Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX

Tianlin Huo[1,4], Xiaoni Meng[1,4], Wenhao Wang[2], Chunliang Hao[3], Pei Zhao[4], Jian Zhai[4] and Mingshu Li[4†]

[1] University of Chinese Academy of Sciences, {tianlin,xiaoni}@nfs.iscas.ac.cn
[2] SKLOIS, Institute of Information Engineering, CAS, wangwenhao@iie.ac.cn
[3] China Electronics Standardization Institute, haochunliangtom@163.com
[4] Institute of Software, CAS, {zhaopei,zhaijian,mingshu}@iscas.ac.cn

Abstract. Software Guard Extension (SGX) is a hardware-based trusted execution environment (TEE) implemented in recent Intel commodity processors. By isolating the memory of security-critical applications from untrusted software, this mechanism provides users with a strongly shielded environment called enclave for executing programs safely. However, recent studies have demonstrated that SGX enclaves are vulnerable to side-channel attacks. In order to deal with these attacks, several protection techniques have been studied and utilized.

In this paper, we explore a new pattern history table (PHT) based side-channel attack against SGX named *Bluethunder*, which can bypass existing protection techniques and reveal the secret information inside an enclave. Comparing to existing PHT-based attacks (such as Branchscope [ERAG+18]), Bluethunder abuses the 2-level directional predictor in the branch prediction unit, on top of which we develop an exploitation methodology to disclose the input-dependent control flow in an enclave. Since the cost of training the 2-level predictor is pretty low, Bluethunder can achieve a high bandwidth during the attack. We evaluate our attacks on two case studies: extracting the format string information in the `vfprintf` function in the Intel SGX SDK and attacking the implementation of RSA decryption algorithm in mbed TLS. Both attacks show that Bluethunder can recover fine-grained information inside an enclave with low training overhead, which outperforms the latest PHT-based side channel attack (Branchscope) by 52×. Specifically, in the second attack, Bluethunder can recover the RSA private key with 96.76% accuracy in a single run.

Keywords: Software Guard Extension · Side-channel Attacks · Branch Prediction · 2-level Directional Predictor

1 Introduction

The Hardware-based Trusted Execution Environment (TEE) is a promising technique to enable secure computation. By running software within an isolated environment, a TEE protects software resources from being accessed by untrusted applications or the operating system (OS). ARM's TrustZone [ARM08], Intel's Trusted Execution Technology (TXT) [Gre12] and Software Guard Extensions (SGX) [Int14] are widely deployed commodity hardware-based TEEs. Among these TEE implementations, Intel's SGX is drawing significant attention these years because of its strong security guarantee, which enables a variety of new applications such as secure data analysis [SCF+15] and secure distributed computing [DSC+15, BWG+16].

[†] Corresponding authors: Wenhao Wang and Mingshu Li.

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925,
Vol. 2020, No. 1, pp. 321–347
DOI:10.13154/tches.v2020.i1.321-347

However, recent researches demonstrated that SGX is vulnerable to the following side-channel attacks: page table based attacks [VBWK+17, XCP15], cache-based attacks [BMD+17, GESM17, HCP17, SWG+17] and branch prediction unit (BPU) based attacks [ERAG+18, LSG+17, CCX+19]. Compared to the other two kinds of attacks, BPU-based attacks are getting considerable attention due to the following two reasons: first, most of current SGX protection approaches aim to defend against page table based and cache-based attacks, leaving the secrets in the enclave vulnerable to BPU-based attacks; Second, BPU-based attacks can identify fine grained control flow information inside an enclave (*i.e.*, instruction level *v.s.* page level and cache line level).

The branch target buffer (BTB) and the pattern history table (PHT) are two main components in the BPU. Most of existing BPU-based side-channel attacks against enclaves [CCX+19, LSG+17] leverage the BTB to conduct attacks. By abusing BTB entry collisions, the attacker can infer or even mislead the execution direction of a target branch in the enclave. However, BTB-based attacks can be prevented by current mitigation techniques [HLLP18, ABPK07, Int18]. Different from BTB-based attacks, PHT-based attacks are the ones which can still break through current protection mechanisms. Branchscope [ERAG+18] firstly performs a PHT-based attack against SGX. By manipulating hashing collisions in the *bimodal predictor* (namely, *the 1-level predictor*), this attack can extract fine-grained information of a target enclave. Nevertheless, Branchscope needs to execute a large number of (*i.e.*, 100,000) branches for activating the vulnerable predictor before each detection, which causes considerable training overhead during the attack and limits the usage of this attack.

In this paper, we propose *Bluethunder*[1], a new PHT-based attack which can reveal fine-grained control flows of an enclave program running on real SGX hardware. Bluethunder abuses collisions in the *2-level predictor* as a side channel. In contrast to Branchscope, which activates the bimodal predictor by executing a large number of branches for each detection, Bluethunder only needs to activate the 2-level predictor predictor once during the attack, which speeds up the attack significantly. Furthermore, since the PHT table is not flushed upon context switches and is shared between the 2 hyper-threads running on the same physical core, Bluethunder is still effective on the processor after updating the latest microcode patches or disabling SMT.

However, exploiting such channel on the 2-level predictor is not so straightforward in practice because ① since both the recent branch history and the address of the target branch are considered when indexing the 2-level predictor entries [LMB+], different entries may be used in different contexts even though predicting for the same target branch. As a result, it is hard for the attacker to construct entry collisions in the 2-level predictor. ② Although it has been disclosed that the key part of the 2-level predictor is a n-bit PHT [ERAG+18], the value of n has not been disclosed to the public; ③ Current BPU-based attacks usually require both the attacker and the victim processes to be executed in a sequential order (*e.g.*, first the attacker trains the predictor, then the victim executes the code, and the attacker detects the state changes at last), which limits the temporal resolutions of the attacks. To overcome these challenges, we developed three novel exploitation techniques: ① fixing the branch history of the victim's core by interrupting the SGX enclave and ② reverse-engineering the inner logic of the entries in the 2-level predictor; ③ proposing a detection method which can improve the temporal resolution of the attack by adjusting the branch directions of the attacker's target branch dynamically.

We evaluate Bluethunder against an SGX enclave on a recent CoffeeLake processor, targeting both the `vfprintf` function in SGX SDK and the sliding-window RSA-2048 decryption algorithm in mbed TLS (Section 6). Both of the experiments show that our

[1] We use Bluethunder to represent lightning (since its color is blue) and thunder, which is fast and powerful.

attack can outperform the latest PHT-based side channel attack (Branchscope) by 52×. Specifically, in the second experiment, Bluethunder extracts the whole 2048-bit private key with 96.76% accuracy by running the decryption only once. The overhead of Bluethunder for recovering one bit is 1.85×10^4 cpu cycles on average, which is only 1.9% of Branchscope (9.56×10^5 cpu cycles).

In summary, the main contributions of this paper are:

- *Novel PHT-based attack.* We introduce Bluethunder, a new PHT-based side-channel attack to extract the control flow of an enclave process. We also reverse-engineer the inner logic of the PHT entries in the 2-level predictor. To the best of our knowledge, this is the first side-channel attack on the 2-level directional predictor, which expands our understanding of side-channel attacks against SGX.
- *New Techniques.* ① We present a new method for constructing a given branch history with only 93 branches. By using this method, we recover the branch history information of an enclave interrupt. ② We propose a novel detection technique which enables the attacker to monitor the enclave's actions in a high temporal resolution, without caring about when the target instruction in the enclave is executed. ③ Bluethunder is the first attempt to use SGX interrupts for fixing the branch history. By interrupting the enclave just before the target branch, the attacker can ensure that the branch history is the same each time when the target branch in the enclave program is executed.
- *Implementation and Evaluation.* We implement Bluethunder on real SGX hardware with the latest hardware patches against speculative execution attacks, and evaluate this implementation on two case studies: attacking the `vfprintf` function and the RSA algorithm. We also confirm Bluethunder on both scenarios when the attacker and victim run on 2 hyper-threads logical cores, and on a same logical core without hyper-threading.

Responsible disclosure. We have disclosed Bluethunder to the security team at Intel before releasing our study to the public. The implementation of Bluethunder against SGX will be open sourced later.

2 Background

2.1 BPU

The BPU is an optimization design of modern pipelined processors. By predicting the possible execution path of a process, the BPU can speed up the fetching process in a processor design, which rescues the processor from waiting for the completion of the previous instructions.

