BranchScope: A New Side-Channel Attack on Directional Branch Predictor

Dmitry Evtyushkin
College of William and Mary
devtyushkin@wm.edu

Ryan Riley
Carnegie Mellon University in Qatar
rileyrd@cmu.edu

Nael Abu-Ghazaleh
University of California Riverside
naelag@ucr.edu

Dmitry Ponomarev
Binghamton University
dponomar@binghamton.edu

Abstract

We present *BranchScope* — a new side-channel attack where the attacker infers the direction of an arbitrary conditional branch instruction in a victim program by manipulating the shared directional branch predictor. The directional component of the branch predictor stores the prediction on a given branch (taken or not-taken) and is a different component from the branch target buffer (BTB) attacked by previous work. *BranchScope* is the first fine-grained attack on the directional branch predictor, expanding our understanding of the side channel vulnerability of the branch prediction unit. Our attack targets complex hybrid branch predictors with unknown organization. We demonstrate how an attacker can force these predictors to switch to a simple 1-level mode to simplify the direction recovery. We carry out *BranchScope* on several recent Intel CPUs and also demonstrate the attack against an SGX enclave.

CCS Concepts • **Security and privacy → Side-channel analysis and countermeasures**; **Hardware reverse engineering**;

Keywords Branch Predictor, Attack, Side-channel, SGX, Microarchitecture Security, Timing Attacks, Performance Counters

ACM Reference Format:
Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173204

1 Introduction

Modern microprocessors rely on branch prediction units (BPUs) to sustain uninterrupted instruction delivery to the execution pipeline across conditional branches. When multiple processes execute on the same physical core, they share a single BPU. While attractive from utilization and complexity considerations, the sharing potentially opens the door for an attacker to manipulate the shared BPU state, create a side-channel, and derive a direction or target of a branch instruction executed by a victim process. Such leakage can compromise sensitive data. For example, when a branch instruction is conditioned on a bit of a secret key, the key bits are leaked directly. This occurs in implementations of exponentiation algorithms [13, 32] and other key mathematical operations [3] of modern cryptographic schemes. The attacker may also change the predictor state, changing its behavior in the victim.

On modern microprocessors, the BPU is composed of two structures: the branch target buffer (BTB) and the directional predictor. Previous work has specifically targeted the BTB to create side channels [1, 3, 21, 35]. In the BTB, the target of a conditional branch is updated only when the branch is taken; this can be exploited to detect whether or not a particular victim branch is taken. The first attack in this area proposed several BTB-based attacks that are based on filling the BTB by the attacker, causing the eviction of entries belonging to the victim. By observing the timing of future accesses [3], the attacker can infer new branches executed by the victim. We describe those attacks and their limitations in the related work section. In other work [21], we recently proposed a side-channel attack on the BTB that creates BTB collisions between the victim and the attacker processes, thus allowing the attacker to discover the location of a particular victim's branch instruction in the address space, bypassing address space layout randomization. Lee et al. [35] built on that work by exploiting the BTB collisions to also discover the direction of the victim's branch instructions. They demonstrated the attack in kernel space against Intel SGX enclaves.

In this paper, we propose a new micro-architectural side-channel attack, which we call *BranchScope*, that targets the *directional predictor* as the source of information leakage. To the best of our knowledge, *BranchScope* is the first attack exploiting the directional predictor structure, showing that BPUs can be vulnerable even if the BTB is protected. *BranchScope* works by forcing collisions between the attacker and selected victim branches and exploiting these collisions to infer information about the victim branch. This attack has new challenges not present in a BTB attack. In order to achieve collisions, we must overcome the unpredictability of the complex hybrid prediction mechanisms used in modern CPUs. *BranchScope* overcomes this by generating branch patterns that force the branch predictor to select the local one-level prediction even when complex multi-level predictors are present in the processor. Second, after collisions are reliably created, the victim's branch direction can be robustly disclosed by an attacker executing a pair of branches with predefined outcomes, measuring the prediction accuracy of these branches, and correlating this information to the predictor state and thus to the direction of the victim's branch.

We demonstrate *BranchScope* on three recent Intel x86_64 processors — Sandy Bridge, Haswell and Skylake. To perform *BranchScope*, the attacker does not need to reverse-engineer the details of the branch predictor operation, and only needs to perform simple manipulations with the prediction state machines from the user space. We also demonstrate how *BranchScope* can be extended to attack SGX enclaves even if recently-proposed protections are implemented. We show that *BranchScope* can be performed across hyperthreaded cores, advancing previously demonstrated BTB-based attacks which leaked information only between processes scheduled on the same virtual core [21]. This capability relaxes the attacker's process scheduling constraints, allowing a more flexible attack. Finally, we describe countermeasures to prevent the *BranchScope* attack in future systems.

The recent Meltdown [36] and Spectre [34] attacks demonstrated the vulnerability of speculative execution to side-channel attacks, directly impacting the security of current systems and leading to data exfiltration. Branch predictors are critical to these attacks since the attacker must mistrain, or even directly pollute (known as a Branch poisoning attack) the branch predictor to force the predictor to guess the address of the victim selected vulnerable code. The branch poisoning attack presented in Spectre is based on the same basic principle as *BranchScope* — exploiting collisions between different branch instructions in the branch predictor data structures. In this context, we believe that *BranchScope* can provide additional tools for attackers to use speculation to perform more advanced and flexible attacks. As the community considers defenses against these attacks, the vulnerability outlined in *BranchScope* must also be addressed.

In summary, the main contributions and the key results of this paper are:

- We propose *BranchScope* — the first side-channel attack explicitly targeted at extracting sensitive information through the directional branch predictor (as opposed to existing work targeting the Branch Target Buffer). *BranchScope* is not affected by defenses against BTB-based attacks.
- We demonstrate that *BranchScope* works reliably and efficiently *from user space* across three generations of Intel processors in the presence of system noise, with an error rate of less than 1%.
- We show that *BranchScope* can be naturally extended to attack SGX enclaves with even lower error rates than in traditional systems.
- We describe both hardware and software countermeasures to mitigate *BranchScope*, providing branch prediction units that are secure to side channel attacks.

2 Background: Branch Predictor Unit

Modern branch predictors [15, 31, 41, 43, 50] are typically implemented as a composition of a simple one-level bimodal predictor indexed directly by the program counter (we refer to it as the *1-level predictor* [49]), and a gshare-style 2-level predictor [57]. The gshare-like predictor exploits the observation that the branch outcome depends on the results of recent branches, and not only on the address of the branch. A selector table indexed by the branch address identifies which predictor is likely to perform better for a particular branch based on the previous behavior of the predictors. This design combines the best features of both component predictors.

Figure 1 illustrates one possible design of such a hybrid predictor. The 1-level predictor stores its history in the form of a 2-bit saturating counter in a pattern history table (PHT). The gshare predictor has a more complex indexing scheme that combines the program counter with the global history register (GHR). The GHR records the outcomes of the last several branches executed by the program. The branch history information is also stored in the PHT using a 2-bit saturating counter; the only difference between the two predictors is how the PHT is indexed.

If the branch is predicted to be taken, the target address of the branch is obtained from a structure called the Branch Target Buffer (BTB), which is a simple direct mapped cache of addresses that stores the last target address of a branch that maps to each BTB entry. Published side channel attacks (described in Section 11) on the BPU have all targeted the BTB. In contrast, *BranchScope* targets the direction prediction unit of the BPU.

