because of a loop or re-execution of the same function. For example, a conditional branch has been taken at its first run and has not been taken at its second run because of the changes of the given condition, removing the corresponding BTB entry. A target of an indirect branch can also be changed according to conditions, which change the corresponding BTB entry. If the branch shadowing attack could not check a BTB entry before it has been changed, it would lose the information.

To solve this problem, we interrupt the enclave process as frequently as possible and check the branch history by manipulating the local APIC timer and the CPU cache. These two approaches slow the execution of a target enclave program a lot such that an attacker needs to carefully use them (i.e., selectively) to avoid detection.

Manipulating the local APIC timer. We manipulate the frequency of the local APIC timer in a recent version of Linux (details are in Appendix A.) We measured the frequency of our manipulated timer interrupts in terms of how many `ADD` instructions can be executed between two timer interrupts. On average, about 48.76 `ADD` instructions were executed between two timer interrupts (standard

deviation: 2.75)[1]. ADD takes only one cycle in the Skylake CPU [25] such that our frequent timer can interrupt a victim enclave per every ∼50 cycles.

Disabling the cache. If we have to attack a branch instruction in a short loop taking < 50 cycles, the frequent timer interrupt is not enough. To interrupt an enclave process more frequently, we selectively disable the L1 and L2 cache of a CPU core running the victim enclave process by setting the cache disable (CD) bit of the CR0 control register. With the frequent timer interrupt and disabled cache, about 4.71 ADD instructions were executed between two timer interrupts on average (standard deviation: 1.96 with 10,000 iterations). Thus, the highest attack frequency we could achieve was around five cycles.

3.7 Virtual Address Manipulation

To perform the branch shadowing attack, an attacker has to manipulate the virtual addresses of a victim enclave process. Since the attacker has already compromised the OS, manipulating the page tables to change virtual addresses is an easy task. For simplicity, we assume the attacker disables the user-space ASLR and modifies the Intel SGX driver for Linux (vm_mmap) to change the base address of an enclave (Appendix B). Also, the attacker puts an arbitrary number of NOP instructions before the shadow code to satisfy the alignment.

3.8 Attack Synchronization

Although the branch shadowing probes multiple branches in each iteration, it is insufficient when a victim enclave program is large. An approach to overcome this limitation is to apply the branch shadowing attack at the function level. Namely, an attacker first infers functions a victim enclave program either has executed or is currently executing and then probes branches belonging to these functions. If these functions contain entry points that can be invoked from outside (via EENTER) or that rely on external calls, the attacker can easily identify them because they are controllable and observable by the OS.

However, the attacker needs another strategy to infer the execution of non-exported functions. The attacker can create special shadow code consisting of always reachable branches of target functions (e.g., branches located at the function prologue). By periodically executing this code, the attacker can see which of the monitored functions has been executed. Also, the attacker can use the page-fault side channel [60] to synchronize attacks in terms of pages.

3.9 Victim Isolation

To minimize noise, we need to ensure that only a victim enclave program and shadow code will be executed in an isolated physical core. Each physical core has the BTB and BPU shared by multiple processes. Thus, if another process runs in the core under the branch shadowing attack, its execution would affect the overall attack results. To avoid this problem, we use the `isolcpus` boot parameter to specify an isolated core that will not be scheduled without certain requests. Then, we use the `taskset` command to run a victim enclave with the isolated core.

[1]The number of iterations was 10,000. We disabled Hyper-Threading, SpeedStep, TurboBoost, and C-States to reduce noise.

```
1  /* Sliding-window exponentiation: X = A^E mod N */
2  int mbedtls_mpi_exp_mod(mbedtls_mpi *X, const mbedtls_mpi *A,
3              const mbedtls_mpi *E, const mbedtls_mpi *N,
4                  mbedtls_mpi *_RR) {
5    ...
6    state = 0;
7    while (1) {
8      ...
9      // i-th bit of exponent
10     ei = (E->p[nblimbs] >> bufsize) & 1;
11
12     // cmpq 0x0,-0xc68(%rbp); jne 3f317; ...
13 ⋆   if (ei == 0 && state == 0)
14       continue;
15
16     // cmpq 0x0,-0xc68(%rbp); jne 3f371; ...
17 ⋆   if (ei == 0 && state == 1)
18 +     mpi_montmul(X, X, N, mm, &T);
19
20     state = 2; nbits++;
21     wbits |= (ei << (wsize-nbits));
22
23     if (nbits == wsize) {
24       for (i = 0; i < wsize; i++)
25 +       mpi_montmul(X, X, N, mm, &T);
26
27 +     mpi_montmul(X, &W[wbits], N, mm, &T);
28       state--; nbits = wbits = 0;
29     }
30   }
31   ...
32 }
```

Figure 5: Sliding-window exponentiation of mbed TLS. Branch shadowing can infer every bit of the secret exponent.

4 Evaluation

In this section, we demonstrate the branch shadowing attack against an implementation of RSA and also describe our case studies of various libraries and applications that are vulnerable to our attack but mostly secure against the controlled-channel attack [60]. The branch shadowing attack's goal is not to overcome countermeasures against branch-prediction side-channel attacks, e.g., exponent blinding to hide an exponent value, not branch executions [34]. Thus, we do not try to attack applications without branch-prediction side channels.

4.1 Attacking RSA Exponentiation

We launch the branch shadowing attack against a popular TLS library, called mbed TLS (also known as PolarSSL). mbed TLS is a popular choice of SGX developers and researchers because of its lightweight implementation and portability [47, 49, 62, 63].

