Branchboozle: A Side-Channel Within a Hidden Pattern History Table of Modern Branch Prediction Units

Andrés R. Hernández C.
The University of Texas at San Antonio

Wonjun Lee
Yeshiva University New York

Wei-Ming Lin
The University of Texas at San Antonio

ABSTRACT

We present *Branchboozle*, a side-channel that can be configured on top of a hidden Pattern History Table (PHT) now found in modern Branch Prediction Units (BPUs). In a similar fashion to known BranchScope attacks, *Branchboozle* works by closely monitoring the directional predictions issued by the BPU, i.e., whether a branch is predicted *Taken* or *Non-Taken*. However, our attack exclusively focuses on analyzing the predictions issued by a secondary, mostly-undocumented 3-bit PHT, whereas BranchScope attacks manipulate the predictions issued by a textbook-like 2-bit PHT.

This work describes how *Branchboozle* can configure an extremely robust covert-channel among independent processes that works even across the physical threads of an execution core with Simultaneous Multi-Threading technology. Additionally, we demonstrate that branches protected by Intel's Software Guard eXtensions are also vulnerable to our attack setting. Finally, we illustrate how *Branchboozle* can potentiate transient execution attacks dependent on branch direction misprediction, i.e., Spectre Variant 1.

CCS CONCEPTS

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

KEYWORDS

Branch prediction, hardware side-channel, speculation

ACM Reference Format:
Andrés R. Hernández C., Wonjun Lee, and Wei-Ming Lin. 2021. Branchboozle: A Side-Channel Within a Hidden Pattern History Table of Modern Branch Prediction Units. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3442035

1 INTRODUCTION

Modern CPUs can enhance system performance by increasing instruction throughput, achievable via optimizations such as pipelined and out-of-order execution. Yet, the effectiveness of these design choices heavily relies on the inclusion of a competent Branch Prediction Unit (BPU), a complex logical component designed to correctly predict the direction of *conditional branches*, i.e., instructions capable of altering the control flow within computer programs.

Because the predictions issued by the BPU are crucial to determine "*what instructions must be executed next*", effective branch prediction has been the subject of a large amount of research that has slowly shaped the internal design of modern BPUs, which now include numerous sub-components that can collectively issue very advanced predictions. Nonetheless, the BPU is a hardware element often shared among processes and unfortunately its sub-components represent a potential attack surface that devoted adversaries can use to infer the control flow of a victim program [1, 2], or even alter it momentarily [16].

In this work we target a hidden Pattern History Table (PHT) sub-component now found in modern BPUs, and describe the specific conditions that lead the BPU to utilize this particular PHT to issue directional predictions, and allows us to reverse-engineer and report the details about the entries found in this hidden PHT. More specifically, we find that the value of each entry is maintained by a Finite-State Machine (FSM) with at least seven states in Intel processors, and five in AMD respectively, meaning that this hidden PHT is an array of 3-bit counters. Analogous to our work, BranchScope attacks by Evtyushkin et al. [9] monitor directional branch predictions issued by a classic 2-bit PHT structure [18].

We describe how an adversary can manipulate individual entries in the hidden 3-bit PHT to mount an extremely robust covert-channel between unrelated processes, and even across stringent security boundaries such as those provided by Intel's Software Guard eXtensions (SGX). Finally, we describe how devoted adversaries can also leverage the 3-bit PHT to configure speculation attacks based on directional branch misprediction, i.e., Spectre Variant 1 [4, 16]. As a summary, this paper makes the following contributions:[1]

- We present *Branchboozle*, a direct attack on the BPU's hidden 3-bit PHT sub-component, which grants an adversary the capability to establish illegal inter-process communication by manipulating the PHT's individual counter entries.
- We illustrate how *Branchboozle* is analogous to BranchScope [9], a known attack on the BPU that targets the classic 2-bit PHT. We complement the results of [9] by implementing BranchScope in a wider variety of micro-architectures, and we demonstrate how these two attacks target different BPU sub-components.
- We prove that *Branchboozle* is applicable to BPUs from both Intel and AMD micro-architectures.
- With *Branchboozle* we potentiate Spectre-like attacks [16] by manipulating the 3-bit PHT to artificially trigger directional branch mispredictions. Under the taxonomy of Canella et al. [4], we deem a necessary finer categorization of Spectre-PHT attacks that takes into actual PHT issuing malicious directional branch

[1] https://github.com/andresr23/Branchboozle

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442035

mispredictions, e.g., new categories such as Spectre-PHT-2b and Spectre-PHT-3b respectively.

This paper is organized as follows. Section 2 provides a general background on branch prediction. Section 3 follows up with a briefer on BranchScope attacks. Section 4 describes the full nature of *Branchboozle*, as well as our reverse-engineering effort in regards to the 3-bit PHT's functionality. Section 5 illustrates *Branchboozle*'s applicability as an inter-process covert channel. Section 6 describes *Branchboozle*'s relevancy in the context of speculation attacks. Finally Section 7 provides a discussion and final comments.

2 BACKGROUND ON BRANCH PREDICTION

Branch instructions allow computer programs to diverge from a sequential execution path, and instead continue executing instructions somewhere else in the program. *Unconditional* branches always diverge to a new execution path, whereas the *conditional* ones only do so if certain conditions are met. In the C language for example, conditional branch instructions are the final implementation of statements like `if-else`, or loops such as `for` and `while`.