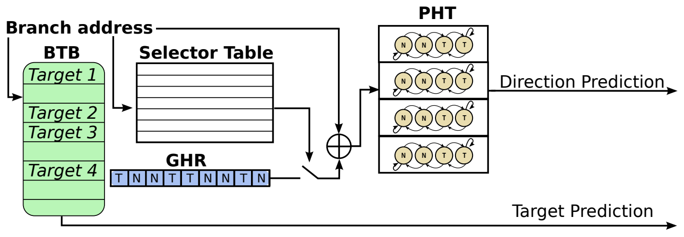


Figure 1. A Combined Branch Predictor

3 Threat Model and Attacker Capabilities

Our attack assumes the existence of a *victim* and a *spy* programs. The *victim* program contains secret information that the *spy* program is trying to infer, without having the authority to access this information directly. The threat model makes three primary assumptions:

- Co-residency on the same physical core: We assume that the *victim* and the *spy* programs are running on the same physical core since the BPU is shared at the virtual core level. Prior work [21] has shown possible techniques for forcing such co-residency.
- Victim slowdown: To perform a high-resolution *BranchScope* attack, where we are able to detect the behavior of an individual execution of a branch, the victim process needs to be slowed down. This slowdown is a common requirement of high-resolution side-channel attacks [26, 33]. Slowing down the victim is an orthogonal issue that can be accomplished by a variety of means, for example by exploiting the Linux scheduler as proposed by Gullasch et al. [26] or performing microarchitectural performance degradation attack [4]. Importantly, in a threat model where a malicious OS is attacking an SGX compartment, the OS can control the scheduling at fine-grain to slow down the victim.
- Triggering victim code execution: We assume that the attacker can initiate code execution of the victim process such that it can force the victim to execute the targeted vulnerable operation at any time. This assumption holds for many applications that are triggered by external input. For example, consider a server that sends out encrypted data; the attacker can trigger a response from this server by sending a request to it. We do not assume that the attacker can observe the contents of the response from the victim.

We believe that these three assumptions hold in a large number of realistic attack scenarios making *BranchScope* a serious threat to modern systems, on par with other side-channel attacks. Later in the paper, we support this claim by demonstrating *BranchScope* on a real SGX-based platform.

4 BranchScope Attack Overview

In this section, we present the an overview of *BranchScope*. We start with background information and a high-level overview of the attack, and then move to the details.

In general, the attack proceeds as follows:

- *Stage 1: Prime the PHT entry.* In this stage, the attacker process primes a targeted PHT entry into a specified state. This priming is accomplished by executing a carefully-selected randomized block of branch instructions. This block is generated one-time, a-priori by the attacker.
- *Stage 2: Victim execution.* Next, the attacker initiates the execution of a branch it intends to monitor within the victim process and waits until the PHT state is changed by the victim's activity.
- *Stage 3: Probe the PHT entry.* Finally, the attacker executes more branch instructions targeting the same PHT entry as the victim while timing them to observe their prediction outcomes. The attacker correlates the prediction outcomes with the state of the PHT to identify the direction of the victim's branch.

The attacker must be able to cause collisions between its branches and the branches of the victim process in the PHT. These collisions, given knowledge of the operation of the predictor, allow the attacker to uncover the direction of the victim's branch. Specifically, by observing the impact of that branch (executed in *stage 2* above) on the prediction accuracy of an attacker's probing branches executed in *stage 3*. If the PHT indexing is strictly determined by the instruction address (as in the 1-level predictor), creating collisions in the PHT between the branches of two processes is straightforward, since the virtual addresses of victim's code are typically not a secret. If address space layout randomization (ASLR) is used to randomize code locations, the attacker can de-randomize using data disclosure [48], or side channel attacks on ASLR [21, 24, 28, 30, 54].

BranchScope requires the following two abilities:

- **Establishing Collisions**. The attack relies on generating collisions within the predictor. Creating collisions is greatly simplified if the predictor in use is the simply indexed 1-level predictor instead of the more complex gshare-like predictor. The attack must force both the attack code and the victim code to use the 1-level predictor.
- **Prime Probe Strategy**. After the attacker forces a collision in the PHT, she still needs to be able to interpret the state of the PHT in order to determine the direction of the victim's branch. Therefore, we need to understand how to prime a particular PHT entry into a desired starting state in *stage 1*. This starting state must enable us to correlate some observable behavior of a probe operation from the attacker in *stage 3* with the direction of the victim's branch.

In the next two sections, we explain how the attacker achieves these two goals.

5 Attack Capability I: Establishing collisions by controlling selection logic

BranchScope's strategy to establish collisions is to force both the spy code and victim code to use the 1-level predictor, which makes the PHT entry used a simple function of the branch address. We start with an experiment that demonstrates how the selection logic works, and then use these observations to force the use of the 1-level predictor for our target branches. We performed these experiments on three recent Intel processors: i5-6200U based on Skylake microarchitecture, i7-4800MQ based on Haswell microarchitecture and i7-2600 based on Sandy Bridge Microarchitecture.

5.1 Understanding the Selection Logic

The selection logic within the hardware attempts to choose the predictor that is more accurate. To gain insight into how this selection operates as the 2-level predictor learns a branch execution pattern we conduct the following experiment. An irregular but repeating sequence of branch outcomes from the same branch instruction cannot be predicted accurately by the simple 1-level predictor since the branch outcome is not a function of the two preceding branches. However, such a sequence is predictable by a 2-level predictor once its history is initialized. To understand how quickly the learning process proceeds and the selection of the gshare-like predictor over the 1-level predictor occurs, we performed the following experiment on two recent Intel processors: i5-6200U based on Skylake microarchitecture, and i7-2600 based on Haswell microarchitecture.

- We initialize an array of 10 bits to a randomly selected state. This bit pattern serves to control whether the branch is taken in our experiment.
- We execute a single branch instruction conditional on the array bits, once for each bit. We repeat the series of branches 20 times in a row and record the total number of incorrect predictions in this branch sequence for each of the iterations. We use hardware performance counters to track prediction, enabling accurate measurement with a resolution of a single branch misprediction.

An 1-level predictor will not be able to predict better than 50% on average, but a gshare style predictor should eventually learn the pattern.

The prediction accuracy from this experiment (averaged over multiple runs) is presented in Figure 2. As seen from the Figure, as the first iteration is executed, the misprediction rate is about 50% (five out of ten branches are mispredicted). This result is expected, since in this stage the 2-level predictor does not have any prior state, while the 1-level predictor is not capable of predicting such patterns in principle. As the branch pattern repeatedly executes, the branch misprediction rate decreases, as more history is accumulated by the 2-level predictor structures. When the branch pattern is repeated about 5 − 7 times, the predictor accuracy approaches 100% and stays at that value. Both CPUs demonstrated similar behavior, with the Skylake processor learning the pattern slightly faster.

These results indicate that eventually (after 5-7 iterations, or 50-70 executions of the branch) for this pattern, the 2-level predictor is used exclusively. However, when the branch is first encountered, either the 1-level or 2-level predictor is used but is not predicting effectively.