Figure 5 shows how mbed TLS implements sliding-window exponentiation, used by RSA operations. This function has two conditional branches (`jne`) marked with ⋆ whose executions depend on each bit (`ei`) of an exponent. These branches will be taken only when `ei` is not

zero (i.e., one). Thus, by shadowing them and checking their states, we can know the value of `ei`. Note that the two branches are always executed no matter how large the sliding window is. In our system, each loop execution (Lines 7–30) took about 800 cycles such that a manipulated local APIC timer was enough to interrupt it. Also, to differentiate each loop execution, we shadow unconditional branches that jump back to the loop's beginning.

We evaluated the accuracy of branch shadowing by attacking RSA-1024 decryption with the default key pair provided by mbed TLS for testing. By default, mbed TLS's RSA implementation uses the Chinese Remainder Theorem (CRT) technique to speed up computation. Thus, we observed two executions of `mbedtls_mpi_exp_mod` with two different 512-bit CRT exponents in each iteration. The sliding-window size was five.

On average, the branch shadowing attack recovered approximately 66% of the bits of each of the two CRT exponents from *a single run* of the victim (averaged over 1,000 executions). The remaining bits (34%) correspond to loop iterations in which the two shadowed branches returned different results (i.e., predicted versus mispredicted). We discarded those measurements, as they were impacted by platform noise, and marked the corresponding bits as unknown. The remaining 66% of the bits were inferred correctly with an accuracy of 99.8%, where the standard deviation was 0.003.

The events that cause the attack to miss about 34% of the key bits appear to occur at random times. Different runs reveal different subsets of the key bits. After at most 10 runs of the victim, the attack recovers virtually the entire key. This number of runs is small compared to existing cache-timing attacks, which demand several hundreds to several tens of thousands of runs to reliably recover keys [20, 35, 65].

Timing-based branch shadowing. Instead of using the LBR, we measured how long it takes to execute the shadow branches using `RDTSCP` while maintaining other techniques, including the modified local APIC timer and victim isolation. When the two target branches were taken, the shadow branches took 55.51 cycles on average, where the standard deviation was 48.21 cycles (1,000 iterations). When the two target branches were not taken, the shadow branches took 93.89 cycles on average, where the standard deviation was 188.49 cycles. Because of high variance, finding a good decision boundary was challenging, so we built a support vector machine classifier using LIBSVM (with an RBF kernel and default parameters). Its accuracy was 0.947 (10-fold cross validation)—i.e., we need to run this attack at least two times more than the LBR-based attack to achieve the same level of accuracy.

Controlled-channel attack. We also evaluated the controlled channel attack against Figure 5. We found that `mbedtls_mpi_exp_mod` conditionally called `mpi_montmul` (marked with +) according to the value of `ei` and both functions were located on different code pages. Thus, by carefully unmapping these pages, an attacker can monitor when `mpi_montmul` is called. However, as Figure 6 shows, because of the sliding-window technique, the controlled-channel attack cannot identify every bit unless it knows `W[wbits]`—i.e., this attack can only know the first bit of each window (always one) and skipped bits (always zero). The number of recognizable bits completely depends on how the bits of an exponent are distributed. Against the default RSA-1024 private key of mbed TLS, this attack identified 334 bits (32.6%). Thus, we conclude that the branch shadowing attack is better than the controlled-channel attack for obtaining fine-grained information.

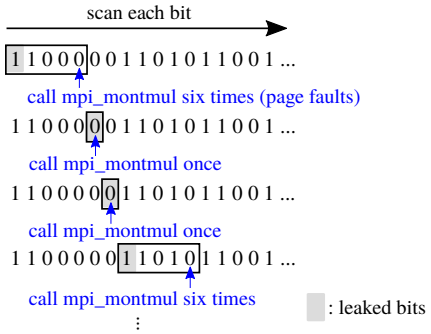


Figure 6: Controlled-channel attack against sliding-window exponentiation (window size: 5). It only knows the first bit of each window (always one) and skipped bits (always zero).

4.2 Case Study

We also studied other sensitive applications that branch shadowing can attack. Specifically, we focused on examples in which the controlled-channel attack cannot extract any information, e.g., control flows within a single page. We attacked three more applications: 1) two libc functions (`strtol` and `vfprintf`) in the Linux SGX SDK, 2) LibSVM, ported to Intel SGX, and 3) some Apache modules ported to Intel SGX. We achieved interesting results, such as how long an input number is (`strtol`), what the input format string looks like (`vfprintf`), and what kind of HTTP request an Apache server gets (`lookup_builtin_method`), as summarized in Table 3. Note that the controlled-channel attack cannot obtain the same information because those functions do not call outside functions at least in the target basic blocks. Detailed analysis with source code is in Appendix C.

5 Countermeasures

We introduce our hardware-based and software-based countermeasures against the branch shadowing attack.

5.1 Flushing Branch State

A fundamental countermeasure against the branch shadowing attack is to flush all branch states generated inside an enclave by modifying hardware or updating mi-

Program/Library	Function	Description	Obtainable information
mbed TLS	`mbedtls_mpi_exp_mod`	sliding-window exponentiation	✓each bit of an exponent
	`mpi_montmul`	Montgomery multiplication	✓whether a dummy subtraction has performed
libc	`strtol`	convert a string into an integer	✓the sign of an input number ✓the length of an input number ✓whether each hexadecimal digit is larger than nine
	`vfprintf`	print a formatted string	✓the input format string ✓the type of each input argument (e.g., int, double)
LIBSVM	`k_function`	evaluate a kernel function	✓the type of a kernel (e.g., linear, polynomial) ✓the length of a feature vector (i.e., # of features)
Apache	`lookup_builtin_method`	parse the method of an HTTP request	✓HTTP request method (e.g., `GET`, `POST`)

Table 3: Summary of example sensitive applications and their functions attacked by branch shadowing.