Figure 1 illustrates one possible design of the BPU, which has two main components: the branch directional predictors and the BTB. The branch directional predictors are for predicting the possible jump direction (jump or not) of a branch instruction; while the BTB is for predicting the jump destination address of this branch. When a branch comes, the BPU usually uses one branch directional predictor to predict the possible execution direction of this branch. If the prediction result turns out to be taken (namely jump), the instructions starting from the address predicted by the BTB will be fetched and executed in advance; or if the branch is predicted to be not-taken, target address prediction made by the BTB will be ignored. At the same time, the predictor is updated according to the actual jump directions of the incoming branch.

PHT. The main component in a branch directional predictor (*i.e.*, the bimodal predictor or the 2-level one) is PHT, which is a recording table containing several n-bit saturating

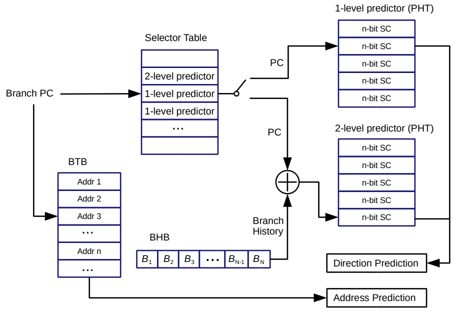


Figure 1: A demonstrating structure of the BPU with a hybrid branch predictor. SC – Saturating Counter, B_N – The Nth branch (or jump).

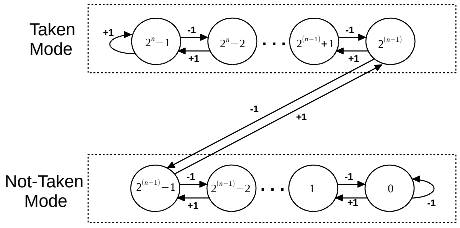


Figure 2: The finite state machine of an n-bit saturating counter. When this counter is in the taken mode, its prediction output will be taken, and vice versa.

counters. When a directional predictor works, one PHT entry in it can be selected for predicting the possible direction of the coming branch. After this branch is actually executed, the counter given by the relative entry of the PHT is updated. A finite state machine (FSM) description of updating an n-bit saturating counter is given in Figure 2. If the counter value is less than 2^{n-1}, this counter is in a "not-taken mode", which means the direction predicted is "not-taken"; otherwise, the counter is in a "taken mode" and the prediction made is "taken". For example, a 3-bit saturating counter with the value 0 is in the "not-taken mode"; while another one with the value 4 is in the "taken mode".

The Bimodal/2-Level Predictor. The bimodal predictor and the 2-level predictor are two main branch directional predictors in modern processors. The bimodal predictor exploits the observation that the branch direction outcome has a relationship with the previous history of this branch; while the 2-level predictor [YP91] believes that the outcome is also affected by the jump history of other recent branches.

When a branch first comes, the BPU usually chooses the bimodal predictor for predicting the possible execution direction of this branch [ERAG+18]. However, if this predictor makes mistakes several times, the BPU will select the 2-level predictor instead in order to achieve high performance. A selector table in the BPU (as shown in Figure 1) can identify which predictor is likely to perform better for the coming branch based on the previous history.

Although both the bimodal and the 2-level predictors [YP91] rely on a PHT table for predictions, their PHT indexing algorithms are different. The bimodal predictor only considers the address of the target branch when indexing; while the 2-level predictor considers both the address of the target branch and the recent branch history, which is recorded in the branch history buffer (BHB), when indexing entries [LMB+, H+18]. Also, their prediction accuracy is different. Since the 2-level predictor utilizes more branch information, its prediction results can be more confident than those made by the bimodal predictor in a number of scenarios.

2.2 SGX

Intel SGX [CD16] is an implementation of hardware-based TEE that is supported by modern Intel CPUs (since Skylake). To enforce the physical memory isolation in hardware, SGX introduces a set of new CPU instructions which can be used to create and manage isolated software components [MAB+13], called enclaves. The data in enclaves cannot be accessed by untrusted software running in non-enclave environments (*i.e.*, privileged software like the operating system and the hypervisor). However, since SGX enclaves can live in the virtual address space of conventional processes via an SGX instruction set, these enclaves are able to access the memory region of non-enclave processes easily.

Non-enclave code and enclave code interaction. Enclaves can only be entered or exited through a few predefined entry points. The `EENTER` and `EEXIT` instructions transfer control between the untrusted software and an enclave. When an interrupt or exception happens, the processor performs an Asynchronous Enclave Exits (AEX), which saves the registers of the interrupted enclave into the enclave memory called State Save Area (SSA), clears the CPU registers, and transfers control to a pre-specified instruction outside the enclave. After the interrupt or exception is handled, an `ERESUME` instruction is executed, which reloads the previously saved context from the SSA frame and continues execution in the enclave.

3 Threat Model

We assume the standard SGX threat model, in which the attacker has full control over privileged software, such as the operating system.

First, we assume that the attacker has access to the target enclave program's source code and/or binary. By analyzing these resources, the attacker can obtain the detailed behavior of an enclave, especially this enclave's control flow and jump information (*i.e.*, the source address and the destination address) of branches. The programs with obfuscated code (*e.g.*, code from remote servers) are outside the scope of our attack.

Second, we assume that the attacker and victim programs are co-resident on the same physical core. This is because the BPU is shared at logical core level, but separated at physical core level. Such co-residency can be forced easily by using thread binding techniques [EPAG16, LSG+17].

Third, the attacker is able to measure the misprediction information of her own branches. Both the performance monitoring counters (PMC) [LSG+17] and the *rdtscp* instruction [ERAG+18, LSG+17] are available for this measurement.

Fourth, the attacker can interrupt an enclave just before the target branch in the enclave is executed, ensuring that no other jump actions are taken before the execution of the target branch. This ability enables the attacker to construct the same recent branch history as the enclave's, as well as to control the execution speed of the enclave. Although this goal is hard to achieve when interrupting a common user-level process, it can be achieved perfectly when interrupting an enclave, since the attacker can leverage the OS's control over hardware timer devices [VBPS17].

4 The Bluethunder Attack

4.1 Overview

The Bluethunder attack aims to obtain the fine-grained control flow of an enclave program by manipulating the 2-level directional predictor in the BPU. To achieve this goal, the attacker should have the ability to activate the 2-level predictor as well as to abuse this predictor to leak secrets in an enclave. In general, the Bluethunder attack consists of the following two stages (shown in Figure 3):

- Stage 1: Activating the 2-level predictor. We force the predictors other than the 2-level predictor to shut down by imposing several mispredictions on them, ensuring that the 2-level predictor is eventually activated and used by processes on the target core. Executing the target branch with the directions described in a TBDV-Q (target branch

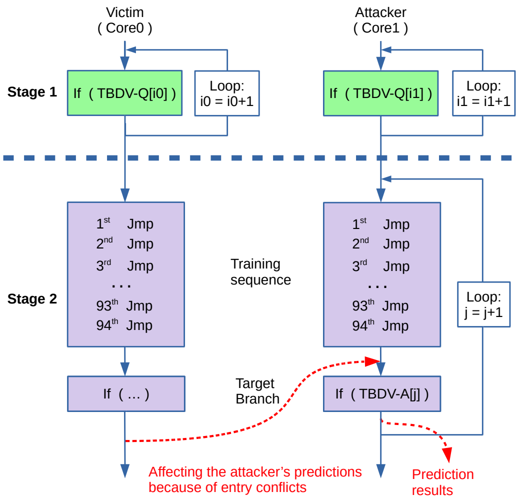


Figure 3: Overview of the Bluethunder attack. The two stages are "Activating the 2-level predictor" and "Leaking Secrets" respectively.

direction vector for quiescing, Section 4.2) direction vector can mislead the BPU to make mispredictions easily. This preparation stage is only executed once during the attack.

- Stage 2: Leaking secrets. Two steps are included in this stage: constructing collisions and abusing collisions. First, the hashing collisions, which are caused by predicting different branches through the same PHT entry, are established between the victim (*i.e.*, in an enclave environment) and the attacker (*i.e.*, in a non-enclave environment) processes. We propose a simple method to construct a given branch history, followed by an advanced method which can be used to recover the branch history of an enclave interrupt. This advanced approach can also reduce the training cost of our attack since only 93 jump instructions are required for history training. Second, the attacker probes the state changes of the collision entries by executing the target branch with a TBDV-A (target branch direction vector for attacking, Section 4.3) vector. By analyzing these probing results, the attacker can infer the control flow of the target enclave program, as well as the secret in it. This attacking stage must be executed for each detection during the attack.