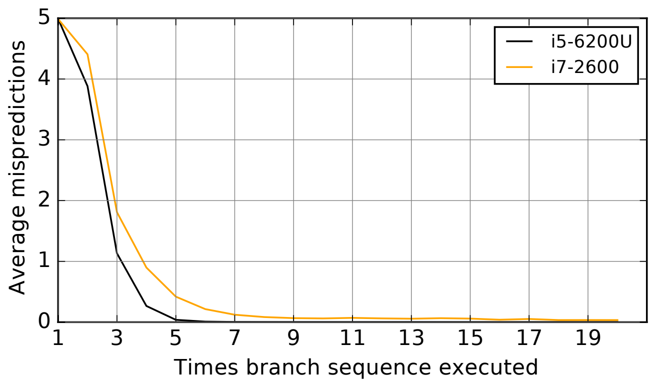


Figure 2. Average number of mispredictions for a sequence of branch instructions in individual runs

Next we focus on the initial behavior of the predictor (early iterations in Figure 2). We conjecture that *for new branches whose information is not stored in the predictor history, the 1-level predictor is used.* This hypothesis intuitively makes sense since the 2-level predictor takes a longer time to learn the branch pattern compared to a simple 1-level predictor. For example, if an "almost-always-taken" branch at the end of the loop is executed, the 1-level predictor will converge to the "strongly taken" state after 2-3 executions. On the other hand, the 2-level predictor will use different history register values and thus different PHT entries for every instance of the branch, making it significantly slower to converge. We carried out experiments to validate the use of 1-level predictor for branches with no history and found that it holds for all three Intel platforms. We can detect the use of the 1-level predictor when collisions can be established simply based on the branch addresses.

5.2 Forcing usage of the 1-level predictor

We will now discuss how to use the knowledge gleaned from our previous experiment in order to force the hardware to choose the 1-level predictor for both the attacker and victim code.

Attacker code We use the observation that new branches use the 1-level predictor directly in the attacker code to force the use of the 1-level predictor: we cycle through a number of branches placed at addresses that collide with the victim branch (if that also uses the 1-level predictor) in the branch predictor, such that at any time the attack branch being used does not exist in the BPU, forcing the unit to use the 1-level predictor.

Victim code The more difficult task is to force the victim code to use the 1-level predictor; the victim code is not under the control of the attacker. To force the BPU to use the 1-level predictor for the targeted victim branch, the attacker needs to accomplish one of two goals: 1) ensure that the branches used by the attack have not been recently encountered, thus starting the prediction for these branches from the 1-level mode; 2) make the 2-level predictor inaccurate and prolong its training time, forcing the selector to choose the 1-level mode at least for several branches. Thus, the attacker must ensure that at least one of these two properties (if not both) hold to force the victim code to use the 1-level predictor.

We accomplish this goal by developing a sequence of branch-intensive code that the attacker executes to drive the BPU to a state that lowers the 2-level predictor accuracy and potentially replaces the victim branches. As a result of executing this sequence, the victim code will use the 1-level predictor when it executes its branch, enabling us to achieve collisions. This code serves another critical function: it forces the PHT entries to a desired state that enables us to reliably detect the branch outcome per the operation of the prediction FSM (reverse engineered in the next section). To maximize its efficiency, the randomizing code has to have two properties. First, the executed branches must not contain any regular patterns predictable by the 2-level predictor. To this end, the directions of branches in the code are randomly picked with no inter-branch dependencies. Second, the code must affect a large number of entries inside the PHT. This is accomplished by executing a large number of branch instructions and randomizing memory locations of the these instructions by either placing or not placing a `NOP` instruction between them. The outcome patterns are randomized only once (when the block is generated) and are not re-randomized during execution. These manipulations with the branch predictor must be performed before the victim executes the target branch (during *stage 1* of the attack).

The total number of branch instructions needed to be executed in this manner depends on the size of BPU's internal data structures on a particular CPU. We experimentally discovered that executing 100,000 branch instructions is sufficient to randomize the state of most PHT entries and to effectively disable the 2-level predictor. An example of such a code is presented in Listing 1. Reducing the size of this code is a topic of future research; for example, if we focus only on evicting a particular branch, we may be able to come up with a shorter sequence of branches that map to the same PHT and replace that entry.

```
randomize_pht:
cmp %rcx, %rcx;
je .L0; nop; .L0: jne .L1; nop; .L1: je .L2;
............
.L99998: je .L99999; nop; .L99999: nop;
```

Listing 1. Pseudo-code of the spy program. `je` and `jne` are randomly selected, achieving random pattern of taken and not-taken branches

6 Attack Capability II: Prime Probe Strategy

Having developed a reliable approach to establish collisions between the attacker and the victim, the next task is to understand the operation of the prediction logic to develop a prime-probe strategy that enables us to infer the victim branch direction. The attack should prime the PHT entry before the victim branch and probe it after the branch to infer the branch direction. At the core of the predictor structures are a set of Finite State Machines (FSM) that produce the prediction decision. Typically, one of these FSMs is maintained for every entry in the PHT table. Both the 1-level and 2-level predictor in a combined predictor structure use the same FSM logic and possibly even the same PHT differing only in the indexing function to the PHT.

6.1 Understanding the prediction logic

We begin with a hypothesis that each PHT entry consists of a textbook two-bit saturating counter FSM with four states: strongly taken (ST), weakly taken (WT), weakly not taken (WN) and strongly not taken (SN). We generate several branch instructions targeting the same PHT and observe the resulting predictions (Figure 3). We note that the actual implementation of the state machine on these processors is unknown and can be more complex. For example, the implementation may include additional state transfers and may rely on inputs from other CPU data structures. However, we discovered that the behavior of the branch predictors on the processors is consistent with this simple textbook model.

Consider the following three steps in which *a single* test branch with no previous history is executed within one process. This essentially mimics our three attack stages, but within the same process. First, we execute the aforementioned branch instruction three times to *prime* the corresponding PHT entry by placing it into one of the strong states (either ST or SN). Second, we execute the same branch one more time with both taken and not-taken outcomes (in two separate trials). This is called the *target* stage, similar to stage 2 of the attack. Finally, we execute the same branch two more times detecting mispredictions (we call it the *probing* stage, similar to stage 3 of the attack). During this stage, we also record the prediction accuracy for each of the two probing branches.

Table 1 depicts our observations for all possible cases. For example, consider the case when the branch in question was executed three times with *not-taken* outcome (the prime stage). The expectation is that this activity will shift the FSM to the SN state. When the branch is executed once with *taken* outcome in the target stage, the FSM is switched to the WN state. Finally, the branch is executed two more times with *taken* outcome during the probing stage. In this case, the first branch executed in the probing stage will be *mispredicted*, while the second branch will be predicted *correctly*. In contrast, if the branch in the target stage was *not-taken*, the FSM would stay in the SN state. In that case, both branches in the probe stage would be *mispredicted*. Therefore, by observing the difference in the prediction outcomes for the two branches in the probing stage, the attacker can determine the direction of the victim's branch in the target stage. This is the key observation exploited by *BranchScope*.