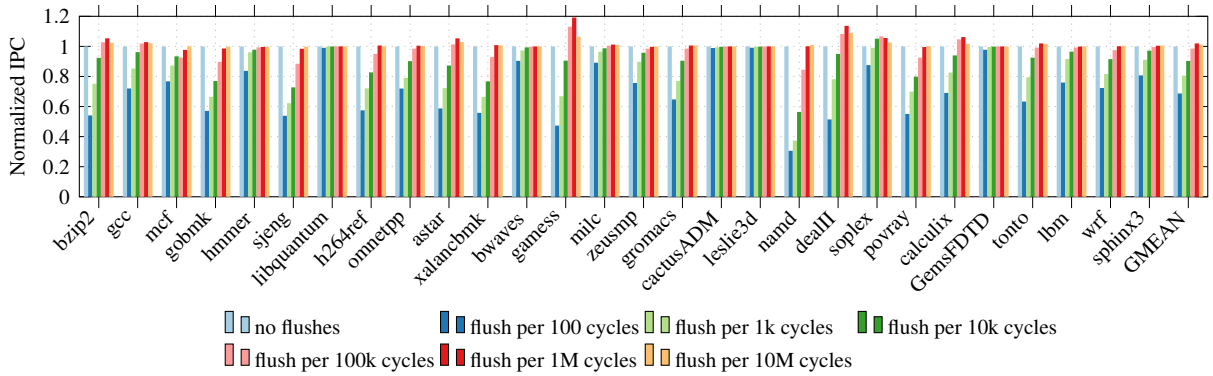


Figure 7: Instructions per cycle of SPEC benchmark in terms of frequency of BTB and BPU flushing.

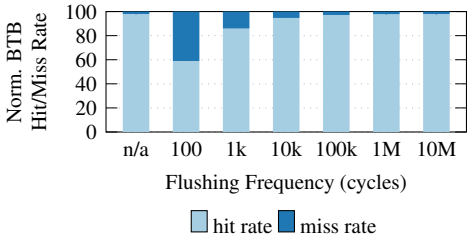


Figure 8: Average BTB hit/miss rate according to frequency of BTB and BPU flushing.

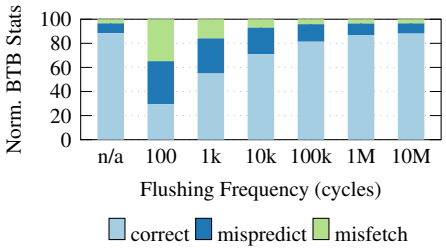


Figure 9: Average BTB statistics according to frequency of BTB and BPU flushing.

crocode. Whenever an enclave context switch (via the `EENTER`, `EEXIT`, or `ERESUME` instructions or AEX) occurs, the processor needs to flush the BTB and BPU states. Since the BTB and BPU benefit from local and global branch execution history, there would be a performance penalty if these states were flushed too frequently.

Parameter	Value
CPU	4 GHz out of order core, 4 issue width, 256 entry ROB
L1 cache	8 way 32 KB I-cache + 8 way 32 KB D-cache
L2 cache	8 way 128 KB
L3 cache	32 way 8 MB
BTB	4 way 1,024 sets
BPU	gshare, branch history length 16

Table 4: MacSim simulation parameters.

We estimate the performance overhead of our countermeasure at different enclave context switching frequencies using a cycle-level out-of-order microarchitecture simulator, MacSim [30]. To simulate branch history flushing for every enclave context switch, we modified MacSim to flush BTB and BPU for every 100 to 10 million cycles; this resembles enclave context switching for every 100 to 10 million cycles. The details of our simulation parameters are listed in Table 4. The BTB is modeled after the BTB in Intel Skylake processors. We used a method similar to that in [1, 58] to reverse engineer the BTB parameters. From our experiments, we found that the BTB is organized as a 4-way set associative structure with a total of 4,096 entries. We model a simple branch predictor, gshare [37], for the simulation. We use traces

that are 200 million instructions long from the SPEC06 benchmark suite for simulation.

Figure 7 shows the normalized instructions per cycle (IPC) for different flush frequencies. We found that if the flush frequency is higher than 100k cycles, it has negligible performance overhead. At a flush frequency of 100k cycles, the performance degradation is lower than 2% and at 1 million cycles, it is negligible. Figure 8 shows the BTB hit rate, whereas Figure 9 shows the BPU *correct*, *incorrect* (direction prediction is wrong), and *misfetch* (target prediction is wrong) percentages. The BTB and BPU statistics are also barely distinguishable beyond a flush frequency of 100k cycles.

According to our measurements with a 4GHz CPU, about 250 and 1,000 timer interrupts are generated per second in Linux (version 4.4) and Windows 10, respectively—i.e., a timer interrupt is generated for every 4M and 1M cycles, respectively. Therefore, if there is no I/O device generating many interrupts and an enclave program generates less frequent system calls, which would be desired to avoid the Iago attack [9], flushing branch states for every enclave context switch will introduce negligible overhead.

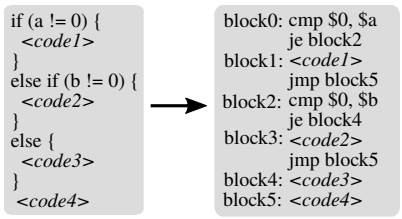
5.2 Obfuscating Branch

Branch state flushing can effectively prevent the branch shadowing attack, but we cannot be sure when and whether such hardware changes will be realized. Especially, if such changes cannot be done with micro code updates, we cannot protect the Intel CPUs already deployed in the markets.