Conditional branch instructions are said to be *Taken* (T) if their conditions are met, meaning that the core's Program Counter (PC) must be set to a new address. On the contrary, when a conditional branch is *Non-Taken* (N) the PC is only incremented by one to fetch and execute the next instruction following the branch. For instance, consider the x86 Jump-if-not-zero (`jnz`) instruction which, as the name implies, only performs a jump if the core's Zero Flag is not set (ZF = 0).

When executing conditional branches, the execution core must rapidly determine whether a branch's conditions are met or not to know what instructions must be executed next. However, the condition of a branch may be the result of a lengthy instruction sequence that can take numerous clock cycles to complete, further causing the execution core to stall. However, modern core designs reduce this performance penalty by incorporating a BPU that fulfills two main functions. First, it attempts to predict the *direction* of a conditional branch (Is it *Taken* or *Non-Taken*?). Second, the BPU also predicts the *target* address (to modify the PC) when a branch is predicted as *Taken*. Throughout the remainder of this paper we state that a *hit* (H) occurs when the directional prediction issued by the BPU is correct, and a *miss* (M) occurs otherwise. Note that directional mispredictions (M) always incur a small time penalty.

2.1 Bimodal Predictor

One of the simplest and most effective designs for directional branch prediction employs an array of 2-bit saturating counters ($\{c_1, c_0\}$) commonly known as the Pattern History Table (PHT) [17, 18, 23]. The BPU issues a bimodal prediction by using the n Least-Significant Bits (LSBs) of a branch's virtual address to index a 2-bit counter whose value is maintained by a 4-state FSM depending on the actual branch direction. Here, the directional prediction is issued by reading a counter's Most-Significant Bit (MSB), *Taken* if $c_1 = 1$ else *Non-Taken*, while the real branch direction increments or decrements the counter. The 4-state FSM is said to be in a Strong-*Taken* state (ST) when the counter's value is 3, and a Strong-*Non-Taken* state (SN) when the value is 0. Figure 1.a illustrates the configuration of a Bimodal predictor.

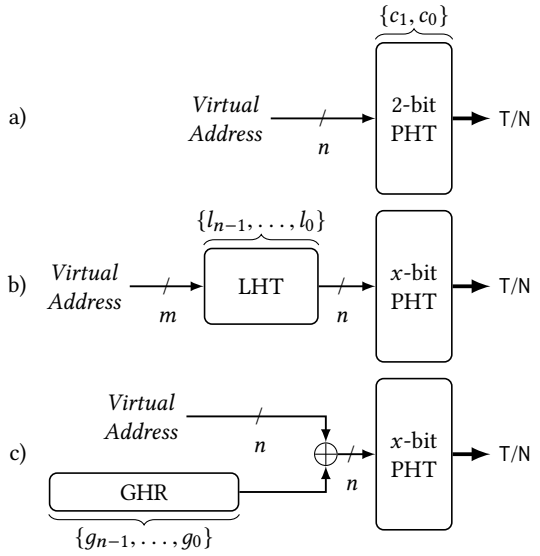


Figure 1: Basic directional predictors [18]. a) Bimodal predictor, b) Local History predictor and c) Global History predictor. Note that a modern BPU may employ a single PHT to implement all three predictors, and the lengths of each LHT entry and the GHR may be other than n.

2.2 Local History Predictor

There may be situations where the behavior of a conditional branch follows a complex, and yet repetitive pattern that a 2-bit counter can't capture. Still, it is possible to take advantage of this predictable behavior by tracking the branch's direction pattern in a Local History Table (LHT), where the contents of each LHT entry are then used to index a PHT counter to construct a so-called Two-Level predictor [18, 23], as shown in Figure 1.b.

2.3 Global History Predictor

Sometimes the direction of a branch depends on the direction of other preceding branches. In such cases a Local History predictor cannot effectively predict the direction of a conditional branch. Nonetheless, it is possible to benefit from this foreseeable behavior by tracking the resolution of all branch instructions in a Global History Register (GHR), to then index a PHT counter by xor'ing the GHR content with the n LSBs of a branch's virtual address. Figure 1.c depicts the design of a Global History predictor, commonly known as a Gshare predictor [18].

2.4 Selection Logic

Having described three possible strategies to predict the direction of a conditional branch, the question now is: *What predictor should the BPU use next?* To make an appropriate decision the BPU includes a sub-component known as the Selection Logic (SL), which can dynamically determine what prediction strategy is performing the best for a given conditional branch.

An Intel patent by Baweja and Kumar [3] proposes a BPU configuration that includes a SL sub-component, as depicted in Figure 2.

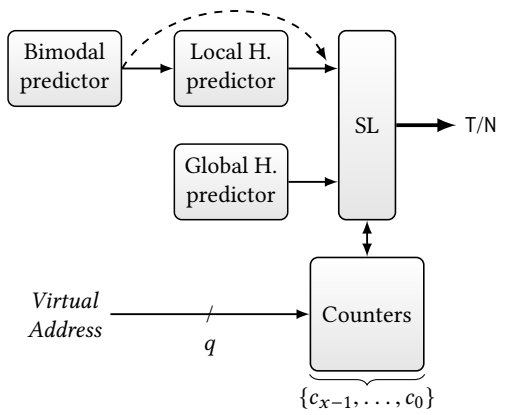


Figure 2: Architecture of a BPU's directional predictor as described in [3]. In cases that the Local History predictor does not have enough information to issue a suitable local prediction, a Bimodal prediction is forwarded instead.