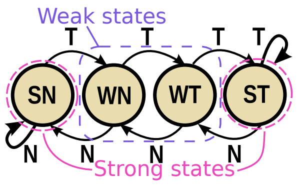


Figure 3. Two-bit FSM with four states: **SN** – strongly not taken, **WN** – weakly not taken, **WT** – weakly taken, **ST** – strongly taken

Prime	State after Prime	Target	State after Target	Probe	Observation
TTT	ST	T	ST	TT	HH
TTT	ST	T	ST	NN	MM
TTT	ST	N	WT	TT	HH
TTT	ST	N	WT	NN	MH[1]
NNN	SN	T	WN	TT	MH
NNN	SN	T	WN	NN	HH
NNN	SN	N	SN	TT	MM
NNN	SN	N	SN	NN	HH

Table 1. FSM transitions for a single PHT entry. The entry is set into one of the strong states in the prime stage, a branch is executed once in the target stage, and the resulting state is recorded using performance counters in the probing stage. MM – two mispredictions in the probing stage, MH – misprediction followed by a hit (correct prediction) in the probing stage

According to Table 1, it is possible to determine a PHT state by performing two individual probes with the same branch instruction, with taken and with not-taken outcomes. For example, assume that the observed prediction pattern of the two probing branches is two hits (HH) when probing with two taken branches (TT) and two mispredictions (MM) when probing with two non-taken branches (NN). In this case, we can conclude that the PHT entry in question is located in the strongly taken (ST) state (rows 1 and 2 in Table 1). Note that a peculiarity that we discovered in Skylake processors makes the strongly taken (ST) and weakly taken (WT) states indistinguishable on that processor. However, this limitation does not prevent recognizing the other states. It also does not prevent *BranchScope* attack on Skylake since the attacker can always pick a PHT randomization code that places the target PHT entry into a state without such ambiguity.

6.2 Setting and probing predictor state

Executing the block of random branch instructions (Listing 1) allows the attacker to force the victim code to use 1-level predictor, as we discussed in previous section. However, a carefully selected randomization code can also serve to prime the targeted PHT entry into a state required by the attacker.

To better understand the nature of PHT randomization and the effects of system noise, and select appropriate randomization code for our attack to reliably place the PHT entries into the attacker-specified state, we performed an experiment composed of 10 000 iterations. In each iteration, we generated a new randomization code block and then performed the following activities 1 000 times: a) executed the generated block of branches, b) performed a PHT probing operation for a fixed address. For probing operation, we considered two scenarios: 1) two taken branches, and 2) two non-taken branches. For every iteration, we collected 1 000 measurements for each probing pattern and determined statistical distribution of the PHT states.

To collect the statistical profiles, we only accounted for the iterations that produced stable PHT states. We assumed that the results are stable if the most frequent prediction pattern in *both* variations of the probing code occurs more than 85% of the time (out of 1 000 executions). Again, the state of a PHT entry is not always the same after executing the same randomization code due to the various system effects. The results show that most randomly generated blocks of branch code produce stable PHT state, the distribution of patterns for both variations of probing code (along with cutoff point) is shown in Figure 4a. Each point on the graph represents the percentage of the most frequent prediction pattern of the probing code for each PHT randomization code block (each iteration of the experiment). Each iteration is depicted by a point on the graph, where the x-axis represents the percentage of the most frequent prediction pattern for the TT probing code, while the y-axis represents the most frequent prediction pattern for the NN probing code. As seen from the graph, 83% of all randomized code blocks result in stable dominant prediction patterns for both probing code sequences. The stable patterns can be translated into one of the FSM states of the PHT entry targeted by the probing branch address using Table 1. However, when the prediction

[1]MH is observed on Haswell and Sandy Bridge, while MM is observed on Skylake

patterns are not stable (the most frequent pattern appears less than 85% of times for either of the probing combinations) we assume that this particular iteration of the experiment is too noisy due to the various system-level effects on the predictor, such as the invocation of the 2-level predictor, or a different PHT state inherited by the randomizing code due to some intermittent processing). In this case, we consider the measurements to be unreliable and too noisy, and drop this particular iteration from our collected statistics. In the following piechart, we classify these cases as unknown.

Figure 4b depicts the distribution of the decoded PHT states for the PHT entry targeted by the probing branches. In addition to the four standard stable states with their distinct patterns, we observed another pattern with a stable behavior. This additional pattern consists of two correct predictions (HH) in the probing code regardless of the type of probing. Such a pattern indicates that the PHT randomization code has no effect on the target branch and the BPU can always produce a correct prediction. This likely indicates 2-level predictor is used for this branch. We refer to this case as *dirty*.

To implement *BranchScope*, the attacker needs to ensure that at the time of victim's execution of *stage 2* of the attack, the PHT entry corresponding to the target branch is in the state desired by the attacker. The attacker cannot simply set this state at will, because she needs to execute the PHT randomization code at the end of *stage 1*, which resets the entire PHT. However, the attacker can randomly generate the blocks of code that randomize the PHT until the block is found that leaves the target PHT entry in the desired state, using the analysis above. Finding the appropriate randomization code is a one-time effort by the attacker and can be performed during the pre-attack stage. This is a key element of *BranchScope*.

We can now create a mapping between the predictor behavior and the direction of the branch in the target stage. The main conclusion is that it is possible for a process to determine the direction of the *target* branch only by examining whether the two probing branches were correctly predicted or not.

6.3 Discussion and Extensions

Knowing the states of PHT entries associated with different memory addresses potentially allows the attacker to spy on multiple branch instructions in victim process in a single episode of execution. To pursue such an aggressive attack, the adversary needs to understand some details of the PHT organization. To this end, we performed the following experiment.

First, we execute the randomization code to set the initial state of the PHTs. Next, for a given range of virtual addresses, we place a branch instruction at each address and execute these branches. Finally, we evaluate the state of the PHT entry corresponding to the virtual address at which each

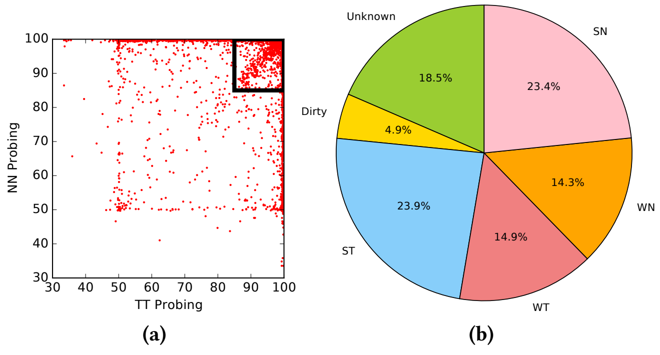


Figure 4. Distribution of PHT States

branch instructions was placed. The PHT state was determined in a similar way as in our previous experiment, using the dictionary that translates the prediction outcomes of the probing code to the PHT state. This experiment allows us to probe the entire PHT. Figure 5a demonstrates the results when the branch instruction was placed in the range of virtual addresses from `0x300000` to `0x30010f`. As can be seen from the figure, two adjacent addresses can be in different states. This experiment shows that the granularity of PHT's indexing function is a single byte.