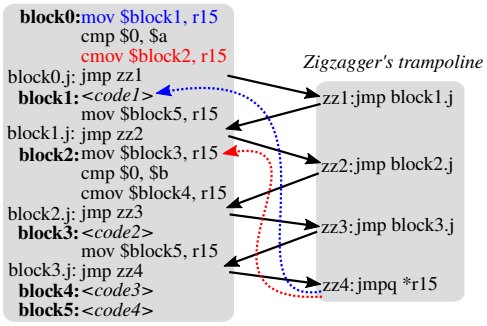
Possible software-based countermeasures against the branch shadowing attack are to remove branches [39] or to use the state-of-the-art ORAM technique, Raccoon [44]. Data-oblivious machine learning algorithms et al. [39] eliminate all branches by using a conditional move instruction, `CMOV`. However, their approach is algorithm-specific, i.e., it is not applicable to general applications. Raccoon [44] always executes both paths of a conditional branch, such that no branch history will be leaked. But, its performance overhead is high (21.8×).

Zigzagger. We propose a practical, compiler-based mitigation against branch shadowing, called *Zigzagger*. It obfuscates a set of branch instructions into a single indirect branch, as inferring the state of an indirect branch is more difficult than inferring those of conditional and unconditional branches (§3.5). However, it is not straightforward to compute the target block of each branch without relying on conditional jumps because conditional expressions could become complex because of nested branches. In Zigzagger, we solved this problem by using a `CMOV` instruction [39, 44] and introducing a sequence of non-conditional jump instructions in lieu of each branch.

Figure 10 shows how Zigzagger transforms an example code snippet having `if`, `else-if`, and `else` blocks. It



(a) An example code snippet. It selectively executes a branch block according to `a` and `b` variables.



(b) The protected code snippet by Zigzagger. All branch instructions are executed regardless of `a` and `b` variables. An indirect branch in the trampoline and `CMOV`s in the translated code are used to obfuscate the final target address. `r15` is reserved to store the target address.

Figure 10: An example of Zigzagger transformation.

converts all conditional and unconditional branches into unconditional branches targeting Zigzagger's trampoline, which jumps back-and-forth with the converted branches. The trampoline finally jumps into the real target address stored in a reserved register `r15`. Note that reserving a register is only for improving performance. We can use the memory to store the target address when an application needs to use a large number of registers. To emulate conditional execution, the `CMOV` instructions in Figure 10b update the target address in `r15` only when `a` or `b` is zero. Otherwise, they are treated as `NOP` instructions. Since all of the unconditional branches are executed almost simultaneously in sequence, recognizing the current instruction pointer is difficult. Further, since the trampoline now has five different target addresses, inferring real targets among them is not straightforward.

Zigzagger's approach has several benefits: 1) security: it provides the first line of protection on each branch block in an enclave program; 2) performance: its overhead is at most 2.19× (Table 5); 3) practicality: its transformation demands neither complex analysis of code semantics nor heavy code changes. However, it does not ensure perfect security such that we still need ORAM-like techniques to protect very sensitive functions.

Implementation. We implemented Zigzagger in LLVM 4.0 as an LLVM pass that converts branches in each function and constructs the required trampoline. We also mod-

Benchmark	Baseline (iter/s)	Zigzagger #Branches (overhead)				
		2	3	4	5	All
numeric sort	967.25	1.05×	1.11×	1.12×	1.13×	1.15×
string sort	682.31	1.08×	1.15×	1.18×	1.15×	1.27×
bitfield	4.5E+08	1.03×	1.10×	1.14×	1.18×	1.31×
fp emulation	96.204	1.10×	1.21×	1.15×	1.27×	1.35×
fourier	54982	0.99×	0.99×	1.01×	1.01×	1.01×
assignment	35.73	1.36×	1.56×	1.50×	1.55×	1.90×
idea	10,378	2.16×	2.16×	2.18×	2.19×	2.19×
huffman	2478.1	1.59×	1.46×	1.61×	1.63×	1.81×
neural net	16.554	0.75×	0.77×	0.85×	0.86×	0.89×
lu decomposition	1,130	1.04×	1.09×	1.08×	1.11×	1.17×
GEOMEAN		1.17×	1.22×	1.24×	1.26×	1.34×

Table 5: Overhead of the Zigzagger approach according to the number of branches belonging to each Zigzagger.

ified the LLVM backend to reserve a register. The number of branches a single trampoline manages affects the overall performance, so our implementation provides a knob to configure it to trade the security for performance.

Our proof-of-concept implementation of Zigzagger, merging every branch in each function, imposed a 1.34× performance overhead when evaluating it with the nbench benchmark suite (Table 5). With optimization (i.e., merging ≤ 3 branches into a single trampoline), the average overhead became $\leq 1.22\times$. Note that reserving a register resulted in a 4%–50% performance improvement.

6 Discussion

In this section, we explain some limitations of the branch shadowing attack and discuss possible advanced attacks.

6.1 Limitations

The branch shadowing attack has limitations. First, it cannot distinguish a not-taken conditional branch from a not-executed conditional branch because, in both cases, the BTB stores no information; the static branch prediction rule is applied. Second, it cannot distinguish an indirect branch to the next instruction from a not-executed indirect branch because their predicted branch targets are the same. Therefore, an attacker has to probe a number of correlated branches (e.g., unconditional branches in `else-if` or `case` blocks) to overcome these limitations. Third, as with the controlled-channel attack, the branch shadowing attack needs repetitions to increase attack accuracy, which can be prohibited by a state continuity solution [55]. However, this requires persistence storage such as that provided by a trusted platform module (TPM).

6.2 Advanced Attacks

We consider how branch shadowing can be improved: hyperthreading and blind approaches.