Table 1: Test CPUs

Model	Vendor	Micro-architecture
i5-2400	Intel	SandyBridge
i5-3470	Intel	IvyBridge
i7-4790	Intel	Haswell
i7-6700	Intel	Skylake
i5-7500	Intel	Kaby Lake
i3-9100	Intel	Coffee Lake
Ryzen 5 1600x	AMD	Zen

Here, the SL determines what predictor to use next by tracking whether previous predictions for a specific branch were correct or not. The SL does so by updating individual entries in a "Counters" sub-component, which comprises an array of x-bit counters that are indexed by the q LSBs of the branch's virtual address.

2.5 Detecting Branch Mispredictions

With enough knowledge about the BPU's design, it is possible to derive precise details about its functionality by monitoring branch mispredictions (`M`). For the ease of experimentation we do this by configuring Performance Monitoring Counters (PMCs) that can be read from the user-space by executing the `rdpmc` instruction. In Intel CPUs we monitor `BR_MISP_RETIRED` events (PMC `0xC5`) with unit mask `CONDITIONAL` (`0x01`) [5]. In AMD we examine the `EX_RET_BRN_MISP` events (PMC `0x0C3`) with mask `ALL_BRANCHES` (`0x00`) [14]. Note that PMCs and the `rdpmc` instruction must be properly configured with kernel-space code and proper privileges.

2.6 Test Environment

All our experimentation is carried out in multiple x86-64 CPUs ranging across different micro-architectures, as shown in Table 1. Our systems run Ubuntu 18.04.5 LTS and all test code has been written in C and compiled with `gcc` 7.5.0.

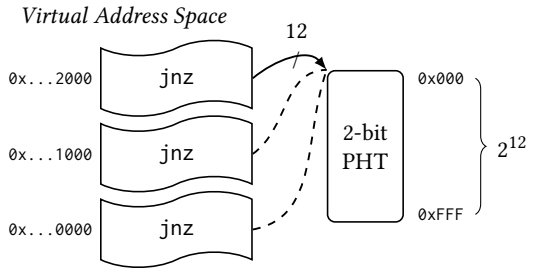


Figure 3: 2-bit PHT aliasing example with $n = 12$.

Table 2: The n LSBs of a branch's virtual address used to index the 2-bit PHT part of the Bimodal predictor.

Micro-architecture	n
SandyBridge	13
IvyBridge	13
Haswell	14
Skylake	15
Kaby Lake	15
Coffee Lake	15
Zen	N/A

3 BRANCHSCOPE

Evtyushkin et al. propose BranchScope [9], an attack on the BPU that leverages the Bimodal predictor's 2-bit PHT as a side-channel to monitor the branch activity of a victim program. Since the BPU is a hardware component that can be shared among different processes, the authors configure a covert-channel to demonstrate that BranchScope is capable of breaking inter-process isolation.

BranchScope compels the SL to select the Bimodal predictor to predict the direction of upcoming branch instructions. BranchScope achieves this by executing at least 100,000 conditional branch instructions that are randomly *Taken* or *Non-Taken*, which in turn "flood" the BPU. This operation is called `randomize_pht` and takes on average more than 130k cycles in Skylake. Right after, any upcoming branch instructions have no local history, and a global prediction yields no benefit due to the random global history. As a result, the SL selects the Bimodal predictor as a fallback to issue simple predictions for the next 50-70 conditional branches [9].

With BranchScope an adversary can detect changes in specific PHT entries by executing malicious branch instructions that have been surgically placed in adequate virtual addresses. That is to say, the adversary can manipulate and put a targeted entry in a controlled state (*Prime*), wait for a victim program to execute, and then observe any changes made to the targeted entry by monitoring branch mispredictions (*Probe*).[2]

3.1 2-bit PHT Aliasing

The 2-bit PHT is a hardware structure with a finite amount of counter entries, hence it is possible to reverse-engineer its size by studying the circumstances that lead to two different branch

[2]This process is detailed in Table 1 of [9].

Listing 1: *History Saturation.*

```
1   movl $s, %ecx
2 .Lloop_start:
3   /* Loop body */
4   loop .Lloop_start
```

instructions to index the same entry. In other words, *2bit PHT aliasing* occurs when two or more conditional branches have the same n-LSBs in their respective virtual addresses, and thus index the same PHT entry. In turn, the ocurrence of this phenomenon strongly suggests that the corresponding 2-bit PHT has 2^n counter entries as depicted in Figure 3.

By implementing BranchScope in our test CPUs we discover the different values of n used to index the 2-bit PHT in the Bimodal predictor, as outlined in Table 2, where our Haswell result matches the one reported by Evtyushkin et al. [9]. Finally, while our implementation works in all our Intel processors, we report that BranchScope fails to expose the 2-bit PHT in our AMD Zen processor.[3]

4 BRANCHBOOZLE

We now describe *Branchboozle*, an attack on a hidden 3-bit PHT implemented in modern BPUs. At large, *Branchboozle* manipulates a counter value stored in an entry of this 3-bit PHT to use it as a covert-channel, and thus break inter-process isolation.

4.1 History Saturation

The foundation of *Branchboozle* lies on its ability to consistently index a particular entry in the 3-bit PHT. To accomplish this, we execute a particular sequence of branches to manipulate the BPU's advanced circuitry that is used to index this PHT, in a procedure that we call *History Saturation*.