The PHT probing data can be used to discover the size of PHT. Assuming the PHT index is calculated with a simple modulo operation, the task of reverse-engineering the PHT size is trivial. The observed patterns repeat after each N addresses, where N is the size of the PHT. We use this insight to discover the PHT size on our experimental machine. All measured states are presented as a vector of states:

$$V = [v_0, \ldots, v_n] \mid v_i \in \{ST, WT, WN, SN, Unk., Dirty\} \quad (1)$$

The vector V can be split into equal-length subvectors of size w. We refer to w as the window size. Then, S_w is the set containing all subvectors of size w:

$$S_w = \left\{ [v_{zw}, \ldots, v_{(z+1)w-1}] \mid 0 \leq z < \frac{|V|}{w} \right\} \quad (2)$$

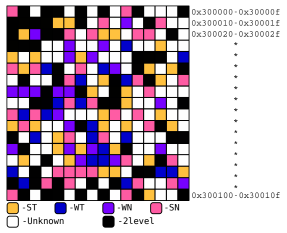
The function $H(w)$ represents the mean of Hamming distances computed over all possible pairs of subvectors in S_w:

$$H(w) = \frac{1}{n} \sum D(x) \quad \forall x \in \binom{S_w}{2};\ n = \left| \binom{S_w}{2} \right| \quad (3)$$

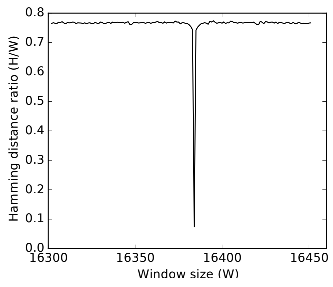
where $D(x)$ is the Hamming distance between two vectors. Based on this, the size of the PHT can be defined as follows:

$$Size_{PHT} = Min\left(\frac{H(w)}{w}\right) \quad \forall w \in \left\{2, \ldots, \frac{|V|}{2}\right\} \quad (4)$$

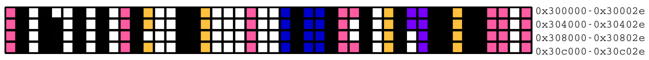
If the resulting function has several local minima, the value with lowest value of w is selected. To find the size of PHT we obtained measurements form 2^{16} contiguous addresses. Then we tested all possible window sizes from 2 to 2^{16} and computed the ratio $\frac{H(w)}{w}$. To speed up the process, instead of trying all possible permutations, we computed Hamming distances of 100 random permutations for each window size. The results showing the minimal value of the ratio are presented in Figure 5b. The minimal value is attained for window size 2^{14}. Thus, we make a conclusion that the size of PHT is 16 384 entries. Figure 5c demonstrates the collected data in the aligned form such that items in each row map to the same PHT entries. The repeated pattern can be clearly observed.



(a)



(b)



(c)

Figure 5. Demonstration PHT probing for a range of addresses and its alignment

7 Implementation of BranchScope

Based on the steps described above, in this section, we construct and evaluate the actual attack. BranchScope consists of a spy process that executes the prime (*stage 1*) and triggers a victim process being attacked to execute (*stage 2*). The spy then executes the probe (*stage 3*) to complete the attack. We assume that the spy can slow down the victim process in order to allow it to execute a single branch instruction during the context switch. In such a scheduling scenario, the spy can prime, then allow the victim to execute a single branch, and then probe. In the standard case, this requirement can be met using [26]. In the case of an SGX enclave, and many other isolated execution solutions [12, 16, 17, 53] this requirement is trivially met because the SGX threat model assumes the attack controls the OS, and hence scheduling.

To demonstrate this attack, we first carry out a covert channel experiment. First, we generate a large array of random bits. This array is loaded to the address space of the victim process (the spy does not have access to this array). The victim repeatedly executes a branch instruction, whose outcome depends on values stored in the array (as shown in Listing 2). The relevant portions of the disassembled victim code is presented in Listing 2(B). The branch is taken when the value of the `if` condition is zero. The spy's pseudo-code is shown in Listing 3. The task of the spy is to determine

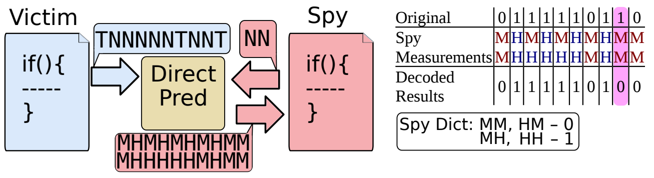


Figure 6. Demonstration of BranchScope. The attacker process primes and probes direction predictor, then uses the dictionary to receive direction of the victim branch

```
int sec_data[]
= {1,0,1,1,..};
i = 0;
void victim_f(){
 //Victim Branch
 if(sec_data[i])
  asm("nop;nop");
 i++;
}
```

(A)

```
mov 0x601080(,%rax,4),%eax
test %eax,%eax
je 300006d <victim_f+0x6d>
nop
nop
```

(B)

Listing 2. Pseudo-code of the Victim Program (A) and Disassembly of the if-statement (B)

```
int probe_array [2] = {1, 1};//Not-taken
int main(){
  for(int i = 0; i < N_BITS; i++){
    randomize_pht();//(1)
    usleep(SLEEP_TIME);//Wait for victim
    spy_function(probe_arr); } }
void spy_function(int array [2]){
  for(int i = 0; i < 2; i++){
    a = read_branch_mispred_counter();
    if(array[i])// <- Spy branch
      asm("nop; nop; nop;");
    b = read_branch_mispred_counter();
    store_branch_mispred_data(b - a); } }
```

Listing 3. Pseudo-code of the attacker program

the contents of the secret array, based on the observed behavior of the branch predictor. The core of the spy program is `spy_function()` which executes a single branch instruction (in the `if` statement) and records the prediction data associated with that branch for future analysis. To achieve a collision of the spy's branch with the victim's branch inside the PHT structure, we placed the two branch instructions at identical virtual addresses in both processes. This ensures that when the BPU uses the 1-level predictor, the two branches will be mapped to the same PHT entry. To obtain more directions of the victim's branches, the three steps are repeated starting from executing the randomized code block that places the PHT entry into a required initial state and turns off 2-level prediction mechanisms.

The attacker process relies on hardware performance counters [51] for precise detection of correct and incorrect prediction events. If access to the performance counters is not available, timing measurements using the time stamp counter can also be used as we discuss in the next section. The spy extracts a sequence of branch misprediction values and decodes this sequence to determine the victim's branch direction. For example, if the attacker observes a sequence of two mispredicted branches or one correctly predicted and one mispredicted branch, then the victim branch is detected as taken, otherwise it is not-taken. An example of data leakage across the covert channel is presented in Figure 6. The figure also demonstrates an erroneously received bit. Note that the dictionary of patterns that we use in this experiment is extended with rarely observed misprediction patterns in order to include all four possible combinations.

To measure this error rate on the covert channel, we use it to transfer 1 million bits, once with all bits set to 0, another with all set to 1, and the third with randomly chosen bit values. For each bit, we execute the branch condition dependent on the bits value, either taken or not taken. The attacker is scheduled on the same core as the victim process. The bits collected by the attacker are compared with the original bits and the error rate calculated. We performed this experiment on three recent x86_64 processors from Intel — Skylake, Haswell and Sandy Bridge — under two settings. In the first setting, the benchmark was scheduled on an isolated physical core, with no other user processes running. In the second setting, no restrictions were set. Since each physical core on our experimental machines has two hardware thread contexts, other normal system activity was simultaneously executed on the core in this noisy setting.