4.2 Activating the 2-Level Predictor

In this section, we describe how to activate the 2-level predictor in the BPU, which is the first step of our Bluethunder attack against SGX. We turn to present the way to abuse this predictor for attacks in Section 4.3.

Our activating method is based on the following observation: the 2-level predictor is chosen by the BPU for predictions only when the bimodal predictor cannot predict well. As a result, it is viable to shut down the bimodal predictor by imposing several mispredictions on this predictor. We construct an efficient instruction sequence needed to shut down the bimodal predictor and correctly train the 2-level predictor with a conditional branch wrapped up in a loop. The execution directions of this conditional branch (i.e., whether the branch is taken or not) is determined by a TBDV-Q direction vector as follows.

- *The TBDV vector.* A "*target branch direction vector (TBDV)*" is a basic vector which is used to form a TBDV-Q vector. The function of this TBDV vector is to mistrain the bimodal predictor. Since the bimodal predictor uses 2-bit saturating counters for predictions [ERAG+18], any direction vector which does not match this pattern can lead to mispredictions of the target branch. TBDV is generated as an L-bit ($0 < L < 100$) vector with randomized 0/1 bits, where 0 means *not-taken* and 1 means *taken*.

- *The TBDV-Q vector.* The TBDV-Q vector is designed to mistrain the bimodal predictor as well as to train the 2-level predictor. It simply repeats the TBDV by N times.

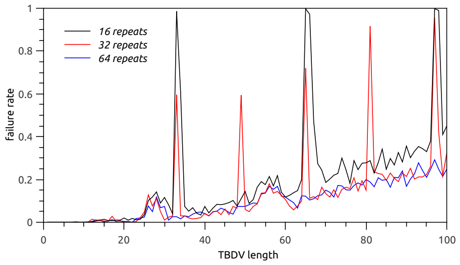


Figure 4: The failure rate of executing TBDVs with different lengths L.

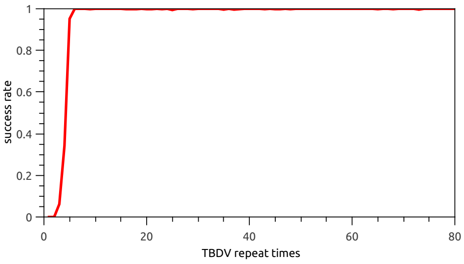


Figure 5: The success rate of executing "111,000" with different repeat times N.

To verify whether the TBDV-Q vector is capable of shutting down the bimodal predictor and activating the 2-level predictor, we check the prediction result of the last TBDV in the TBDV-Q (*e.g.*, using PMC or timing information). If all the directions of the last TBDV are predicted correctly, we conclude that the pattern of this TBDV has been learnt by the BPU successfully. We ensure that the bimodal predictor cannot achieve that by setting a large L and randomized 0/1 bits, then we can conclude that the 2-level predictor has been selected and trained. Otherwise, we cannot infer which directional predictor is being used by the BPU.

The remaining task is to find a proper setting of L and N. Firstly, we'd like to figure out the value for L ($0 < L < 100$) such that the TBDV could mistrain the bimodal predictor, but can be predicted well by the 2-level predictor. It should be noticed that L cannot be too small (to mis-train the bimodal predictor) nor too big (to fit into the size of the PHT used by the 2-level predictor). We used fixed values for N (*i.e.*, 16, 32 and 64), which are large enough to train the 2-level predictor and can be used to detect whether the bimodal predictor is disabled. It is worth noting that the value of N is not limited to multiples of 16. We test each length for 500 cases, repeating 3 tests for each case. Any success among the 3 repeating tests proves that this TBDV-Q is able to activate the 2-level predictor. The results are shown in Figure 4. It can be seen when the length L is less than or equal to 22, the BPU almost always predicts the target branch successfully, which means that the 2-level predictor is activated. Also worth noting is that when L equals 33, 49, 65, 81, or 97 (i.e., the distances between which are multiples of 16), the predictions fail a lot. We conjecture this is related to the structure of the PHT table. Since this does not affect our attack, we plan to conduct another research to explain this effect in the future.

In order to reduce the value of N, we perform another experiment by running TBDV-Qs with varied N. All of the tested TBDV-Qs are generated with the TBDV "111,000", since this TBDV will be finally used in our attacks. We test 500 cases for each N in this experiment and record the success rate of each N. The results are shown in Figure 5. We find that the 2-level predictor can be activated and trained by executing the TBDV-Q vector containing at most 7 TBDV segments.

In the following sections if not explicitly stated otherwise the TBDV-Q vector where $L = 6$ and $N = 7$ (i.e., the one which repeats the TBDV "TTTNNN" for 7 times) is chosen to activate the 2-level predictor.

4.3 Leaking Secrets

After activating the 2-level predictor, we move to manipulate this predictor for side-channel attacks. We first describe the way to construct entry collisions between an enclave process (*i.e.*, the *victim* process) and a non-enclave process (*i.e.*, the *attacker* process), then turn to abuse these collisions. For simplicity, we let the attacker process and victim process run on two hyper-threads in a same physical core. However, as PHT states are not flushed during context switches, similar techniques could also be used if hyper-threading is not supported and the two processes run on the same physical core[2].

4.3.1 Constructing Collisions

The main component of the 2-level predictor is a PHT table which has several recording entries. In order to construct entry collisions, we have the assumption that the 2-level predictor usually makes the same predictions in the same contexts. In other words, the same PHT entry is used in this case. If the context can be recovered, its relative PHT entry can be accessed even though the entry indexing function is opaque. According to Ben Lee *et al.* [LMB+], the branch history and the address of the branch target are two main elements affecting the context. As a result, these two elements have an influence on the PHT indexing.

To verify whether the entry indexing of the 2-level predictor is only affected by the two elements described in [LMB+], we carried out another experiment. In this experiment, the victim and attacker threads run the same piece of code. This code contains several taken branches called *training sequence* and one branch following them called *target branch.* The training sequence is for fixing the branch history of a logical core, and the target branch is for activating the 2-level predictor and manipulating the target 2-level predictor entry. Note that the number of branches in the training sequence should be enough for fixing the branch history. The execution of the target branch can be divided into two parts: the *activating part* is used to activate the 2-level predictor; and the *abusing part* is to test whether hashing collisions of the 2-level predictor have been constructed successfully. We first run the attack process alone, recording the attacker's misprediction result of the abusing part. Then we run the victim and attacker programs simultaneously across hyper-threaded cores, also record the attacker's misprediction result of the abusing part during the test. We use semaphores to force the two processes to execute each branch in turn precisely. By comparing the misprediction results of the two tests, we can infer whether hashing collisions have been constructed successfully.

The attacker's misprediction results of the two tests are given in Table 1. If the attacker process runs alone, only 4 out of the 11 predictions are incorrect. While the attacker runs with the victim, 9 mispredictions occur. This difference shows that the predictions of the attacker's branch operations can be influenced by the victim. We repeat this test several times, and the results are always the same.

We also reduce the number of taken branches in the training sequence, detecting the minimum number required for fixing the branch history. The result shows that 93 taken branches are enough to ensure that the branch history is fixed. To check the effect of non-taken branches, we add several not-taken branches into the attacker's code and *nop* instructions into the victim's code to keep the invariance of the branch addresses between the two programs. The result shows that no matter how many not-taken branches are added or where they are, the entry collisions still exist. In conclusion, not-taken branches have no effects on the entry indexing of the 2-level predictor.

Based on the experiment above, we draw the conclusion that if both the branch history and the address of the target branch are the same between two processes, these processes

[2] We confirmed Bluethunder on the platforms available to us, with the latest microcode patches at the date of writing.