Hyperthreaded branch shadowing. Since two hyperthreads simultaneously running in the same physical core share the BTB and BPU, a malicious hyperthread can attack a victim enclave hyperthread by using BTB entry conflicts if a malicious OS gives the address information of the victim to it. We observed that branch instructions with the same low 16-bit address were mapped into the same BTB set. Thus, a malicious hyperthread can monitor a BTB set for evictions by filling the BTB set with four branch instructions (§5.1). The BTB flushing cannot prevent this attack because it demands no enclave mode switch, so disabling hyperthreading or preventing the hyperthreads from sharing the BTB and BPU is necessary.

Blind branch shadowing. A blind branch shadowing attack is an attempt to probe the entire or selected memory region of a victim enclave process to detect any unknown branch instructions. This attack would be necessary if a victim enclave process has self-modifying code or uses remote code loading, though this is outside the scope of our threat model (§3.1). In the case of unconditional branches, blind probing is easy and effective because it does not need to infer target addresses. However, in the case of conditional and indirect branches, blind probing needs to consider branch instructions and their targets simultaneously such that the search space would be huge. We plan to consider an effective method to minimize the search space to know whether this attack is practical.

7 Related Work

Intel SGX. The strong security guarantee provided by SGX has drawn significant attention from the research community. Several security applications of SGX are proposed, including secure and distributed data analysis [7, 11, 39, 46, 66] and secure networking service [31, 41, 48]. Also, researchers implemented SGX layers [5, 6, 51, 57] to run existing applications inside an enclave without any modifications. The security properties of SGX itself are also being intensively studied. For example, Sinha et al. [52, 53] develop tools to verify the confidentiality of enclave programs.

However, researchers find security attacks against Intel SGX. Xu et al. [60] and Shinde et al. [50] demonstrate the first side-channel attack on SGX by leveraging the fact that SGX relies on an OS for memory resource management. The attack is done by intentionally manipulating the page table to trigger a page fault and using a page-fault sequence to infer the secret inside an enclave. Weichbrodt et al. [59] also show how a synchronous bug can be exploited to attack SGX applications. Further, concurrently with our work, Hähnel et al. [21] exploit a frequent timer in Windows to realize a precise cache side-channel attack against the Intel SGX simulator.

To address the page-fault-based side-channel attack, Shinde et al. [50] obfuscate the memory access pattern of an enclave. Shih et al. [49] propose a compiler-based solution using Intel TSX to detect suspicious page faults inside an enclave. Also, Costan et al. [10] propose a new en-

clave design to prevent both page-fault and cache-timing side-channel attacks. Finally, Seo et al. [47] enforce fine-grained ASLR on enclave programs, which can raise the bar of exploiting any vulnerabilities and inferring control flow with page-fault sequences. However, all of these solutions heavily use branch instructions and do not clear branch states, such that they would be vulnerable to our attack.

Microarchitectural side channel. Researchers considered the security problems of microarchitectural side channels. The most popular and well-studied microarchitectural side channel is a CPU cache timing channel first developed by [29, 34, 40] to break cryptosystems. This attack is further extended to be conducted in the public cloud setting to recognize co-residency of virtual machines [45, 64]. Several researchers further improved this attack to exploit the last level cache [27, 35] and create a low-noise cache storage channel [19]. The CPU cache is not the sole source of the microarchitectural side channel. For example, to break kernel ASLR, researchers exploit a TLB timing channel [23], an Intel TSX instruction [28], a `PREFETCH` instruction [18], and a BTB timing channel [13]. Ge et al. [14] conducted a comprehensive survey of microarchitectural side channels.

8 Conclusion

A hardware-based TEE such as Intel SGX demands thorough analysis to ensure its security against hostile environments. In this paper, we presented and evaluated the branch shadowing attack, which identifies fine-grained execution flows inside an SGX enclave. We also proposed hardware-based countermeasure that clears the branch history during enclave mode switches and software-based mitigation that makes branch executions oblivious.

Responsible disclosure. We reported our attack to Intel and discussed with them to find effective solutions against it. Also, after having a discussion with us, the authors of Sanctum [10] revised their eprint paper that coped with our attack.

Acknowledgments. We thank the anonymous reviewers for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] The BTB in contemporary Intel chips—Matt Godbolt's blog. http://xania.org/201602/bpu-part-three. (Accessed on 11/10/2016).

[2] Kernel self protection project - Linux kernel security subsystem. https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project.

[3] Aciicmez, O., Koc, K., and Seifert, J. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2007).

[4] ARM. ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone.

[5] Arnautox, S., Tarch, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016).

[6] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[7] Brenner, S., Wulf, C., Lorenz, M., Weichbrodt, N., Goltzsche, D., Fetzer, C., Pietzuch, P., and Kapitza, R. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2016).

[8] Brumley, D., and Boneh, D. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2003).

[9] Checkoway, S., and Shacham, H. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013).

[10] Costan, V., Lebedev, I., and Devadas, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[11] Dinh, T. T. A., Saxena, P., Cang, E.-C., Ooi, B. C., and Zhang, C. M2R: Enabling stronger privacy in MapReduce computation. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[12] Evtyushkin, D., Ponomarev, D., and Abu-Ghazalch, N. Covert channels through branch predictors: A feasibility study. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2015).

[13] Evtyushkin, D., Ponomarev, D., and Abu-Ghazalch, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Taipei, Taiwan, Oct. 2016).

[14] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Cryptology ePrint Archive, Report 2016/613, 2016. http://eprint.iacr.org/2016/613.pdf.

[15] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium (Security)* (Bellevue, WA, Aug. 2012).