We performed the above experiment 10 times and computed the average rates. The results are presented in Table 2. *BranchScope* features excellent accuracy on both processors with slightly better results on Skylake and Haswell. The Skylake and Haswell processors showed very low error rate even with the presence of external noise. This can be explained by a larger size of the predictor tables in the improved branch predictor design [46] when compared to the older Sandy Bridge processor.

8 Detecting Branch Predictor Events with Timestamp Counter

A key functionality required for the *BranchScope* attack is the ability to detect branch predictor events. In Sec 7 we made use of hardware performance counters to detect the missed branches. This approach, relies on the hardware explicitly providing the branch prediction result. In order to make use of this, however, an attacker would need at least partially elevated privileges.

	All 0	**All 1**	**Random**
SL isolated	*0.46%*	*0.51%*	*0.63%*
SL with noise	*0.64%*	*0.63%*	*0.74%*
Haswell isolated	*0.16%*	*0.27%*	*0.46%*
Haswell with noise	*0.37%*	*0.29%*	*0.67%*
SB isolated	*0.68%*	*1.76%*	*2.44%*
SB with noise	*1.76%*	*4.88%*	*3.38%*

Table 2. Average error rate for transmitting bits using *BranchScope* on Intel Skylake (SL), Haswell and Sandy Bridge (SB) processors

An alternative approach is to detect branch related events by observing their effect on the CPU performance. An incorrectly-predicted branch results in fetching of wrong-path instructions and significant cycles lost for restarting the pipeline. Therefore, the attacker can track the number of cycles to determine if the branch was predicted correctly. This timekeeping can be realized with `rdtsc` or `rdtscp` instructions on Intel processors. These instructions provide user processes with direct access to timekeeping hardware, bypassing system software layers.

The *BranchScope* attack requires the attacker to detect whether a single instance of a branch execution was correctly or incorrectly predicted, rather than relying on the aggregate BPU performance. To evaluate the applicability of the `rdtscp` instruction as a dependable measurement mechanism for the purposes of our attack, we performed a series of experiments. First, we collected time measurements of a single branch instruction when it is correctly and incorrectly predicted for two cases: taken branch and non-taken branch. For each case, 100 000 samples were collected. The resulting data, along with computed mean values, is presented in Figure 7. The case when the actual branch outcome was not-taken is depicted in Figure 7a, while the case with taken outcome is shown in 7b. As seen from the figures, a branch misprediction has a noticeable performance impact, and the effect is present regardless of the actual direction of the branch. The slowdown is clear in the individual data points, as well as in the mean values. To eliminate the impact of caching on these measurements, we executed each branch instance two times, but only recorded the latency during the second execution, after the instruction has been placed in the cache.

Specifically, we recorded the latencies of a single branch instruction executed two consecutive times when the BP correctly predicts the outcome (prediction hit). We refer to this measurements as H_1 and H_2. Then we performed the same measurement for the case when the direction was mispredicted. We refer to these measurements as M_1 and M_2. Since the latency of a mispredicted branch must be higher that the correctly predicted one, we can compute the branch event detection error rate as the percentage of cases when $H_1 > M_1$ or $H_2 > M_2$. We compute this error rate individually for the

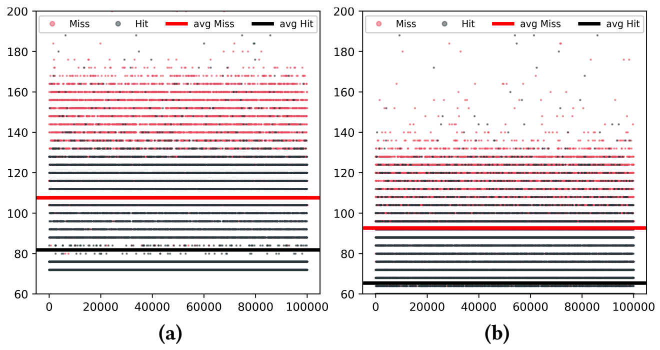


Figure 7. Latency (cycles) of a not-taken (a) and taken (b) branch instruction

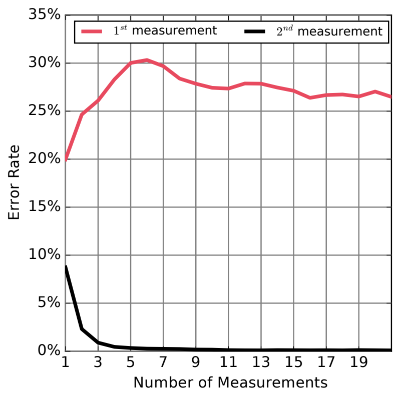


Figure 8. Branch predictor event detection error rate as a function of the number of RDTSCP measurements

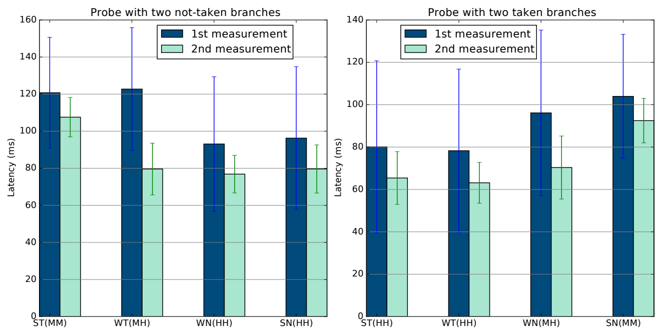


Figure 9. Probe latency (both first and second measurement) depending on the state of PHT

first the second measurements. In addition, to amortize the noise, instead of relying on a single time measurement, we collected multiple measurements and computed the mean value. The results are presented in Figure 8. As expected, the error rate is higher in the first measurement (due to caching effects), within the range of 20-30%. The second measurement has a low error rate of about 10% when a single measurement is used and further reduces to almost 0 as the number of measurements approaches 10.

These results demonstrate the feasibility of time measurements using the `rdtscp` instruction as the branch event detection mechanism. Even though correctly detecting events in the first execution of a branch is challenging, it does not affect the *BranchScope* attack, because the attacker can place a PHT entry into a state that revels the outcome of the victim's branch based only on the observations of the second branch execution. To illustrate this, consider the case when the state of the PHT entry associated with the victim branch is strongly taken (ST) and the attacker uses non-taken branch for probing. If the outcome of the victim's branch is taken, then the attacker will observe the **MM** pattern. When the victim's branch outcome is not-taken, the spy will observe **MH** pattern. Therefore, to reveal the direction of the victim's branch, only the observations from the second branch execution is relevant.

Figure 9 demonstrates how different states of PHT entry affect the timing of probing branches. The graph features measurements for all four states and for both types of the probing and also depicts the standard deviation for each result. It is easy to see from the graph that the PHT states can be reliably distinguished using time measurements.

9 Attack Applications of BranchScope

BranchScope can be directly leveraged to target a system that supports isolated execution, such as Intel's SGX [42], or be used as a general side channel attack in conventional environments. In this section, we first overview Intel SGX and attack considerations in such an environment, then describe a series of specific attacks that can be conducted on a victim even when it is running inside of an SGX enclave.

9.1 Overview of Intel SGX

Intel's Software Guard Extensions (SGX) is a hardware-based isolated execution system designed to protect application secrets from compromised system software, such as operating system kernels and hypervisors.SGX extensions to the x86-64 ISA offer applications a set of instructions which can be used to launch a secure *enclave* that is embedded within the address space of the application. Accesses to enclave memory are controlled by the SGX hardware to prevent access from the outside of the enclave. Therefore, if an application stores sensitive code and data inside an enclave, the secrets are inaccessible to even system software.