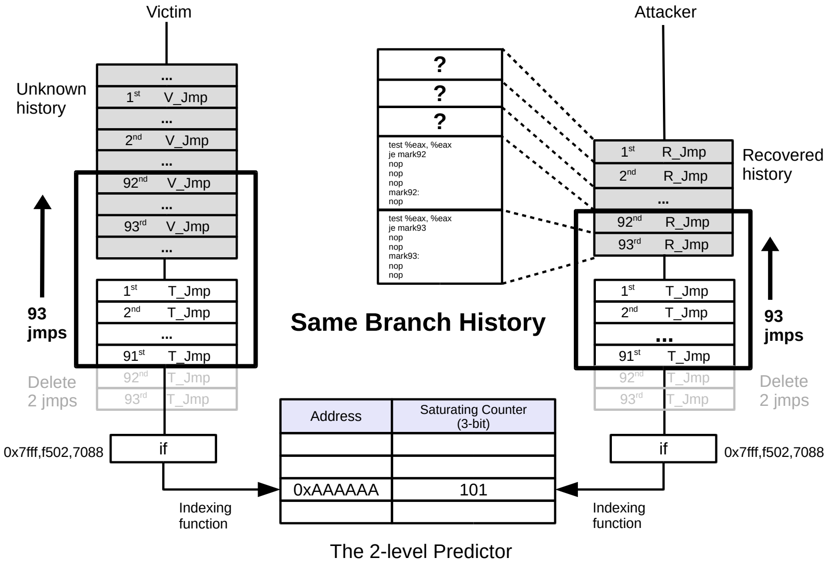
Table 1: The impact of the victim program on the misprediction result of the attacker program. M – misprediction, H – hit (correct prediction), N – not taken, T – taken.

Index	Attacker Alone		Attacker and victim			
	Attacker Direction	Attacker Result	Victim Direction	Victim Result	Attacker Direction	Attacker Result
i	T	M	T	M	T	M
$i+1$	T	M	T	M	T	M
$i+2$	N	H	T	H	N	M
$i+3$	N	H	T	H	N	M
$i+4$	N	H	T	H	N	M
$i+5$	N	H	T	H	N	M
$i+6$	N	H	T	H	N	M
$i+7$	N	H	T	H	N	M
$i+8$	N	H	T	H	N	M
$i+9$	T	M	T	H	T	H
$i+10$	T	M	T	H	T	H

will share the same 2-level predictor entry. However, although it is easy to infer the address of the target branch, it is pretty complicated to recover the branch history of the victim process since its value varies in different contexts. A simple way for the attacker to recover the victim's branch history is to executes the same flow of the branch (or jump) instructions, whose addresses are exactly the same as those in the victim program. However, since the recent branch history changes in different contexts, an enclave may use different predictor entries for predicting the target branch each time. As a result, the attacker needs to catch up with the execution of the victim enclave and monitor the right predictor entries during the attack, which is difficult to achieve in real-world attacks. To deal with this difficulty, we send interrupts to the enclave, in order to fix the branch history of the enclave's core. This is because when an interrupt comes during the enclave's execution, this enclave will be suspended and several interrupt handling operations defined in the kernel will be taken. We observe that these handling operations (*e.g.*, resuming the enclave) usually form the branch history of the target core into a fixed state. This is because the last operations are always about interrupt recovery, which is the same for any APIC interrupt. Since enough jump operations are included in the last operations, they can force the branch history to achieve a fixed state. As a result, these operations can be used as the training sequence for our attacks. At the same time, since some pieces of code the enclave executes for interrupt handling are secret to us (*e.g.*, the leaf function `ERESUME`), we cannot recover the interrupt history by simply copying the target code. Instead, we turn to detect the branch history updating algorithm of recent Intel processors, and demonstrate an advanced way for recovering the branch history after an enclave interrupt.

Recovering the Branch History. In recent Intel processors, both the target branch address and the recent branch history are used to index PHT entries. It is observed that in Intel Haswell processors, the prediction scheme uses a global branch history buffer, which contains information about the last K ($K = 28$ for Haswell) branches [H+18, KHF+19]. The authors found it is likely that each update of the history buffer shifts the history buffer by 2 bits, and the least significant 2 bits of the new history buffer are calculated using the last 2 bits of the destination address and bits 0x40 and 0x80 of the source address of the latest branch. In this section, we try to understand the branch history updating algorithm on our testbed with Skylake and CoffeeLake processors and show how to recover the enclave's branch history following an interrupt.

According to recent studies [H+18, KHF+19], we observed that the branch history update can be controlled by manipulating the last two bits of the latest branch's destination



```
test %eax, %eax
je mark92
nop
nop
nop
mark92:
nop
```

```
test %eax, %eax
je mark93
nop
nop
mark93:
nop
nop
```

Figure 6: Recovering the branch history when 2 branches in the victim process have been recovered. The grey block in the victim program is the branch history the attacker needs to recover, and the grey block in the attacker program is the history result recovered.

address in Haswell processors. In order to test whether this observation also works for the Skylake and the CoffeeLake processors, we do the following test based on the previous conclusion that 93 jumps are enough for constructing a given branch history. We create two processes in this test, namely the attacker and the victim. In the victim's code, we add 93 jumps and a target branch. We also add 93 branches and a target branch into the attacker's code, but only the last 92 jumps and the target branch are the same as the victim's. We vary the jump destination of the first jump instruction added in the attacker's code, while fixing the source address of this jump. By checking whether the two processes can affect each other, we can infer that an entry collision occurs. Since the addresses of the two target branches in the two processes are the same, we can deduce that the branch history of the two processes is the same. The test result shows that entry collisions occur only when the last two bits of the first jump's destination are fixed values (*e.g.*, 01). This result means that in either Skylake or CoffeeLake processors, only the last two bits of a jump's destination address are enough to represent the effects of this jump on the branch history.

We make use of this result to recover the enclave's branch history after an interrupt (as shown in Figure 6). Two steps are included in this recovery: first, we add 93 jump instructions, called recovered jumps, into the attacker's code. The source addresses of these jumps are fixed upon insertion, while their destination addresses can be changed. We also add 93 jump instructions, which are called test jumps (T__Jmp), and a target branch in both the victim's code and the attacker's code. Note that the source and destination addresses of each victim's T__Jmp instruction (or the target branch) should be the same as that of the corresponding one in the attacker's code. Now the two target branches share the same 2-level predictor entry for predictions, since the added 93 branches can flush the recent branch history and set the branch history of the two processes to be the same. Then, we delete the last (*i.e.*, 93rd) T__Jmp instruction in both processes, and change the destination of the last (*i.e.*, 93rd) recovering jump (R__Jmp) instruction in the R__Jmp sequence. By checking whether an entry collision has been established, we can infer when the branch history of the two processes is the same. According to the

Table 2: The result of executing the attacker's target branch with the TBDV "000,000,111,111", in which "0" means "not taken" and "1" means "taken" (running the attacker process alone). M – misprediction, H – hit (correct prediction), N – not taken, T – taken.

Index	Attacker Direction	Attacker Result	Index	Attacker Direction	Attacker Result
i	N	M	$i+6$	T	M
$i+1$	N	M	$i+7$	T	M
$i+2$	N	M	$i+8$	T	M
$i+3$	N	M	$i+9$	T	M
$i+4$	N	H	$i+10$	T	H
$i+5$	N	H	$i+11$	T	H

conclusions drawn above, four times are enough for recovering the last R_Jmp instruction in the attacker's program. Next, we delete the penultimate (*i.e.*, 92nd) T_Jmp instruction in both processes, and recover the jump destination of the penultimate (*i.e.*, the 92nd) R_Jmp instruction by checking again. We repeat this step 93 times in total, and finally the branch history after an enclave interrupt can be reconstructed. Figure 6 shows the recovering details when the last two (*i.e.*, 92nd and 93rd) R_Jmp instructions have already been recovered.

4.3.2 Abusing collisions

By constructing entry collisions, the attacker has the ability to access the target entry in the 2-level predictor. Now we try to abuse this entry to conduct Bluethunder attacks. We start with detecting the inner working logic of a 2-level predictor entry. Then we try to launch side-channel attacks by making use of this logic.

Detecting the 2-level predictor logic. Although it has been uncovered that the main component in the 2-level predictor is a PHT with several n-bit saturating counters [ERAG+18], the value of n is still unknown. To detect the value of n, we execute an attacker process alone, which contains 93 training branches and one target branch. The TBDV we use is "$111\cdots1000\cdots0$", which has L_1-bit 0s and L_1-bit 1s. L_1 is set to be 6 in this test. We repeat this TBDV for 100 times and generate the final TBDV-Q for the target branch. We noticed that this TBDV-Q vector can be used to quiesce the bimodal predictor and activate the 2-level one. By executing the target branch with this vector and checking the misprediction result of the target branch, we can infer the exact value of n.

