Title

Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor.

Permalink

https://escholarship.org/uc/item/6pp6577n

Authors

Li, Luyi
Yavarzadeh, Hosein
Tullsen, Dean M

Publication Date

2024

Copyright Information

Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor

Luyi Li[*] Hosein Yavarzadeh[*] Dean Tullsen

University of California San Diego

[*] Equal contribution joint first authors

Abstract

This paper introduces novel high-precision Branch Target Injection (BTI) attacks, leveraging the intricate structures of the Indirect Branch Predictor (IBP) and the Branch Target Buffer (BTB) in high-end Intel CPUs. It presents, for the first time, a comprehensive picture of the IBP and the BTB within the most recent Intel processors, revealing their size, structure, and the precise functions governing index and tag hashing. Additionally, this study reveals new details into the inner workings of Intel's hardware defenses, such as IBPB, IBRS, and STIBP, including previously unknown holes in their coverage. Leveraging insights from reverse engineering efforts, this research develops highly precise Branch Target Injection (BTI) attacks to breach security boundaries across diverse scenarios, including cross-process and cross-privilege scenarios and uses the IBP and the BTB to break Address Space Layout Randomization (ASLR).

1 Introduction

Modern processors include extensive support for predicting the outcome of control flow operations. As the potential for transient execution attacks [14, 19, 24, 29, 31, 32, 37, 52, 60, 69] has become apparent, these structures have become a favored target for direction poisoning [24, 29, 31, 52, 69] and target injection attacks [14, 31, 32]. Most of those attacks have been launched with limited knowledge of the actual structure of those predictors, and were thus forced to rely on heavyweight brute force algorithms to discover and create the type of aliasing necessary to launch an attack.

Of particular concern are Branch Target Injection (BTI) attacks, which dramatically open up the potential gadgets that can be employed to exploit a vulnerable branch instruction. These attacks exploit the structures responsible for predicting the target address of taken branches. In modern processors, those structures are the Branch Target Buffer (BTB), the Indirect Branch Predictor (IBP), and the Return Stack Buffer (RSB). This paper examines the first two (BTB and IBP), with particular focus on the IBP because it is the least documented and least understood of all the branch prediction structures.

Indirect branches are branches whose address is computed at runtime, with the address loaded from a register or memory. They can be used for switch and case statements, for calling functions passed as arguments, for object-oriented languages that can overload function calls, etc. While direct branches always have the same target (and are easily predicted with simple structures), accurate prediction of indirect branches requires capture of past history patterns [14, 31, 44, 49, 59, 69, 70], much like conditional branch predictors, and requires more complex structures.

For the first time in the literature, this work reveals exactly how the BTB and the IBP cooperate to make a prediction of indirect branches on Intel processors. It also reveals the precise structure of the IBP, including how history is captured, how that history is used to index the several tables of the IBP, how the tags are formed from the history and the branch address, how many bits of the target are stored and how many are assumed, and lastly, the size, associativity and structure of each table in the IBP.

This information enables several new, high precision attacks on the table itself – these include two new target injection attacks and a new technique to break ASLR. While these machines can capture nearly 400 bits of branch history (representing a huge search space), precise understanding of how that history is folded down to index one of the actual tables allows us to instead systematically probe only 512 sets, incurring no wasted computation – even in the cases where we must use brute force methods, for example in aliasing a branch of unknown address or history. Being able to launch these attacks with orders of magnitude greater efficiency has immense implications. Current mitigations against these attacks are heavyweight and costly. Ultimately, the frequency at which such mitigations need to be deployed is directly connected to the minimum runtime of a successful attack. Reducing the runtime of these attacks by orders of magnitude means that the perceived performance cost of these mitigations to the user can go from slightly noticeable to unpalatable, and for all practical purposes render these solutions no longer viable for any but the most protected code.

Intel has provided several mitigation mechanisms aimed at protecting the BTB and IBP from different types of target injection attacks, which vary in goals, actual operation, and performance implications. These include Indirect Branch Restricted Speculation (IBRS) [8], Single Thread Indirect Branch Predictors (STIBP) [3], and Indirect Branch Predictor Barrier (IBPB) [7]. This work carefully deconstructs each of these mitigations on both pre-Spectre and post-Spectre Intel processors, and reveals for the first time in the literature the precise actions employed by each. Those actions vary by machine, and do not always correspond closely to their advertised goals. We also describe some surprising attack surfaces that remain uncovered by these mitigations.

Contributions. The contributions of this work are:

- It presents the first comprehensive analysis of the Indirect Branch Predictor and its interaction with the Branch Target Buffer in the recent Intel processor families, detailing the size, structure, and precise indexing and tagging hash functions of each table.

- It carefully analyzes mitigation mechanisms (IBRS, STIBP, and IBPB) designed to protect against BTB and IBP target injection attacks on both pre-Spectre and post-Spectre Intel processors, revealing for the first time in the literature the specific actions employed by each mechanism.