In addition to providing runtime access control to enclave memory pages, SGX supports mechanisms for memory encryption and integrity checking to provide protection against physical attacks on memory. SGX is a major effort from the industry to provide hardware support for security and

	All 0	**All 1**	**Random**
SGX with noise	*0.008%*	*0.53%*	*0.73%*
SGX isolated	*0.003%*	*0.153%*	*0.51%*

Table 3. Covert channel benchmark: average error rate for transmitting bits using BranchScope on Intel Skylake when a trojan (victim) executes in an SGX enclave and the spy is a regular process assisted by the OS

trusted computing, and is currently the subject of a significant amount of research.

Isolated execution environments such as SGX can be vulnerable to side channel attacks. While memory is protected, many CPU hardware resources still remain shared between enclave and non-enclave code. The side-channel threat in an isolated execution context may be even more serious for two reasons. First, users tend to place more trust in systems claiming advanced security features [5]. Second, the threat model assumes that the attacker has full control over system software. This means that the attacker has full control over scheduling an enclave, the ability to control noise from prefetchers, caches, as well as other workloads. The OS can also control other parameters such as the CPU core frequency, page translation, low-level performance counters and many other functions which otherwise add noise. Several recent works have studied this problem in detail. For example, Moghimi et al. [44] investigated how SGX can "amplify" known cache attacks, making isolated entities extremely vulnerable to such attacks. Schwarz et al. [47] demonstrated how SGX can be used to conceal cache attacks, making anti-malware software, even one running at the kernel level, incapable of detecting cache side-channel attacks. Finally, SGX enclaves were shown to be vulnerable to traditional cache side-channel attacks [22] as well as some new attacks unique to SGX, in particular page table side-channel attacks [54].

9.2 *BranchScope* attack scenarios

In an SGX environment, the control over the OS gives the attacker unique capabilities to perform the *BranchScope* attack in a low-noise environment. The success of the attack largely depends on the ability to perform branch manipulations with precise timing. The attacker controlled OS can easily manipulate victim execution timings. For example the attacker can configure the Advanced Programmable Interrupt Controller (APIC) in such a way that enclave code is interrupted after several instructions are executed [35]. Alternatively, the attacker can unmap certain memory pages to force a interrupt when an enclave executes certain code [54].

Covert channel attack on SGX: To illustrate *BranchScope* in an SGX environment, we repeat our covert channel benchmark with the sender running inside the SGX enclave using *BranchScope* to communicate to a receiver outside SGX. Table 3 illustrates *BranchScope*'s covert channel quality: the error rates are acceptable even in the presence of noise; however, when the OS controls the noise (by preventing other processes from running), the quality of the channel is improved.

Next we overview other examples of applications that can be attacked using BranchScope. The attacks would work whether these applications are running as usual or inside of an SGX enclave.

Montgomery ladder: The Montgomery ladder is a popular algorithm used in modular exponentiation [32] and scalar multiplication [45] algorithms. Both these mathematical operations constitute the key components of traditional RSA as well as elliptic curve (ECC) implementations of public-key cryptography. Montgomery ladder is based on performing operations regardless of bit value k_i in secret key k. This implementation mitigates timing and power side channels by equalizing the execution paths. However it requires a branch operating with direct dependency from the value of k_i. Yarom et. al. [55] demonstrated the vulnerability of the OpenSSL implementation of ECDSA cipher using the FLUSH+RELOAD cache side channel attack. In this attack the CPU cache was used to spy on the direction of the target branch. *BranchScope* can directly recover the direction of such branch. Although most recent versions of cryptographic libraries do not contain branches with outcomes dependent directly on the bits of a secret key, often some limited information can still be recovered [6, 8] and many outdated libraries are still in use.

libjpeg: Another example of how our attack can reveal sensitive information is an attack against libjpeg, a popular JPEG encoding/decoding library. The attack is possible because of the inverse cosine transform (IDCT) operation performed during decompression. In this optimization elements in rows and columns of coefficient matrices are compared to 0 to avoid costly computations. Each such comparison is realized as an individual branch instruction. By spying on these branches the *BranchScope* is capable of recovering information about relative complexity of decoded pixel blocks. Attacks on libjpeg were previously demonstrated using the page fault side channel [27, 54] by counting the number of times the optimization can be applied, resulting in recovery of an original image. The *BranchScope* attack is advantageous as it not only allows to distinguish the cases when all row/column elements are zero, but also indicates which element is not equal to zero.

ASLR value recovery: *BranchScope* can also be used to infer control code within victim enclaves. The attacker may learn not only whether a certain branch was taken or not, but also detect the location of branch instruction in a victim's virtual memory by observing branch collisions. This allows the attacker to bypass the address space layout randomization

(ASLR) protection. Previously, similar attacks were demonstrated using the BTB [21, 35]. As indicated by Gruss [23] the BTB-based attack does not work on recent Intel's processors. This makes the direction predictor a unique candidate for this class of attacks.

10 Mitigating BranchScope

The root cause of branch-based attacks is the execution of branch instructions that are conditioned on the state of secret data. Our goal in this paper is to highlight this new source of leakage in a branch predictor unit as a source of vulnerability. In this section, we overview several possible defenses against *BranchScope* both in software and hardware. Exploring these defenses is an interesting direction for future research.

10.1 Software-only Mitigations

Software-only solutions can be highly sensitive to the underlying organization of the branch predictor unit. In addition to the side channel threat, malicious entities can communicate between each other using *BranchScope*, bypassing existing restrictions. For example, a sealed SGX enclave can transfer sensitive information to regular process violating security properties of the SGX system. Software mitigation techniques cannot provide protections from covert channels as they do not remove the source of leakage in hardware, leaving attackers free to use it to communicate covertly.

One possible mitigation technique is to algorithmically remove dependencies of branch outcomes on secret data [3]. However, it is challenging to apply such protection to large code bases, thus this mechanism can only be limited to the key parts of programs operating with sensitive data.

Another possible approach that has a broader applicability is to eliminate conditional branches from target programs. This technique, known as *if-conversion* [10], is a compiler optimization that converts conditional branches to sequential code using conditional instructions such as `cmov`, effectively turning control dependencies into data dependencies. If-conversion removes conditional branch instructions, thus mitigating the *BranchScope* attack. Several studies [9, 11] used if-conversion as a mitigation for timing side-channel attacks. It is easy to apply this method to simple branches with few dependencies. However, conversion of complex control flow (different code is executed depending on branch outcomes) is challenging. It is unknown if it possible to convert real-world applications to branch-free code. Moreover, highly-predictable branches typically perform worse when if-converted [10].

10.2 Hardware-supported Defenses

The design of the branch predictor mechanism can be rearchitected to mitigate leakage through the directional branch predictor unit. In this section, we overview several possible such mitigations. Exploring effective mitigations is an interesting direction for future research.

Randomization of the PHT: *BranchScope* requires the ability to create predictable collisions in the PHT (e.g., based on virtual address). To prevent such collisions, the PHT indexing function can be modified to receive as input some data unique to this software entity. For example, this can be part of the SGX hardware state, or simply some random number generated by the process. One time randomization may be vulnerable to a probing attack that examines PHT entries one by one until it finds the collision; periodic randomization can be used (sacrificing some performance). This solution is similar to randomizing the mapping of caches as a protection against side-channel attacks [52].