[16] Grance, T., and Jansen, W. Guidelines on security and privacy in public cloud computing. https://www.nist.gov/node/591971.

[17] Grubbs, P., McPherson, R., Naveed, M., Ristenpart, T., and Shmatikov, V. Breaking web applications built on top of encrypted data. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[18] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[19] GUANCIALE, R., NEMATI, H., BAUMANN, C., AND DAM, M. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[20] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games–bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2011).

[21] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[22] HAND, S. M. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (New Orleans, LA, Feb. 1999).

[23] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2013).

[24] INTEL. Intel software guard extensions programming reference (rev2), Oct. 2014. 329298-002US.

[25] INTEL. Intel 64 and IA-32 architectures optimization reference manual, June 2016.

[26] INTEL. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c and 3d, Sept. 2016.

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

[28] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[29] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side channel cryptanalysis of product ciphers. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS)* (Belgium, Sept. 1998).

[30] KIM, H., LEE, J., LAKSHMINARAYANA, N. B., SIM, J., LIM, J., AND PHO, T. MacSim: A CPU-GPU heterogeneous simulation framework.

[31] KIM, S., HAN, J., HA, J., KIM, T., AND HAN, D. Enhancing Security and Privacy of Tor's Ecosystem by using Trusted Execution Environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2017).

[32] KLEEN, A. Advanced usage of last branch records, 2016. https://lwn.net/Articles/680996/.

[33] KLEEN, A. An introduction to last branch records, 2016. https://lwn.net/Articles/680985/.

[34] KOCHER, P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96* (1996), Springer, pp. 104–113.

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

[36] LU, K., SONG, C., KIM, T., AND LEE, W. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[37] MCFARLING, S. Combining branch predictors. *Technical Report TN-36, Digital Western Research Laboratory* (1993).

[38] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (Denver, Colorado, Oct. 2015).

[39] OHRIMENKO, O., MANUEL COSTA, C. F., NOWOZIN, S., MEHTA, A., SCHUSTER, F., AND VASWANI, K. SGX-enabled oblivious machine learning. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[40] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive* (2002).

[41] PIRES, R., PASIN, M., FELBER, P., AND FETZER, C. Secure content-based routing using Intel Software Guard Extensions. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2016).

[42] POPA, R. A. *Building Practical Systems That Compute on Encrypted Data*. PhD thesis, Massachusetts Institute of Technology, 2014.

[43] POULIOT, D., AND WRIGHT, C. V. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[44] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[45] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Chicago, IL, Nov. 2009).

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

[47] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[48] SHIH, M.-W., KUMAR, M., KIM, T., AND GAVRILOVSKA, A. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV* (New Orleans, LA, Mar. 2016).

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

[50] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing your faults from telling your secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and*

Communications Security (ASIACCS) (Xi'an, China, May–June 2016).

[51] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXENA, P. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[52] SINHA, R., COSTA, M., LAL, A., LOPES, N. P., RAJAMANI, S., SESHIA, S. A., AND VASWANI, K. A design and verification methodology for secure isolated regions. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Santa Barbara, CA, June 2016).

[53] SINHA, R., RAJAMANI, S., SESHIA, S., AND VASWANI, K. Moat: Verifying confidentiality of enclave program. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (Denver, Colorado, Oct. 2015).

[54] SONG, C., LEE, B., LU, K., HARRIS, W. R., KIM, T., AND LEE, W. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[55] STRACKX, R., AND PIESSENS, F. Ariadne: A minimal approach to state continuity. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[56] TRUSTED COMPUTING GROUP. Trusted platform module (TPM) summary. http://www.trustedcomputinggroup.org/trusted-platform-module-tpm-summary/.

[57] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[58] UZELAC, V., AND MILENKOVIC, A. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 207–217.

[59] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)* (Heraklion, Greece, Sept. 2016).

[60] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[61] YANG, K., HICKS, M., DONG, Q., AUSTIN, T., AND SYLVESTER, D. A2: Analog malicious hardware. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[62] ZHANG, F. mbedtls-SGX: A SGX-friendly TLS stack (ported from mbedtls). https://github.com/bl4ck5un/mbedtls-SGX.

[63] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An authenticated data feed for smart contracts. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[64] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2011).

[65] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (Raleigh, NC, Oct. 2012).

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

A Manipulating Local APIC Timer

The local APIC is a component of Intel CPUs to configure and handle CPU-specific interrupts [26, §10]. An OS can program it through memory-mapped registers (e.g., device configuration register) or model-specific registers (MSRs) to adjust the frequency of the *local APIC timer*, which generates high-resolution timer interrupts, and deliver an interrupt to a CPU core (e.g., inter-processor interrupt (IPI) and I/O interrupt from the I/O APIC).

Intel CPUs support three local APIC timer modes: periodic, one-shot, and timestamp counter (TSC)-deadline modes. The periodic mode lets an OS configure the initial-count register whose value is copied into the current-count register the local APIC timer uses. The current-count register's value decreases at the rate of the bus frequency, and when it becomes zero, a timer interrupt is generated and the register is re-initialized by using the initial-count register. The one-shot mode lets an OS configure the initial-count counter value whenever a timer interrupt is generated. The TSC-deadline mode is the most advanced and precise timer mode allowing an OS to specify when the next timer interrupt will occur in terms of a TSC value. Our target Linux system (kernel version 4.4) uses the TSC-deadline mode, so we focus on this mode.