Modern BPU designs now track the direction of all the recent *Taken* branches in an n-bit Branch History Buffer (BHB) which, in recent Intel micro-architectures, is updated via a rolling-hash function. Kocher et al. [16] detail a 58-bit BHB implemented in some Haswell processors, updated by left-shifting the BHB content by 2 bits to then xor it with a portion of the branch's virtual address (or *source*, bits [19:04]) and its target address (or *destination*, bits [04:00]). In contrast to the classic prediction designs discussed in §2, we note that the BHB content is often used to index the counter entries of this hidden 3-bit PHT to issue directional predictions.[4]

We discover the practicality of prompting the BPU's SL circuitry to invariably select the predictions issued by this 3-bit PHT, done by leveraging the x86 `loop` instruction. Under normal circumstances, the number of iterations required to configure a software loop is configured by first writing a value into the `ecx` register. Then, the `loop` instruction performs the following when executed:

- Decrement the iterator in the `ecx` register (`dec %ecx`).
- Jump back to the loop's start if `ecx` $\neq 0$ (`jnz`).

Since the second sub-operation acts as an x86 jump-if-not-zero (`jnz`) instruction, `loop` benefits from directional branch predictions, and its execution also alters the BPU's internal state. Therefore, it is possible to fix the BHB content by executing the *History Saturation* code provided in Listing 1. By first writing a sufficiently high value (`$s`) into the `ecx` register, the repeated execution of `loop` generates a branch direction history as follows:

`TTTTTTTTTTTTTTT . . . TN.`

This sequence updates the BHB an amount (`$s` -1) of times due to the *Taken* resolutions required by `loop` to jump back to the start. In this case however, the BHB is repeatedly updated with the same *source* and *destination* bits, causing the BHB content to stagnate after a certain number of iterations (`$s`). Furthermore, the BHB content remains invariant even after subsequent *Taken* executions of this `loop` instruction, (i.e., `$s` $+1$, `$s` $+2$,...) We verify this observation by simulating a BHB that uses the rolling-hash function from [16], updated by a fictitious `loop` instruction, placed at a random virtual address, in a similar configuration to the `loop` in Listing 1. Indeed, the BHB content becomes invariant when the number of *Taken* resolutions is greater or equal to half the bit size of the BHB, i.e., when $\$s - 1 \geq n/2$.

After fixing the BHB content to index a particular entry in the 3-bit PHT, the subsequent *Taken* resolutions of the `loop` instruction also transition a 3-bit entry to its Strong-Taken state. Therefore, we can estimate the real BHB bit size (n) by analyzing the minimum amount of iterations (`$s`) that force the last *Non-Taken* (`N`) execution of `loop` to always reflect a branch misprediction (`M`). Figure 4 depicts the occurrence of this branch misprediction by testing different iteration values (`$s`). Here, *History Saturation* successfully stagnates the BHB in all our Intel processors when `$s` > 94, and the same holds in AMD's Zen when `$s` > 36. Therefore, by considering a 2-bit shift as part of the BHB's rolling-hash function, we can estimate that the BHB has a bit size of approximately 186 bits in our Intel CPUs, and about 70 in AMD's Zen. For simplicity we use `$s` $= 100$ in all the experimentation discussed in this paper.

Finally, one can consider *History Saturation* as a procedure analog to BranchScope's `randomize_pht`. However, in Skylake *History Saturation* is orders of magnitude faster with an average latency of only 600 cycles when `$s` $= 100$ (cf. 130k from `randomize_pht`). Additionally, *History Saturation* can also be implemented directly from C via an empty `for` that iterates at least `$s` times, and even a `while(--$s)` loop with an appropriate initial value `$s`.

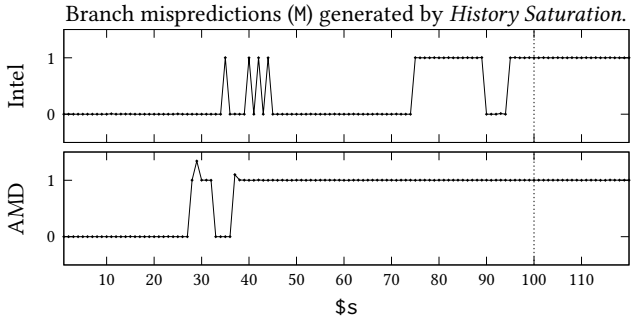


Figure 4: Averaged count of branch mispredictions (y-axis) that occur when executing the *History Saturation* code with different `$s` values (Listing 1).

[3] We do not discard the possibility that an enhanced version of BranchScope may be capable of targeting the BPU in AMD's Zen. Achieving this is beyond our scope.
[4] Hennessy and Patterson [12] briefly discuss the inclusion of a 3-bit PHT to implement state-of-the-art Local History predictors in newer BPU designs.

Table 3: Interactions with the internal FSM of the 3-bit PHT in Skylake. The symbol U stands for *unknown*, meaning that the test does not consistently observe either branch prediction hits (H) or misses (M).