Removing prediction for sensitive branches Since not all branch instructions can leak sensitive information, a mitigation approach can be taking favoring this observation. A software developer can indicate the branches capable of leaking secret information and request them to be protected. Then the CPU must avoid predicting these branches, rely always on static prediction and avoid updating any BPU structures after such branches are executed. Although this mitigation technique has a negative performance overhead it offers perfect security for most security sensitive branches. As with the software techniques, this method does not protect again ts the covert channel attack.

Partitioning the BPU The BPU may be partitioned such that attackers and victims do not share the same structures. For example, SGX code may use a different branch predictor than normal code. Alternatively, mechanisms to request a private partition of the BPU may be supported [37]. With partitioning, the attacker loses the ability to create collisions with the victim.

Other solutions. Other solutions are also possible. For example, we may remove the attacker's ability to measure the outcome of a branch accurately, by removing or adding noise to the performance counters or the timing measurements [39]. Another solution may change the prediction FSM to make it more stochastic, interfering with the attacker's ability to precisely infer the direction of the branch taken by the victim. Finally, a class of solutions may focus on detecting the attack footprint and invoking mitigations such as freezing or killing the attacker process if an ongoing attack is detected. In an SGX context where the attacker has compromised the OS this may be difficult; alternatively, the SGX code may decide to remap itself or stop execution if it detects an ongoing attack.

11 Related Work

The first research studying branch predictor based side-channels was conducted by Aciicmez et al. [1–3]: they presented four different attacks, demonstrating them against implementations of the RSA encryption standard. The first attack exploits the deterministic behavior of the branch predictor by simulating the exponentiation steps and measuring the time differences that depend on the prior state of the predictor. The second attack assumes that the spy process runs on a parallel virtual core alongside with the victim. The spy constantly removes the victim's entries from the BTB in order to force the branch predictor to predict all branches as not-taken (assuming that BTB misses result in not-taken predictions). The third attack is also based on the spy filling the BTB with its own data. The main difference here is that this attack is synchronous, meaning that the attacker can perform the BTB filling right before the target branch is executed. Finally, in the last attack, the spy also executes in parallel and fills the BTB, but this time instead of measuring the total execution time of the cryptographic algorithm, the spy detects evictions of its BTB entries when the victim process executes taken branches. They later significantly improved the last attack's accuracy by carefully adjusting the intensiveness of the BTB filling. This attack is most interesting due to the demonstrated practical results with high accuracy.

All attacks described above are substantially different from *BranchScope*. All but the first attack (which is a timing-analysis attack) rely on filling the BTB (which is a cache-like structure) and thus are similar to the cache side-channel attacks [38, 56]. This makes it possible to apply existing cache protection techniques [14, 52] to protect the BTB. In contrast, *BranchScope* exploits the hybrid property of modern branch predictors and manipulates the data directly in the directional predictor, thus opening up a previously unexplored side-channel. Branch predictors have been used for constructing covert channels in [19, 20, 29]. However, these works rely on the did not investigate the possibility of fine-grained branch direction recovery. Bhattacharya et al.[7] considered a fault attack on RSA combined with the analysis of the number of branch mispredictions.

Recent works exploited microarchitectural features to construct covert channels [18, 40]. These channels allow attackers to bypass system isolation, including Intel SGX [25]. As we demonstrated, *BranchScope* can be used in a similar fashion to transfer information across isolation boundaries.

12 Concluding Remarks

In this paper we presented *BranchScope* — a new micro-architectural side-channel attack that exploits directional branch predictor to leak secret data. We demonstrated the attack on recent Intel processors. Our results showed that secret bits can be recovered by the attacker with very low error rate and without the knowledge of the internal predictor organization. Therefore, researchers and system developers have to consider *BranchScope* as a new security threat while designing future systems. We proposed several countermeasures to protect future systems from *BranchScope*.

13 Acknowledgments

This paper was made possible by NPRP grant 8-1474-2-626 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] O. Aciicmez, K. Koc, and J. Seifert. On the power of simple branch prediction analysis. In *Symposium on Information, Computer and Communication Security (ASIACCS)*. IEEE, 2007.

[2] O. Aciicmez, K. Koc, and J. Seifert. Predicting secret keys via branch prediction. In *The cryptographers' track at the RSA conference*, 2007.

[3] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *Cryptography and Coding*, pages 185–203. Springer, 2007.

[4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 422–435. ACM, 2016.

[5] Iosif Androulidakis and Gorazd Kandus. Feeling secure vs. being secure the mobile phone user case. In *Global security, safety and sustainability & e-Democracy*, pages 212–219. Springer, 2012.

[6] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.

[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. Fault Attack revealing Secret Keys of Exponentiation Algorithms from Branch Prediction Misses. Cryptology ePrint Archive, Report 2014/790, 2014.

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

[9] Jonathan Burket and Samantha Gottlieb. If-Conversion to Combat Control Flow-based Timing Attacks. 2014.

[10] 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.

[11] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 45–60. IEEE, 2009.

[12] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[13] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *Smart Card Research and Applications*, pages 167–182. Springer, 2000.

[14] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side-Channel Attacks. In *ACM Transactions on Architecture and Code*

Optimization, Special Issue on High Performance and Embedded Architectures and Compilers, January 2012.

[15] Marius Evers, Po-Yung Chang, and Yale N Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 3–11. ACM, 1996.

[16] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE Computer Society, 2014.

[17] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Flexible hardware-managed isolated execution: Architecture, software support and applications. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[18] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 843–857. ACM, 2016.

[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, page 5. ACM, 2015.

[20] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and Mitigating Covert Channels Through Branch Predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[22] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, , and Tilo Müller. Cache Attacks on Intel SGX. 2017.

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[24] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[25] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. 2017.

[26] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505, 2011.

[27] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, Santa Clara, CA, 2017. USENIX Association.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[29] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 639–650. IEEE, 2015.

[30] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[31] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 197–206. IEEE, 2001.

[32] Marc Joye and Sung-Ming Yen. The Montgomery powering ladder. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 291–302. 2002.

[33] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, January 2018.

[35] 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 *Usenix Security Symposium*, 2017.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[37] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *36th IEEE Symposium on Security and Privacy (S&P 2015)*, 2015.

[39] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[40] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[41] Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[42] 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. *HASP@ ISCA*, 10, 2013.

[43] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 292–303. ACM, 1997.

[44] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *arXiv preprint arXiv:1703.06986*, 2017.

[45] Thomaz Oliveira, Julio López, and Francisco Rodríguez-Henríquez. The Montgomery ladder on binary elliptic curves. *Journal of Cryptographic Engineering*, pages 1–18, 2017.

[46] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: don't trust folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 103–114. IEEE Computer Society, 2015.

[47] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[48] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS)*, pages 298–307, 2004.

[49] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[50] Eric Sprangle, Robert S Chappell, Mitch Alsup, and Yale N Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 284–291. ACM, 1997.

[51] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pages 59–67. IEEE, 2008.

[52] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[53] Johannes Winter. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

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

[55] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[56] Yuval Yarom and Katrina E Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[57] 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. ACM, 1991.