The test result is presented in Table 2. After executing the target branch 4 times with taken (or not-taken) directions, the 2-level predictor can be trained. In other words, there can be at most 4 continuous mispredictions. For the worst case of training the 2-level predictor whose PHT entry is an n-bit saturating counter, after 2^{n-1} mispredictions in the use of the same PHT entry, the predictor can be trained. Hence, we conclude that n is 3. In other words, the 3-bit saturating counters are utilized by the 2-level branch predictor.

In order to verify that this conclusion also works across SMT, we repeat this test with two processes (i.e., the attacker and the victim) running on sibling cores this time, instead of one process (i.e., the attacker) alone. By executing the same piece of code, these two processes point to the same 2-level predictor entry. We utilize semaphores to force them to execute the target branches one after the other. When the attacker (or the victim) process executes the target branch with a not-taken direction, the value of this saturating counter decreases by 1; otherwise, its value increases by 1. The saturating counter values after each attacker's (or victim's) execution are listed in Table 3. This test result shows that the execution directions of the two target branches can affect each other, and their

Table 3: The result of executing the attacker's target branch with the TBDV "000,000,111,111", in which "0" means "not taken" and "1" means "taken" (run both processes together). M – misprediction, H – hit (correct prediction), N – not taken, T – taken, Value After – the binary value of the saturating counter after executing the target branch.

Index	Victim Direction	Victim Result	Value After	Attacker Direction	Attacker Result	Value After
i	N	M	110	N	M	101
$i+1$	N	M	100	N	M	011
$i+2$	T	M	100	N	M	011
$i+3$	T	M	100	N	M	011
$i+4$	N	H	010	N	H	001
$i+5$	T	M	010	N	H	001
$i+6$	T	M	010	T	M	011
$i+7$	N	H	010	T	M	011
$i+8$	N	H	010	T	M	011
$i+9$	T	M	100	T	H	101
$i+10$	N	M	100	T	H	101
$i+11$	T	H	110	T	H	111

behaviours are in accordance to the FSM of a 3-bit saturating counter. For example, before the execution of the i-th branch, the value of the 3-bit saturating counter is 7 ("taken mode"), and its prediction result is taken. When the victim's i-th not-taken target branch comes, a misprediction occurs and the value of the saturating counter is updated to be 6. Then it is the attacker process's turn. The attacker also executes her target branch with a not-taken direction. Since the current value of the counter is 6 (also "taken mode"), another misprediction occurs and the value of the saturating counter is updated to 5.

Manipulating the 2-level predictor. The predictor achieves a balanced state for predicting a branch if the relative saturating counter used for the prediction reaches a critical value. After updating to the critical value, the change of the saturating counter can cause a different prediction of the branch direction. Both 2^{n-1}-1 and 2^{n-1} are critical values for the n-bit saturating counter. For a 3-bit saturating counter as an example, suppose its current value is 3, which is a critical value. If a taken branch uses this counter for the prediction, the counter is updated to 4, indicating a taken mode for the prediction; otherwise, if a not-taken branch uses this counter, the counter changes to 2, staying at the not-taken mode for the prediction. By monitoring the changes of the predictions, the branching behaviors can be detected.

In order to set a saturating counter to the critical value, we adjust the branch directions of the target branch dynamically. The next branch direction is set opposite to the predicted direction. For example, if the relative 2-level PHT entry uses the saturating counter to give a taken mode prediction, the attacker sets the direction for the target branch to be not-taken, resulting in the decrease of the saturating counter; otherwise, if the saturating counter currently gives a not-taken prediction, the attacker sets the direction for the target branch to be taken, resulting in the increase of the saturating counter. We call this direction vector which the attacker executes the "target branch direction vector for attacking (TBDV-A)".

Without the execution of the victim program, the TBDV-A which the attacker executes should reach a stable state eventually, repeated with the basic sequence "TTTNNN". The explanation of this phenomenon is presented in Table 4. We suppose that after executing the target branch $i-1$ times, the binary value of the 3-bit saturating counter is 101 ("taken mode"). When the i-th not-taken branch comes, this counter will make a misprediction and

Table 4: The relationship between the attacker's execution and the value of the target 2-level predictor entry. M – misprediction, H – hit (correct prediction), N – not taken, T – taken, Value Before (or After) – the binary value of the saturating counter before (or after) executing the target branch.

Index	Value Before	Attacker Direction	Attacker Result	Value After
i	101	N	M	100
$i+1$	100	N	M	011
$i+2$	011	N	H	010
$i+3$	010	T	M	011
$i+4$	011	T	M	100
$i+5$	100	T	H	101

Table 5: The corresponding relationship between the attacker's execution directions and the victim's action. T – a taken branch, N – a not-taken branch

Case No.	Attacker's Sequence	Victim's Action
1	TTTNNN	None
2	TTTNN	N
3	TTTNNNN	T
4	TTNNN	T
5	TTTTNNN	N
6	OTHER	Noise

its value decreases by 1. Finding that the predictor makes mispredictions when predicting a not-taken branch, the attacker learns that the target PHT entry is in the "taken mode" now, and decides to run a not-taken branch next time. After repeating $(i+5)$ times, the value of the relative saturating counter turns to 101 again. As a result, another turn of the circle starts.

However, if the attacker process is run with the victim process, this stable state will be broken. This is because both programs share the same 2-level PHT entry, and the execution of the victim's target branch has an influence on the execution of the attacker program. Any irregular sequence in the attacker's execution path which do not match the "TTTNNN" may be caused by the victim process. By analyzing these irregular sequences, the attacker can finally deduce the fine-grained control flow of the victim process. The relationship between the possible irregular sequences and the victim's actions are presented in Table 5. If the number of "N" contained in the detected execution sequence is larger than that of "T", a target branch with a taken direction is executed by the victim process; if the number of "N" contained in the detected execution sequence is smaller than that of "T", the victim should execute the target branch with a not-taken direction. For example, if the execution direction sequence of the attacker's target branch is "TTTNNNN", we conclude that the victim's execution direction is "T". This is because the attacker needs to decrease the value of the saturating counter one more time in order to cancel the victim's effects on this counter.

Misprediction Measurement. Measuring the misprediction information of a branch is necessary for conducting BPU-based side-channel attacks. A common method is to read the PMC, which is also suitable to Bluethunder. Although PMC provides accurate information of hardware events, it requires the user to have the system-level authority, which limits its usage. So, we also try an alternative measurement approach utilizing the time stamp counter (TSC). By tracking the number of cycles to execute a branch, the attacker can infer whether it is incorrectly predicted, and thus the current state of the target 2-level predictor entry. We use the *rdtscp* instruction to get a timestamp. Also, two *mfence*

Table 6: Comparison between Bluethunder and other attacks against SGX.

Attack	Spatial Resolution	Size	Temporal Resolution	Detectability
TLBLEED [GRBG18]	Page	4KB	Medium	Low
Prime+Probe [OST06]	Cache-set	512B	Medium	High
MemJam [MWES19]	Intra cache-line	4B	Medium	Medium
PORTSMASH [ABuH+19]	Execution port	uop	High	Low
Branchscope [ERAG+18]	PHT entry	instruction	Low	High
Bluethunder	PHT entry	instruction	High	Low

instructions are added before and after the *rdtscp* instruction, avoiding the out-of-order execution of the load and store operations. Note that in Bluethunder attacks, the time latency caused by the caching mechanism in this TSC-based measurement is neglectable. This is because the instructions which are measured are frequently used, and they are likely to be cached.

Comparison with other side channels. We also compare the Bluethunder attack with other SGX attacks, and the result is shown in Table 6. It is worth noting that since no shared memory exists between the victim enclave and the attacker, "Flush+Reload" [YF14] and "Flush+Flush" [GMWM16] cannot be used. According to Table 6, Bluethunder is able to recover instruction-level information, whose spatial resolution is much higher than page-level [GRBG18] or cache-level attacks [OST06, MWES19]. Also, since Bluethunder does not require cache eviction [OST06] or executing a large number of jump instructions for training [ERAG+18], its detectability is low. As for the temporal resolution, Bluethunder enjoys a high accuracy since its training overhead is extremely low. At the same time, we note that both Bluethunder and PORTSMASH enjoy high temporal resolutions and low detectability, and PORTSMASH can even achieve a higher spatial resolution than Bluethunder. However, since Bluethunder abuses a totally different processor component, it can bypass most of the defensive techniques, such as disabling SMT or preventing the port contention, which are available to defend PORTSMASH attacks.