Prime	*Probe*	*Probe* Misprediction Rates (Averaged over 1 million measurements)
TTTTTTT	NNNNNNNN	M M M U U H H H (1.000 - 1.000 - 1.000 - 0.570 - 0.398 - 0.000 - 0.000 - 0.000)
NNNNNNN	TTTTTTTT	M M M U U H H H (1.000 - 1.000 - 1.000 - 0.567 - 0.415 - 0.000 - 0.000 - 0.000)

Listing 2: Manipulate and observe an entry in the 3-bit PHT.

```
1 uint32_t pht3b_branch(uint32_t bd){
2   uint32_t pmc_start, pmc_end;
3   asm volatile(...loop...); // History Sat.
4   pmc_start = __rdpmc;
5   if (bd) {} // Taken if bd!=0, else Non-Taken
6   pmc_end   = __rdpmc;
7   return (pmc_end - pmc_start);
8 }
```

4.2 Interacting with the 3-bit PHT

Having discussed *History Saturation*, Listing 2 outlines the fundamental C function that *Branchboozle* uses to interact with the 3-bit PHT, where line 3 implements the assembly code presented in Listing 1. The function works by executing the conditional branch instruction that implements the `if` statement in line 5, where an adversary can select if this branch should be *Taken* or *Non-Taken* via the branch direction parameter (bd). Additionally, the function checks if the execution of this branch incurs a branch misprediction by reading the corresponding PMC (§2.5). It is important to keep in mind that the branch implementing the `if` statement can be interpreted as an x86 jump-if-zero (`jz`), and it is this branch instruction the one that interacts with a counter entry in the 3-bit PHT.

4.3 The Finite-State Machine of the 3-bit PHT

The `pht3bit_branch` function provides a simple way to manipulate an entry in the 3-bit PHT, hence we now study and determine how an internal FSM maintains the information of each entry.

We notice that a branch's direction determines how the internal FSM transitions across states, and that numerous independent FSM states are stored in 3-bit counters in the hidden PHT. We wish to properly sketch the FSM's structure, therefore we *Prime* the FSM into its Strong-*Taken* state (ST) by repeatedly invoking the `pht3bit_branch` function with an appropriate bd parameter:[5]

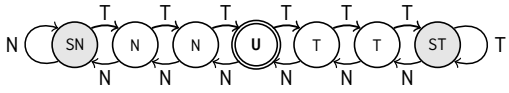
```
1 pht3b_branch(TAKEN);
2 ...
3 pht3b_branch(TAKEN);
```

Then, we *Probe* by monitoring branch mispredictions while transitioning the FSM towards the opposite strong state. In this example we transition the FSM to its Strong-*Non-Taken* state (SN):

```
1 miss_count[  0] += pht3b_branch(NON_TAKEN);
2 ...
3 miss_count[x-1] += pht3b_branch(NON_TAKEN);
```

[5]Following Listing 2, `TAKEN` = 0 and `NON_TAKEN` ≠ 0.

a) Intel's Sandybridge, Ivybridge and Haswell:



b) Intel's Skylake, Kaby Lake and Coffee Lake:

c) AMD's Zen:

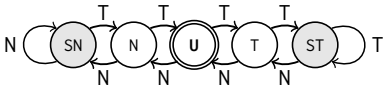


Figure 5: The different FSMs part of the hidden 3-bit PHT in all our test CPUs. Nodes depict directional predictions and edges show state transitions. The U states appear to issue adaptive predictions according to the global branch history.

Table 4: The x LSBs of a branch's virtual address (*source*) are used to update the BHB during *History Saturation*.

Micro-architecture	x
SandyBridge	20
IvyBridge	20
Haswell	20
Skylake	19
Kaby Lake	19
Coffee Lake	19
Zen	26

In this way, we can discern how the FSM part of the 3-bit PHT selects and issues predictions by averaging the measurements gathered from numerous *Prime+Probe* rounds.

Table 3 outlines the results of our *Prime+Probe* experiment in Skylake, where we *Prime* by calling `pht3bit_branch` seven times and *Probe* by doing so another eight times. From these results we discover that the 3-bit PHT incorporates an 8-state FSM, with six states capable of issuing simple predictions, and two middle states that issue adaptive predictions with the help of undocumented circuitry. Moreover, by examining all our CPUs we discover three different FSM designs, as shown in Figure 5, which allows us to firmly conclude that each entry in this hidden PHT in all our test CPUs is comprised of at least 3 bits.

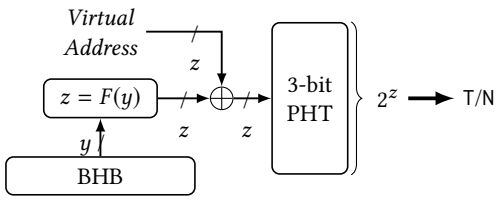


Figure 6: Possible indexing scheme for the hidden 3-bit PHT. The BHB content is xor-folded and combined with the virtual address of a branch that requires prediction.

4.4 3-bit PHT Aliasing

In the same way that two unrelated branch instructions can index the same PHT entry in the Bimodal predictor, a similar aliasing phenomenon occurs when indexing the hidden 3-bit PHT.

We evaluate the conditions that lead to *aliasing* in the 3-bit PHT by programming identical copies of the `pht3bit_branch` function (Listing 2) with different alignments. To be precise, we designate a *reference* function where we do *Prime+Probe*, and program an identical *aliasing* function that is 2^n bytes apart from the *reference* in the virtual address space. We do the same with an additional *aliasing* function that is 2^{n+1} bytes apart, another one 2^{n+2} bytes, and so on.[6] This works because when two instructions are 2^n bytes apart in the virtual address space, the n LSBs in their respective virtual addresses are identical. That being so, we test which *aliasing* functions interact with the *reference* function to discover and verify how many LSBs (x) of a branch's virtual address (*source*) are used to update the BHB during *History Saturation*:

n	***Prime***	***Aliasing***	***Probe***	***Probe* M. Rates**
$x-1$	TTTTTTT	NNNNNNN	NNNN	M M M U
x	TTTTTTT	NNNNNNN	NNNN	H H H H

Where Table 4 reports the x values for all the test CPUs at hand.