Figure 11 shows how we modified the `lapic_next_deadline()` function specifying the next TSC deadline and the `local_apic_timer_interrupt()` function called whenever a timer interrupt is fired. We made and exported two global variables and function pointers to manipulate the behaviors of `lapic_next_deadline()` and `local_apic_timer_interrupt()` with a kernel module: `lapic_next_deadline_delta` to change the delta; `lapic_target_cpu` to specify a virtual CPU running a victim enclave process (via a CPU affinity); and `timer_interrupt_hook` to specify a function to be called whenever a timer interrupt is generated. In our evaluation environment having an Intel Core i7 6700K CPU (4GHz), we were able to have 1,000 as the minimum delta value; i.e., it fires a timer interrupt about every 1,000 cycles. Note that, in our environment, a delta value lower than 1,000 made the entire system freeze because a timer interrupt was generated before an old timer interrupt was handled by the interrupt handler.

B Modifying SGX Driver

Figure 12 shows how we modified the Intel SGX driver for Linux to manipulate the base address of an enclave.

```
/* linux-4.4.23/arch/x86/kernel/apic/apic.c */
...
// manipualte the delta of TSC-deadline mode
unsigned int lapic_next_deadline_delta = 0U;
EXPORT_SYMBOL_GPL(lapic_next_deadline_delta);

// specify the virtual core under attack
int lapic_target_cpu = -1;
EXPORT_SYMBOL_GPL(lapic_target_cpu);

// a hook to launch branch shadowing attack
void (*timer_interrupt_hook)(void*) = NULL;
EXPORT_SYMBOL_GPL(timer_interrupt_hook);
...
// update the next TSC deadline
static int lapic_next_deadline(unsigned long delta,
                   struct clock_event_device *evt) {
  u64 tsc;
  tsc = rdtsc();
* if (smp_processor_id() != lapic_target_cpu)
    wrmsrl(MSR_IA32_TSC_DEADLINE,
        tsc + (((u64) delta) * TSC_DIVISOR));
* else
*   wrmsrl(MSR_IA32_TSC_DEADLINE,
*       tsc + lapic_next_deadline_delta); // custom deadline
  return 0;
}
...
// handle a timer interrupt
static void local_apic_timer_interrupt(void) {
  int cpu = smp_processor_id();
  struct clock_event_device *evt = &per_cpu(lapic_events, cpu);

* if (cpu == lapic_target_cpu && timer_interrupt_hook)
*   timer_interrupt_hook((void*)&cpu); // call attack code
  ...
}
```

Figure 11: Modified local APIC timer code. We changed `lapic_next_deadline()` to manipulate the next TSC deadline and `local_apic_timer_interrupt()` to launch attack code.

```
/* isgx_ioctl.c */
...
static long isgx_ioctl_enclave_create(struct file *filep,
                      unsigned int cmd, unsigned long arg) {
  ...
  struct isgx_create_param *createp =
    (struct isgx_create_param *) arg;
  void *secs_la = createp->secs;
  struct isgx_secs *secs = NULL;
  // SGX Enclave Control Structure (SECS)
  long ret;
  ...
  secs = kzalloc(sizeof(*secs), GFP_KERNEL);
  ret = copy_from_user((void *)secs, secs_la, sizeof (*secs));
  ...
* secs->base = vm_mmap(file, MANIPULATED_BASE_ADDR, secs->size,
*              PROT_READ | PROT_WRITE | PROT_EXEC,
*              MAP_SHARED, 0);
  ...
}
```

Figure 12: Modified Intel SGX driver to manipulate the base address of an enclave

C Case Study in Detail

We study other sensitive applications the branch shadowing can attack. Specifically, we focus on examples in which the controlled-channel attack cannot extract any information, e.g., control flows within a single page.

mbed TLS. We checked mbed TLS's another function: the Montgomery multiplication (`mpi_montmul`). As shown

```
/* bignum.c */
static int mpi_montmul(mbedtls_mpi *A, const mbedtls_mpi *B,
               const mbedtls_mpi *N, mbedtls_mpi_uint mm,
               const mbedtls_mpi *T) {
  size_t i, n, m;
  mbedtls_mpi_uint u0, u1, *d;

  d = T->p; n = N->n; m = (B->n < n) ? B->n : n;

  for (i = 0; i < n; i++) {
    u0 = A->p[i];
    u1 = (d[0] + u0 * B->p[0]) * mm;

    mpi_mul_hlp(m, B->p, d, u0);
    mpi_mul_hlp(n, N->p, d, u1);

    *d++ = u0; d[n+1] = 0;
  }

* if (mbedtls_mpi_cmp_abs(A, N) >= 0) {
*   mpi_sub_hlp(n, N->p, A->p);
*   i = 1;
* }
* else { // dummy subtraction to prevent timing attacks
*   mpi_sub_hlp(n, N->p, T->p);
*   i = 0;
* }
  return 0;
}
```

Figure 13: Montgomery multiplication (`mpi_montmul()`) of mbed TLS. The branch shadowing attack can infer whether a dummy subtraction has performed or not.

in Figure 13, this function has a dummy subtraction (Lines 24–27) to prevent the well-known remote timing attack [8]. The branch shadowing attack was able to detect the execution of this dummy branch. In contrast, the controlled-channel cannot know whether a dummy subtraction has happened because both real and dummy branches execute the same function: `mpi_sub_hlp()`.

Linux SGX SDK. We attacked two libc functions, `strtol()` and `vfprint()`, Linux SGX SDK provides. Figure 14a shows `strtol()` converting a string into an integer. The branch shadowing can infer the sign of an input number by checking the branches in Lines 7–12. Also, it infers the length of an input number by checking the loop branch in Lines 14–24. When an input number was hexadecimal, we were able to use the branch at Line 16 to know whether each digit was larger than nine.