5 Implementation of Bluethunder

We implemented Bluethunder attacks based on SGX-Step [VBPS17], which is an attacking framework against enclave programs with single-step interrupt execution control. Since SGX-Step is able to interrupt each instruction in the enclave, the APIC interrupt handling operations are executed just before each instruction. As a result, the recent branch history before each instruction should be the same, and the same 2-level predictor entry is usually selected for predicting the same target branch. By monitoring the entry affected by the victim's target branch, the attacker can infer the control flow of the victim process and the secret in it.

In order to train the attacker's branch history to be the same as the interrupt handling operations, we construct a training branch sequence in the attacker's code using the method described in Section 4.3.1. It has been verified that by using the sequence generated, hashing collisions can be established successfully. Based on the fact that both the branch history and the address of the target address of the two processes are the same, the attacker can share the same 2-level predictor entry with the victim (as shown in Figure 7). Note that only 93 instructions are required in this training sequence. Considering that Branchscope [ERAG+18] needs to execute 100,000 branches to train the BPU before each detection, the training cost of Bluethunder is negligible.

The whole execution period of the victim program running inside the enclave is

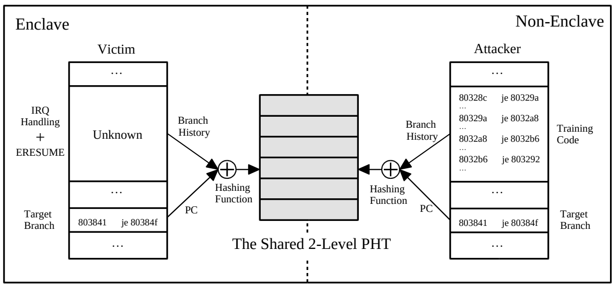


Figure 7: Implementation in an SGX-based environment. The same branch history and the same PC lead to the same 2-level PHT entry.

monitored by the attacker. We rely on an open source tool called `testp`[3] to measure whether a misprediction occurs. To recognize when the victim's target branch is interrupted, Bluethunder records the address of each instruction interrupted in the enclave. This address information can be provided by SGX-Step. Only the instruction whose address equals the target branch's address can be considered.

The *rdtscp* measurement is also implemented in this test. To improve accuracy, we add a threshold-deciding logic into the attacker's code, which can decide the timing threshold between predictions and mispredictions dynamically according to the current state of the processor. To further reduce the noise in our measurements, we also determine the attacking window by recording the times of entering and leaving the enclave, respectively. As the time spent for one instruction's interrupt handling is much longer than that for executing one instruction in the enclave, monitoring the execution time within an attacking window can reduce the amount of noise significantly.

6 Evaluation

We performed Bluethunder attacks on a B360-HD3 mainboard with a CoffeeLake i7-8700 processor and 32GB DDR4-2400 memory. This platform runs Ubuntu 16.04 with a generic 64-bit Linux 4.15.0 kernel, equipped with SGX Driver 2.0. We use the `taskset` command to bind the victim and attacker processes to a specific core, and the `isolcpus` command to isolate these cores from certain requests scheduled from the operating system. We demonstrate 2 attack cases when the attacker process and victim process run on two hyper-threads on the same physical core, first against the `vfprintf` function (Section 6.1) and then against the RSA implementation of the mbed TLS library (Section 6.2). We also show the results against the RSA algorithm when hyper-threading is not supported (Section 6.3).

6.1 Attacking Vfprintf Function

In order to demonstrate how Bluethunder reveals instruction-level secret information inside an enclave, we focus on examples in which neither the controlled-channel attack nor the cache-based attack can extract secret information ideally. Specifically, we choose the *vfprintf* function defined in SGX SDK as our target, whose function is to print a format string in an enclave. This function uses a switch-case statement to interpret the string format, which can be leveraged by the attacker. However, since several control flows in this switch-case statement are within a single page or even a single cache line, both the

[3] https://www.agner.org/optimize

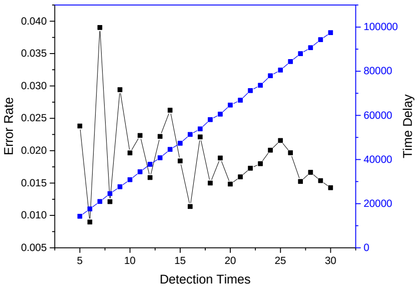


Figure 8: The error rate and time latency of different detection times.

controlled-channel attack and the cache-based attack cannot infer the fine-grained format string information in this *vfprintf* function.

Compared to the attacks above, Bluethunder can disclose the format string information by checking the switch-case statement in the *vfprintf* function. We set the string format to be "%llx" and repeat this print action 100,000 times. The average error rates of the experiments conducted with the CoffeeLake processor are presented in Figure 8. With the increase of the attacker's detection times, the delay of our attack increases linearly, and the accuracy is also improved in our attack. When the number of detection times is six, 17610 timestamps are required to recover one bit of the victim's secret, and the error rate is 0.83%. Since this error rate is the smallest one in our experiment, we choose six to be the detection times for our attacks eventually. It is worth noting that the time delay caused by the SGX-Step framework is not considered in this test. This is because Bluethunder is independent of SGX-Step and by using other interrupting techniques, this extra overhead can be decreased significantly.

We also test Bluethunder with the *rdtscp* measurement. To analyze the threshold of predictions and mispredictions, we execute 20,000 samples branches (10,000 for predictions and 10,000 for mispredictions separately) on the CoffeeLake processor. The result illustrates that when the target branch is mispredicted, the target branch takes 31 cycles on average; otherwise, the target branch takes 25 cycles on average. By configuring the threshold between predictions and mispredictions carefully, the attacker is able to recover 87.93% bits in a single test.

6.2 Attacking RSA Algorithm

We also launch Bluethunder against the mbed TLS Library[4], which is a popular choice for conducting encryption and decryption operations in SGX-based environments [LSG+17, SLK+17, SLKP17]. It has been shown that the RSA algorithm implemented in this library is vulnerable to control flow attacks [LSG+17]. The core function relating to the attacks is named `mbedtls_mpi_exp_mod` (presented in Appendix A), which is used for modular exponentiation calculations by leveraging the sliding-window exponentiation algorithm. This function has two conditional branches whose execution directions are affected by the private key directly. By detecting the execution directions of these two branches, the attacker is able to uncover the control flow of the target process, and thus the private key of the victim.

Our attack target is the decryption operation in the RSA-2048 algorithm. The enclave calls this RSA decryption function defined in mbed TLS library directly, with a random pair of keys generated by mbed TLS. The attacker's goal is to recover the RSA private key the victim (i.e., the enclave program) uses. We start the attacker process before the victim

[4] https://tls.mbed.org

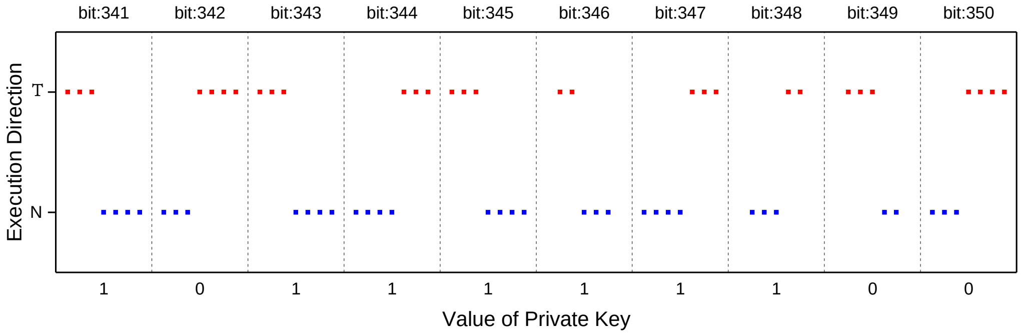


Figure 9: The TBDV-A of the attacker's target branch during the attack against the RSA-2048 decryption algorithm in mbed TLS.

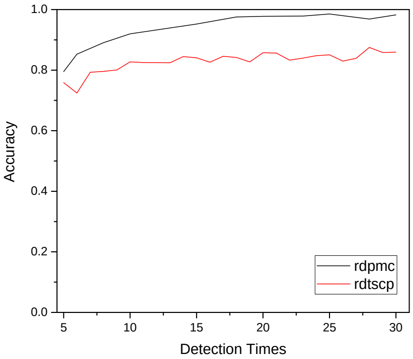


Figure 10: The accuracy of the two measurements with different detection times.

one, in order to provide the attacker with enough time to active the 2-level predictor and initialize the value of the target entry.