We further observe that the BHB bit size (y) is extremely high to directly index the hidden 3-bit PHT, e.g., $y \approx 186$ bits in our Intel CPUs. Hence we theorize that the 3-bit PHT has a reasonably small size, and that its indexing is handled by an undocumented xor-folding scheme $F : y \rightarrow z$, where z depicts the real amount of bits that index the hidden 3-bit PHT of size 2^z, as illustrated in Figure 6. Ultimately, the lack of information about this xor-folding circuitry does not prevent us from effectively exploiting particular counter entries in the 3-bit PHT as a covert-channel.

4.5 Monitoring the 3-bit PHT via the TSC

In view that PMCs must be configured by code in the kernel-space, an unprivileged adversary cannot directly use the `pht3bit_branch` function in Listing 2 as-is. Yet, it is still possible to monitor the time penalty of branch mispredictions via the processor's Time-Stamp Counter (TSC), which functions as a general cycle counter. The modification is done by replacing the `rdpmc` instructions in Listing 2 with unprivileged `rdtsc` instructions that can detect the cycle count increase caused by branch mispredictions. For instance, in Skylake it takes approximately 51 cycles to execute the `if` statement in line 5 when a branch prediction hit (`H`) occurs, and around 78 cycles on a branch misprediction (`M`).

[6]This can be easily done in C by using the `aligned(n)` attribute for functions.

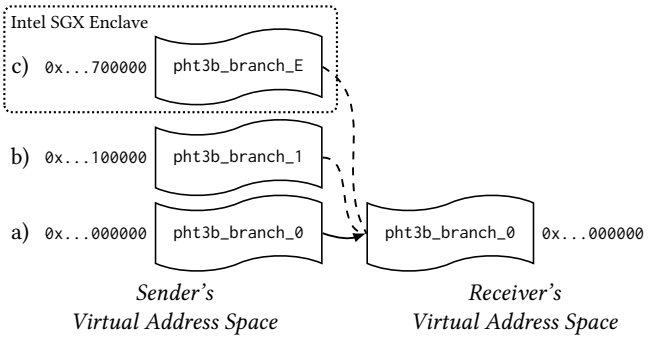


Figure 7: Configurations for a covert-channel in the 3-bit PHT. a) By cloning processes via `fork`, b) Between aliasing functions. c) With a sender inside an Intel SGX Enclave.

5 COVERT-CHANNEL

The BPU is a hardware component shared among all the processes that execute in a physical execution core, meaning that its subcomponents can be exploited as covert-channels to break inter-process isolation. We now describe how to configure a covert-channel on top of a counter entry in the 3-bit PHT.

The 3-bit counter covert-channel is established between two processes that act as *sender* and *receiver* agents, where both use the `pht3bit_branch` function (Listing 2) to manipulate a specific counter in the hidden PHT. Here, it is possible to use the `fork` system call to align the respective `pht3bit_branch` functions by cloning the virtual address space layout from one of the processes, as depicted in Figure 7.a. Furthermore, the two agents can also communicate by using aliasing functions, as shown in Figure 7.b, and even establish illegitimate communication via a custom sender function (`pht3b_branch_E`) that is protected by an Intel SGX Enclave, as in Figure 7.c. In the end, the only requirement is that both processes execute in the same execution core, and if the core implements Simultaneous Multi-Threading (SMT) technology, each process can even execute in one of the core's physical threads.

Information is transmitted one bit at a time by using the two strong states of the FSM part of the 3-bit PHT. Therefore, both agents must synchronize such that the *receiver* can *Prime+Probe* the counter to recover the bit information written by the *sender*. In our example, the *sender* communicates a "1" by setting the FSM into its Strong-*Non-Taken* state, and a "0" via the Strong-*Taken* one. By using PMCs the *receiver* agent just needs to *Probe* once to detect the corresponding branch prediction miss (`M`), or hit (`H`):

Bit	***Prime***	**FSM**	***Sender***	**FSM**	***Probe* (H/M)**
0	TTTTTTT	ST	TTTTTTT	ST	N (M)
1	TTTTTTT	ST	NNNNNNN	SN	N (H)

We evaluate this covert-channel configuration in our Skylake CPU by performing three tests: Send only "0" bits, send only "1" bits, and send random bits. For each test, we transmit 1 million bits to compute the following metrics:

Test	Accuracy	Total time (s)	bits/s
All "0"	99.9993%	1.869	534996
All "1"	99.9971%	1.860	537748
Random	99.9966%	1.855	539004

Where we remark that this evaluation is done in a mostly-idle system, with no noise-canceling techniques such as isolating the tested execution core.

5.1 Communication with an Intel SGX Enclave

Intel's SGX technology [6] is a hardware solution to provide a trusted execution environment for critical pieces of code. SGX works by reserving a protected range of physical memory that is mapped to a range of a process' virtual address space known as an *Enclave*. By design, communication with code inside Enclaves is strictly regulated by a well defined API, and no code outside the Enclave can read or write the protected address range. However, SGX is susceptible to hardware side-channel attacks configured on multiple components of the CPU, such as the cache [10, 19], and even the BPU itself as proved with BranchScope attacks [9].