Figure 14b shows `vfprintf()` printing a formatted string. The branch shadowing was able to infer the format string by checking the `switch-case` statement in Lines 4–13 and the types of input arguments to this function according the `switch-case` statement in Lines 15–23. In contrast, the controlled-channel attack cannot infer this information because the functions called by `vfprint()`, including `ADDSARG()` and `va_arg()`, are inline functions. No page fault sequence will be observed.

LIBSVM. LIBSVM is a popular library supporting support vector machine (SVM) classifiers. We ported a classification logic of LIBSVM to Intel SGX because it would be a good example of machine learning as a service [39]

```
1  /* linux-sgx/sdk/tlibc/stdlib/strtol.c */
2  long strtol(const char *nptr, char **endptr, int base) {
3    ...
4    s = nptr;
5    do { c = (unsigned char) *s++; } while (isspace(c));
6
7  ★ if (c == '-') {
8  ★   neg = 1; c = *s++;
9  ★ } else {
10 ★   neg = 0;
11 ★   if (c == '+') c = *s++;
12 ★ } // infer the sign of an input number
13
14 ★ for (acc = 0, any = 0;; c = (unsigned char) *s++) {
15 ★   if(isdigit(c)) c -= '0';
16 ★   else if (isalpha(c)) c -= isupper(c) ? 'A'-10 : 'a'-10;
17 ★   //  infer hexademical
18     else break;
19
20     if (!neg) {
21       acc *= base; acc += c;
22     }
23     ...
24 ★ } // infer the length of an input number
25   ...
26 }
```

(a) Simplified `strtol()`. The branch shadowing attack can infer the sign and length of an input number.

```
1  /* linux-sgx/sdk/tlibc/stdio/vfprintf.c */
2  int __vfprintf(FILE *fp, const char *fmt0, __va_list ap) {
3    ...
4    for (;;) {
5      ch = *fmt++;
6      switch (ch) {
7      ...
8  ★   case 'd': case 'i': ADDSARG(); break;
9  ★   case 'p': ADDTYPE_CHECK(TP_VOID); break;
10 ★   case 'X': case 'x': ADDUARG(); break;
11     ...
12     }
13   } // infer input format string
14   ...
15   for (n = 1; n <= tablemax; n++) {
16     switch (tyypetable[n]) {
17 ★   case T_INT:
18 ★     (*argtable)[n].intarg = va_arg(ap, int); break;
19 ★   case T_DOUBLE:
20 ★     (*argtable)[n].doublearg = va_arg(ap, double); break;
21     ...
22     }
23 ★ } // infer the types of input arguments
24   ...
25 }
```

(b) Simplified `vfprintf()`. The branch shadowing attack can infer the format string and variable arguments.

Figure 14: libc functions attacked by the branch shadowing

while hiding the detailed parameters. Figure 15 shows the LIBSVM's kernel function code running inside an enclave. The branch shadowing attack can recognize the kernel type such as linear, polynomial, and radial basis function (RBF) because of the `switch-case` statement in Lines 4–28. Also, when a victim used an RBF kernel, we were able to infer the number of features (i.e., the length of a vector) he or she used (Lines 11–20).

Apache. We ported some modules of Apache to SGX. Figure 16 shows its lookup function to parse the method of an HTTP request. Because of its `switch-case` state-

```
1  /* svm.cpp */
2  double Kernel::k_function(const svm_node *x,
3          const svm_node *y, const svm_parameter& param) {
4    switch(param.kernel_type) {
5  ★ case LINEAR:
6  ★   return dot(x,y);
7  ★ case POLY:
8  ★   return powi(param.gamma*dot(x,y)+param.coef0,
9                  param.degree);
10 ★ case RBF:
11     double sum = 0;
12     while (x->index != -1 && y->index != -1) {
13 ★     if (x->index == y->index) {
14 ★       double d = x->value - y->value;
15 ★       sum += d*d; ++x; ++y;
16 ★     }
17 ★     else {
18 ★       ...
19 ★     }
20       ...
21 ★   } // infer the lengths of x and y
22 ★   return exp(-param.gamma*sum);
23 ★ case SIGMOID:
24 ★   return tanh(param.gamma*dot(x,y)+param.coef0);
25 ★ case PRECOMPUTED:
26 ★   return x[(int)(y->value)].value;
27   default:
28     return 0;
29 ★ } // infer the kernel type
30 }
```

Figure 15: Kernel function of LIBSVM. The branch shadowing attack can infer the kernel type.

```
1  /* http_protocol.c */
2  static int lookup_builtin_method(const char *method,
3                                   apr_size_t len) {
4    ...
5    switch (len) {
6  ★ case 3:
7      switch (method[0]) {
8  ★   case 'P': return (method[1] == 'U' && method[2] == 'T'
9  ★     ? M_PUT : UNKNOWN_METHOD);
10 ★   case 'G': return (method[1] == 'E' && method[2] == 'T'
11 ★     ? M_GET : UNKNOWN_METHOD);
12     default: return UNKNOWN_METHOD;
13     }
14   ..
15 ★ case 5:
16     switch (method[2]) {
17 ★   case 'T': return (memcmp(method, "PATCH", 5) == 0
18 ★     ? M_PATCH : UNKNOWN_METHOD);
19 ★   case 'R': return (memcmp(method, "MERGE", 5) == 0
20 ★     ? M_MERGE : UNKNOWN_METHOD);
21     ...
22     }
23   ...
24 ★ }
25 }
```

Figure 16: Apache HTTP method lookup function. The branch shadowing infers the type of HTTP method sent by clients.

ments, we can easily identify the method of a target HTTP request, such as `GET`, `POST`, `DELETE`, and `PATCH`. Since this function invokes either no function or `memcmp()`, the controlled-channel attack has no chance to identify the method.