The result illustrates that interrupts are conducted precisely in our attack, and the core function `mbedtls_mpi_exp_mod` executes about 140 minutes. For each instruction, it takes nearly 11810 cycles to resume the enclave and execute this instruction in the enclave, while 6 million cycles to handle the incoming interrupt in the non-enclave environment. Figure 9 demonstrates the execution directions of the attacker's target branch, which are affected by ten bits (341-350) of the private key. In order to make this figure be clear, we only show the attacker's execution directions which are affected by the enclave's execution. Take bit 341 as an example. The three red spots mean the attacker's execution directions are "TTT", and the four blue spots mean the attacker's execution directions are "NNNN". According to case 3 in Table 5, we can easily draw the conclusion that now the victim's execution direction is "T" (i.e., taken), and bit 341 of the private key is "1". Figure 9 shows that the victim's target branch direction vector executed is "1,011,111,100". It is worth noting that since each bit's recovery is independent from others', the error of one bit would not affect the recovery of following bits. With the help of post-processing analysis scripts, we can reliably recover the full 2048-bit RSA private key by checking and analyzing the execution vector of the attacker's target branch during the attack.

Besides using PMC, we also try the TSC for measurements. The accuracy of the two measurement methods is displayed in Figure 10. Compared with the PMC measurement, there is a degradation of the attacking accuracy using the TSC measurement. Also, the result illustrates that there is a strong relationship between the detection times and the attacking accuracy. When the detection times is 5, Bluethunder can achieve the accuracy

Table 7: Comparison between the two measurements in SGX, including both the accuracy and the latency.

Detection Times	5	10	15	20	25	30
Rdpmc Error Rate (%)	20.5	8.0	4.8	3.3	1.4	1.3
Rdpmc Latency ($\times 10^4$ cycles)	4.2	7.3	10.4	13.4	16.5	19.5
Rdtscp Error Rate (%)	24.1	17.3	15.9	14.3	14.9	14.1
Rdtscp Latency ($\times 10^4$ cycles)	4.2	7.2	10.3	13.3	16.4	19.2

Table 8: Comparison between Branchscope and Bluethunder (detection times = 6).

–	Bluethunder	Branchscope
Error Bits	163	47
Error Rate	1.63%	0.47%
Attack Overhead	18,532	956,304

of 79.50% (PMC) and 75.90% (TSC). With the increase of the detection times, the error rates of the two measurement methods decrease. Their attacking accuracy reaches even 98.7% (PMC) and 85.9% (TSC) when the repeat time is 30. However, this is at the cost of a longer attacking time. The comparison between the two measurements with different detection times is shown in Table 7. The result demonstrates that with the increase of the detection times, the time cost of the two measurement approaches also increases.

6.3 Bluethunder evaluation when hyper-threading is not supported

Although Bluethunder can achieve excellent performance in SMT environments, this attack is not limited in such scenarios. In other words, Bluethunder can also be conducted in non-SMT environments. This is because the PHT is not flushed during context switches. As a result, the effects on the PHT caused by the enclave process can be detected by the attacker process executing on the same logical core. In order to verify the effectiveness of such a non-SMT Bluethunder attack, we use it to attack the RSA algorithm and try to recover the RSA private key. The result shows that although this non-SMT attack costs much longer time than SMT ones, its accuracy is still pretty high, which can achieve 83.73% in a single run.

6.4 Comparison with Branchscope

We also compare Bluethunder with Branchscope [ERAG+18] by using these two attacks to recover a same 10,000-bit key separately. Since Branchscope does not rely on interrupting the enclave, we did not use SGX-Step. Instead, we assume that the enclave can catch up with the attacker, and we only record the fastest speed Branchscope can achieve. Table 8 gives the comparisons of a Bluethunder attack and a Branchscope attack. Both of them are PHT-based side channels, while they use different components of the BPU to conduct attacks. The result shows that it takes 18,532 cycles for training the predictor before each detection in Bluethunder, which is 1.9% of Branchscope (*i.e.*, 956,304 cycles). In other words, the speed of Bluethunder is about 52× faster than Branchscope. Also, the accuracy of Bluethunder is similar to the accuracy of Branchscope.

7 Limitations and Future work

Limitations. Bluethunder has the following limitations: ① Since Bluethunder relies on interrupting the enclave, several interrupt auditing mitigation techniques, such as

T-SGX [SLKP17], may detect our attack. In machines deployed with these protection methods, the attacker can only force limited number of interrupts in a time period, which may slow down our attack. (2) Although most of the time, the attacker's execution direction sequence follows the "TTTNNN" pattern, it matches other patterns (such as "TTNN") sometimes. However, the desired output can be generated by repeating the experiment.

Future Work. Since the execution of a taken branch updates the branch history record, the state of the current branch history can be used to uncover the victim's control flow accurately. This branch history can be inferred by monitoring the state changes of several 2-level predictor entries. This is because the branch history is one of the elements used for indexing the 2-level predictor, and different branch history leads to the state update of different 2-level predictor entries. By measuring the state changes of possible entries and analyzing which one has been updated, the attacker can eventually deduce the control flow of the victim program, and thus the secret. Because more predictor entries can be detected concurrently in this attack, it can achieve a higher speed than current Bluethunder.

8 Countermeasures

Software Mitigations. One software-based approach against Bluethunder is removing conditional branches in the target enclave program. Choi *et al.* [CKGN01] propose a technique called "if-conversion", which is able to replace conditional branches using conditional instructions such as `CMOV`. Since the conditional branches in an enclave program are removed, our attack cannot be conducted any more. However, this approach is usually algorithm-specific, and applying it to general applications is challenging.

A more practical countermeasure is auditing [GYCH18], which requires no changes in processor designs. The main idea of this method is to analyze the abnormal behaviors of running processes. Since Bluethunder attacks introduce a large number of mispredictions to the target system, by monitoring the misprediction number in the target systems, these attacks can be detected. Also, two recent works, T-SGX [SLKP17] and Déjà Vu [CZRZ17], proposed that detecting the frequent interrupt of the victim enclave can also detect side-channel attacks. Since Bluethunder relies on frequently interrupting the enclave to construct PHT entry collisions, these two methods are also valuable to defend against our attacks. However, for both of these two detecting approaches, selecting the threshold used for differentiating between benign programs and adverse programs is difficult. This is because a large number of mispredictions or interrupts can be caused by many benign programs, just as Bluethunder does.

Hardware Mitigations. In order to prevent Bluethunder, one approach is to randomize the PHT indexing logic. The root cause of Bluethunder is the 2-level predictor entry collisions between enclave and non-enclave processes. To make it harder for the attacker to construct entry collisions, we may change the PHT indexing logic randomly, just like randomizing the mapping of caches [WL07]. Considering that the attacker may uncover the randomization indexing logic if we randomize only once, periodically randomization can be used. This solution can keep the high usage of the PHT, since PHT entries are still shared between enclave and non-enclave processes. However, extra hardware components (such as buffers) may be required if we randomize the indexing algorithms multiple times, in order to record the current indexing relationship between the branches and their PHT entries.

Another solution is to prevent predicting sensitive branches. A naive way is to disable the branch prediction. However, this may cause severe penalty to the processor's performance. Branchscope [ERAG+18] proposes an optimization method, which enables software developers to mark the branches which relate to the sensitive information. When executing these marked instructions, the CPU will avoid predicting them. This method can protect the victim process from side channel attacks, as well as cause less performance overhead than just disabling the BPU. However, this solution cannot protect against covert channel attacks.

Other possible hardware mitigations also exist. For example, we may double the BPU logic, one for enclave processes and the other one for non-enclave processes. However, this mitigation requires a more complex BPU design and increases the costs of manufacturing processors. Another solution may flush the PHT states whenever the context switches to or from enclave mode, but this mitigation may lengthen the training time of the BPU, especially when context switches happen frequently.