Naturally, *Branchboozle* can also be used to establish illegal communication with code inside an Enclave, as depicted in Figure 7.c, where the only requisite to configure this covert-channel is to make slight modifications to the `pht3bit_branch` function of Listing 2. To be precise, `rdpmc` and `rdtsc` instructions cannot be executed inside Enclaves [20], and thus must be removed and replaced with `nop` instructions that act as padding. The shortcoming of this configuration is that the modified function (`pht3b_branch_E`) is restricted to only send information.

6 SPECULATION ATTACKS

Kocher et al. disclose the Spectre Variant 1 attack [16] that works by prompting the BPU to mispredict the direction of a conditional branch, in turn causing the execution core to erroneously execute *transient instructions* whose results are ultimately discarded. A devoted adversary can exploit this phenomenon since it has been proved that the execution core can incorrectly by-pass fundamental security checks when executing transient instructions. Furthermore, Spectre can even be used to trick a victim program into leaking its own private data via a hardware side-channel.

6.1 The `if` Statement

To describe the Spectre Variant 1, Kocher et al. [16] consider the case of a basic `if` statement in a victim function that performs a security bounds check before accessing a data array:

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

Here, it is assumed that the `x` variable is an input that can be freely manipulated by an external agent.

The adversary's objective is to trick the execution core to transiently execute the `if` statement's body by using an invalid `x` value. greater than the size of `array1`. To achieve this, the adversary issues valid `x` values multiple times to *train* the BPU's directional prediction mechanisms, causing it to assume that the condition of this `if` statement will be `true` in the future. Having done this, the adversary now inputs an invalid `x` value, which invariably results in the `if` statement's condition being `false`. However, the core can still go into a so-called *speculation mode* to transiently execute the `if` statement's body ahead of time. As a consequence, the core makes an out-of-bounds access to `array1` and leaves a trace in the CPU's cache memory by accessing `array2`. This cache trace can potentially leak the data that was erroneously accessed during the out-of-bounds read, and such data can be recovered with a cache examination technique such as *Flush+Reload* [22].

6.2 The Branch Misprediction Origin

A study on transient execution attacks by Canella et al. [4] categorizes Spectre Variant 1 [16], and its sub-variant 1.1 [15], as Spectre-PHT attacks. The category is established on the observation that an adversary *trains* the BPU by manipulating a counter in some PHT to ultimately trigger the directional misprediction. Nonetheless, with *Branchboozle* we demonstrate the implementation of a hidden 3-bit PHT in all our test CPUs, and by implementing BranchScope [9] we confirm that there are two distinct PHT sub-components in modern BPUs, each with its own geometry and indexing scheme. Therefore, we propose two new Spectre sub-categories depending on the PHT that actually issues the directional misprediction, namely the Spectre-PHT-2b sub-category for Spectre attacks that explicitly target the 2-bit PHT in the Bimodal predictor, and a Spectre-PHT-3b sub-category for attacks that leverage mispredictions issued by the hidden 3-bit PHT, as reported in this work.

6.3 Triggering Speculation with *Branchboozle*

Branchboozle can generate branch mispredictions to trigger transient execution. The principle behind this speculation attack is similar to that of our `pht3bit_branch` function in Listing 2, where *History Saturation* (Listing 1) forces the BPU to issue a 3-bit PHT prediction. Following the example of Kocher et al. [16], a potential victim function executes a `for` or `while` loop before executing the `if` statement that triggers transient execution:

```
1 void victim_function(unsigned int x) {
2   /* Accidental History Saturation. */
3   for (int s = 0; s < 100; s++) {...}
4   ...
5   if (x < array1_size)
6     y = array2[array1[x] * 4096];
7 }
```

Therefore, an adversary only needs to invoke the victim function enough times with a valid `x` input to manipulate the corresponding 3-bit PHT counter that has been indexed by the stagnate BHB. As a result, the BPU will make a directional misprediction when the adversary finally provides an illegal `x` input. In this case, however, the misprediction explicitly originates from the hidden 3-bit PHT and the corresponding counter entry, and not from the 2-bit PHT part of the Bimodal predictor.

We investigate how *Branchboozle* can potentiate Spectre by evaluating the effectiveness of manipulating the 3-bit PHT to trigger transient execution. To do so, we test two versions of the victim function described above, one with *History Saturation* and one without it. Then, we perform the entire attack sequence multiple times within a `for` loop to allow the BPU to capture the `if` statements' branch direction pattern. The pseudo-core of our test is as follows:

```
for (int r = 0; r < 1000; r++) {
  /* Train the BPU. */
  victim_function(valid_x);   \\ 1
  ...
  victim_function(valid_x);   \\ 7
  flush();
  victim_function(invalid_x); \\ Spectre
  secret = reload();
}
```

Where `array1[invalid_x]` references a memory address that contains a `secret` value that can be recovered from the cache side-channel via *Flush+Reload* [22].[7] In the end, for a thousand repetitions we observe that `secret` can be recovered a $99 - 100\%$ of the time when implementing *History Saturation*, whereas with the baseline victim function the `secret` value can only be recovered with a $0.001 - 0.003\%$ success rate since the BPU can quickly capture the `if` statement's repetitive direction pattern.