9 Related Work

SGX Side Channels. Researchers have shown that Intel SGX is vulnerable to controlled channel attacks [VBWK$^+$17, XCP15], cache-based attacks [BMD$^+$17, GESM17, HCP17, SWG$^+$17] and BPU-based attacks [LSG$^+$17, ERAG$^+$18, CCX$^+$19]. ① According to Oleksii Oleksenko *et al.* [OTK$^+$18], page table based attacks can be classified into two categories: page fault based [XCP15, SCNS16] and page bit based [WCP$^+$17, VBWK$^+$17]. Xu *et al.* [XCP15] and Shinde *et al.* [SCNS16] demonstrate page fault side-channel attacks on SGX, which leak secrets inside enclaves at page size granularity by marking the enclave pages as non-present. Afterward, researchers find that these controlled channel attacks against SGX can also be carried out without page faults. By observing the accessed and dirty bits of the page table, Wang *et al.* [WCP$^+$17] and Jo Van Bulck *et al.* [VBWK$^+$17] conduct page bit attacks and leak enclave's secrets successfully. ② cache-based attacks [WCP$^+$17, GESM17, BMD$^+$17, SWG$^+$17] are also leveraged to conduct high-resolution SGX attacks. By using "Prime+Probe" approach [OST06], these attacks are able to learn which memory locations are accessed by the enclave and extract enclave's secrets at fine granularity. ③ BPU-based attacks against SGX can be conducted in two ways. One way is to leverage the speculative prediction results made by the BPU. For example, SgxPectre [CCX$^+$19] exploits the famous Spectre bug [KHF$^+$19] in an SGX environment and subverts the confidentiality and integrity of SGX enclaves. Another way to attack enclaves is to exploit the current state of the BPU. For example, Lee *et al.* [LSG$^+$17] conduct an attack called "branch shadowing", which infers the fine-grained control flow of an enclave by abusing BTB entry collisions between enclaves and non-enclaves. Another example is Branchscope [ERAG$^+$18], which leaks the sensitive data in the target program by abusing the bimodal predictor in the BPU, whose core component is a 2-bit saturating counter. However, Branchscope has a limitation that it requires nearly 100,000 branches to initialize the BPU state before each detection during the attack, which slows down the attacking speed significantly. Branchscope is closely related to our attack. The difference is that we reverse-engineer and abuse the 2-level directional predictor for attack, which is a completely different prediction component in the BPU.

Micro-architectural Side Channels. Micro-architectural side-channel attacks have been investigated for decades since first mentioned by Lampson [Lam73] in 1973. Usually, micro-architectural side channel exploits shared resources in processors, such as cache [DKPT17, Per05, GMWM16, IES15, GBK11, KAGPJ16, LYG$^+$15, OST06, YF14, ZJRR12], branch prediction unit [ERAG$^+$18, KHF$^+$19, MR18, LSG$^+$17, EPAG16, CCX$^+$19], out-of-order execution [VBMW$^+$18, LSG$^+$18] and floating-point unit [KS17] to perform specific attacks. Several attacks have been implemented through side channels, such as performing keylogging [GSM15], leaking private keys [VBWK$^+$17, YF14, BB05, AS08, KAGPJ16, LYG$^+$15], inferring sensitive user data [ZWB$^+$18] and breaking the encryption of TLS [IIES15]. Ge *et al.* [GYCH18] conduct an excellent survey of these micro-architectural timing side-channel attacks.

10 Conclusion

In this paper, we propose a new PHT-based attack against SGX named Bluethunder, which leverages the 2-level directional predictor as a side channel. By exploiting hashing collisions in the predictor, this attack can precisely identify fine-grained (i.e., instruction-level) control flows inside an enclave. We implement Bluethunder and evaluate this attack on two case studies: attacking the *vfprintf* function in the Intel SGX SDK and RSA decryption algorithm in mbed TLS. Both of the attacks show that even though several defensive techniques are used, Bluethunder attacks are indeed practical and pose a serious threat on SGX. In particular, our implementation is able to derive the RSA private key with 96.76% accuracy in a single run, whose speed outperforms the latest PHT-based side channel attack (Branchscope) by 52×. We stress that hardware-based countermeasures against Bluethunder attacks (*i.e.*, flushing the PHT during enclave mode switch) need to be taken, in order to prevent large-scale exploitation of Bluethunder.

Acknowledgments

We are grateful to Billy Brumley, Yeping He, Qiusong Yang, Shan Zhao and Yiwei Ci for their helpful comments. This work was supported by Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDA-Y01-01. Wenhao Wang was supported by National Natural Science Foundation of China (Grant No.61802397).

References

[ABPK07] Giovanni Agosta, Luca Breveglieri, Gerardo Pelosi, and Israel Koren. Countermeasures against branch target buffer attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 75–79. IEEE, 2007.

[ABuH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[ARM08] ARM. Building a secure system using TrustZone technology. https://developer.arm.com/docs/genc009492/latest/trustzone-hardware-architecture/processor-architecture/processor-architecture/multiprocessor-systems-with-the-security-extensions, Dec 2008.

[AS08] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Cryptographers' Track at the RSA Conference*, pages 256–273. Springer, 2008.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[BWG+16] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[CCX+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*, 2019.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[CKGN01] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the Intel® Itanium™ processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 182–191. IEEE Computer Society, 2001.

[CZRZ17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18. ACM, 2017.

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[DSC+15] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling stronger privacy in MapReduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 447–462, 2015.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[ERAG+18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, volume 53, pages 693–707. ACM, 2018.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy (S&P'11)*, pages 490–505. IEEE, 2011.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[Gre12] J. Greene. Intel trusted execution technology. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html, 2012.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[H$^+$18] Jann Horn et al. Reading privileged memory with a side-channel. *Project Zero*, 39, 2018.

[HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 299–312, 2017.

[HLLP18] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on Intel SGX using control flow randomization. In *3rd Workshop on System Software for Trusted Execution (SysTEX)*, 2018.

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*, pages 591–604. IEEE, 2015.

[IIES15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96. ACM, 2015.

[Int14] Intel. Software guard extensions programming reference, revision 2. http://kib.kiev.ua/x86docs/SDMs/329298-001.pdf, 2014.

[Int18] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, Jan 2018.

[KAGPJ16] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, page 72. ACM, 2016.

[KHF$^+$19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[KS17] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 69–81, 2017.

[Lam73] Butler Lampson. A note on the confinement problem. *Communications of the Acm*, 16(10):613–615, 1973.

[LMB$^+$] Ben Lee, Alexey Malishevsky, Douglas Beck, Andreas Schmid, and Eric Landry. Dynamic branch prediction. http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/.

[LSG$^+$17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017.

[LSG$^+$18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[LYG$^+$15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*, pages 605–622. IEEE, 2015.

[MAB$^+$13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *The Second Workshop on Hardware and Architectural Support for Security and Privacy*, 10(1), 2013.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[MWES19] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. In *International Journal of Parallel Programming*, volume 47, pages 538–570. Springer, 2019.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[OTK$^+$18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, pages 227–240, 2018.

[Per05] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[SCF$^+$15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*, pages 38–54. IEEE, 2015.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[SLK$^+$17] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the Symposium on Network and Distributed System Security*, 2017.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Symposium on Network and Distributed System Security*, 2017.

[SWG$^+$17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[VBMW$^+$18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[VBPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, page 4. ACM, 2017.

[VBWK$^+$17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.

[WCP$^+$17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.

[WL07] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494–505, 2007.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*, pages 640–656. IEEE, 2015.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[YP91] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

[ZWB+18] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. OS-level side channels without procfs: Exploring cross-app information leakage on iOS. In *Proceedings of the Symposium on Network and Distributed System Security*, 2018.

A Sliding-Window Exponentiation

The sliding-window exponentiation algorithm implemented in mbed TLS is listed below. This function has two conditional branches (marked with *), whose execution directions depend on the value of ei.

```
/* Sliding-window exponentiation:
  X = A^E mod N */
int mbedtls_mpi_exp_mod(mbedtls_mpi *X, const mbedtls_mpi *A, const
  mbedtls_mpi *E, const mbedtls_mpi *N, mbedtls_mpi *_RR) {
 ...
 state = 0;
 while (1) {
  ...
  // i-th bit of exponent
  ei = (E->p[nblimbs] >> bufsize) & 1;

  /* skip leading 0s */
* if (ei == 0 && state == 0)
   continue;

* if (ei == 0 && state == 1)
  {
    /* out of window, square X */
    mbedTLS_MPI_CHK(mpi_montmul(X, X, N, mm, &T));
    continue;
  }

  state = 2; nbits++;
  wbits |= (ei << (wsize-nbits));

  if (nbits == wsize) {
   for (i = 0; i < wsize; i++)
    mpi_montmul(X, X, N, mm, &T);
```

```

      mpi_montmul(X, &W[wbits], N, mm, &T);
      state--; nbits = wbits = 0;
    }
  }
  ...
}
```