6.4 Out-of-Place Training

An adversary does not need to explicitly invoke the victim function with valid x values to train the BPU. In the case of seminal Spectre-PHT attacks, the adversary can train the BPU by "flooding" the multiple PHTs with numerous branch executions. The system of Canella et al. [4] refers to this procedure as *Out-of-Place Training*.

We prove that it is also possible to train the BPU by executing unrelated branches that have been properly aligned to alias with a vulnerable victim branch in the hidden 3-bit PHT (§ 4.4). Our testing shows that this form of training does not diminish *Branchboozle*'s effectiveness to trigger transient execution.

7 DISCUSSION

We have identified a hidden 3-bit PHT part of modern BPU designs, which exists in both Intel and AMD micro-architectures. With *Branchboozle* we have demonstrated how this sub-component can be utilized to construct a powerful covert-channel, and to easily trigger transient execution to facilitate Spectre-like attacks.

Branchboozle emanates from the adverse effects of *History Saturation*, where the execution behavior of a lengthy sequence of conditional branches (e.g., `loop`) degrades the BHB content used to predict subsequent branches. This phenomenon indirectly deprives some conditional branch instructions from the full prediction benefits offered by state-of-the-art BPU designs.

7.1 Impact

Based on our experimentation we consider that *Branchboozle* can configure an exceptionally powerful covert-channel across processes. Additionally, we know that branches placed right after lengthy loops, which involuntarily stagnate the BHB content, change the state of specific entries in the hidden 3-bit PHT, which can be observed by an adversary via a *Prime+Probe* sequence. These vulnerable branches may exist on cipher block implementations, and even within the Linux kernel itself. Yet, identifying specific examples and exploiting such potential vulnerabilities remains as an interesting research vector.

[7] Recovering data from the cache-side channel is a topic out of scope.

7.2 Software Mitigation

It is already considered a bad practice to make a program's control flow dependent on secret values, a.k.a. *branching on secrets*. If for any reason a programmer needs to determine the conditions of an `if` statement based on sensitive information, it is imperative that such a conditional statement does not immediately succeed high-iteration loops that may stagnate the BHB content. Otherwise, a devoted adversary can mimic the branch direction sequence from such a loop to easily *Prime+Probe* the corresponding entry in the 3-bit PHT, thus revealing the victim's secret values.

7.3 Related Work

Evtyushkin et al. [8] defeat Address Space Layout Randomization (ASLR) by attacking the Branch Target Buffer (BTB) part of the BPU's *target* prediction facility. The attack works by identifying *collisions* that occur when the branch instructions of two processes, or a malicious process and the kernel, index the same BTB entry (*BTB aliasing*), causing a time penalty that can be identified via the `rdtsc` instruction. However, the attack was only proved feasible in the Haswell micro-architecutre [8], and more recently Gruss et al. [11] report that the attack fails in Skylake CPUs. In contrast, we have proved that *Branchboozle* works in even newer Intel micro-architectures, such as Coffee Lake.

Additionally, Evtyushkin et al. [7] propose a simple covert channel through the BPU that works by manipulating most, if not all, of the entries in one of the BPU's multiple PHTs. Here, the sender agent sends "1" by executing a bulk of *Taken* conditional branches, and a "0" via *Non-Taken* ones. Then, the receiver measures the execution time of its own bulk of *Taken* branches, and infers the transmitted bit by measuring the performance penalty introduced by the *Non-Taken* branches carried out by the sender. Analog to this covert-channel configuration, we demonstrate that *Branchboozle* is capable of configuring a covert-channel by manipulating the strong states of the FSM part of the hidden 3-bit PHT, which can be quickly done by performing *History Saturation*.

More recently Huo et al. [13] propose a BHB-based Bluethunder attack against Intel SGX enclaves, where the authors also note that the BHB's rolling-hash function is only updated when (secret-dependent) *Taken* branches are executed. However, their attack is mounted on top of the SGX-Step framework [21] which requires an extremely powerful adversary model capable of deploying code in the kernel-space and altering the kernel's normal functionality. In comparison, we have demonstrated how *History Saturation*, in both Intel and AMD CPUs, can directly expose the hidden 3-bit PHT to mount an unprivileged inter-process covert channel, and additionally potentiate transient execution attacks.

8 CONCLUSION

We have presented *Branchboozle*, a direct attack on a hidden 3-bit PHT part of modern BPU designs. With the assistance of *History Saturation*, an adversary can freely interact with this PHT's internal FSM to manipulate the prediction of conditional branches. This capability allows an adversary to establish a fast and robust covert-channel across processes, and even potentiate Spectre-like attacks based on branch misprediction.

REFERENCES

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 312–320.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 225–242.

[3] Gunjeet Baweja and Harsh Kumar. 2001. Branch prediction architecture. US Patent 6,332,189.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 249–266.

[5] Intel®Corporation. 2019. Intel®64 and IA-32 architectures software developer's manual combined volumes 3A, 3B, 3C, and 3D: System programming guide.

[6] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[7] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: a feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. 1–8.

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

[9] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[10] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.

[12] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[13] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2020. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 321–347.

[14] Advanced Micro Devices Inc. 2017. Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors. http://developer.amd.com/wordpress/media/2017/11/54945_PPR_Family_17h_Models_00h-0Fh.pdf.

[15] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[17] Johnny KF Lee and Alan Jay Smith. 1984. Branch prediction strategies and branch target buffer design. *Computer* 1 (1984), 6–22.

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

[19] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 69–90.

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

[21] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[22] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

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