- It proposes iBranch Locator, an efficient and high-resolution tool capable of locating any indirect branch within the IBP without requiring prior history information about the branch. Using this tool, it introduces high-precision target injection attacks and successfully breaks address space layout randomization.

Outline. Section 2 provides background and related work. Section 3 delves into the analysis of the BTB and IBP structures. Section 4 deconstructs the Intel hardware defenses against BTI attacks. Section 5 introduces our attack primitive to extract the index and tag of any victim indirect branch. Section 6 describes our new target injection attacks. Section 7 discusses potential mitigations, and Section 8 concludes the paper.

Responsible Disclosure. We disclosed our findings to Intel before submitting to USENIX Security 2024.

2 Background and Related Work

This section provides relevant background information about the branch prediction mechanism in modern high-performance processors, focusing on Indirect Branch Predictors (IBP) and their role in providing target addresses for indirect branch instructions. We review state-of-the-art IBP structures, followed by a discussion of the most relevant research on branch-based side-channel attacks and mitigations.

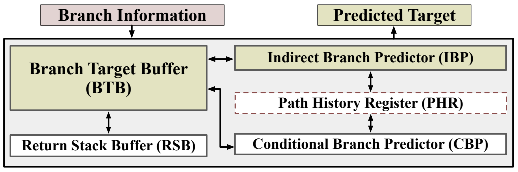

Branch Information
Predicted Target
Branch Target Buffer (BTB)
Indirect Branch Predictor (IBP)
Path History Register (PHR)
Return Stack Buffer (RSB)
Conditional Branch Predictor (CBP)

Figure 1: Branch Prediction Unit Internals in Modern CPUs.

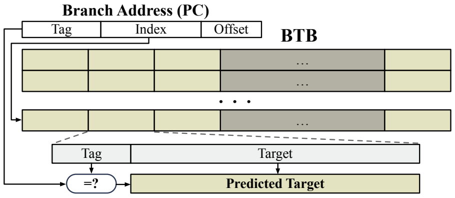


Figure 2: Overall Structure of the Set-Associative BTB.

2.1 Branch Prediction

The Branch Prediction Unit (BPU) plays a crucial role in modern high-performance processors by steering the processor front-end pipeline through the identification of branch instructions, their directions, and target addresses. To provide predictions for the variety of control flow mechanisms, the BPU incorporates various dedicated structures, as Figure 1 illustrates [2,14,24,29,31,38,70]. Among these are the Branch Target Buffer (BTB) [28], which predicts whether an instruction is a branch instruction and, if so, predicts the target address. Additionally, the Indirect Branch Predictor (IBP) [44] serves as a dedicated unit for predicting the target addresses of indirect (computing) branches—those whose targets reside in registers or memory. Furthermore, a Conditional Branch Predictor (CBP) [50, 70] is employed to determine whether conditional branch instructions will be taken or not-taken.

Branch Target Buffer (BTB): The BTB keeps track of the recently executed target addresses of branch instructions and predicts the occurrence of future branch instructions along with their targets. Earlier research [2,5,23,27,41,58,59,74] reverse engineered the BTB structures of several modern CPUs, revealing that they each exhibit an almost identical cache-like set-associative structure (shown in Figure 2), albeit with varying sizes. In Section 3, we delve into a more detailed examination of the BTB's structure in high-end Intel processors.

Indirect Branch Predictor (IBP): In addition to the BTB, there is a dedicated unit specifically designed for predicting the target addresses of indirect branch instructions, known as the Indirect Branch Predictor (IBP) [44]. Indirect branches, also known as computed branches, differ from direct branches

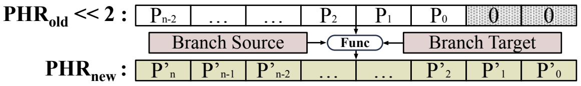


Figure 3: PHR Update Process: (1) PHR is first left-shifted by 2 bits and then (2) combined with branch and target addresses via a *Function*, detailed in Appendix A.

in how they determine the next instruction to execute. Instead of specifying the target address directly, indirect branches rely on a value stored in a register or memory location, which is calculated at runtime. This approach enables dynamic control flow patterns, such as switch-case statements in high-level languages. Predicting the target address of these branches is challenging because they may have multiple targets, and cannot be simply predicted by the BTB (which captures a single address). Consequently, a dedicated unit, such as the IBP, is necessary to accurately predict their target addresses.

The state-of-the-art IBP design in the literature is known as Indirect Target Tagged GEometric length (ITTAGE) [49] predictor, which uses history-based structures [39, 48, 50] to predict the targets of indirect branches. This prediction is made based on the global history which captures the record of executed branch instructions within the processor. There exist two forms of global history: the first is the Global History Register, or GHR - inserting 1 when a branch is taken and 0 when it is not. The second form is the Path History Register, or PHR, where updates are influenced by one or more bits in the branch source and/or target address. The ITTAGE predictor comprises a tagless base predictor responsible for delivering a default prediction along with a set of (partially) tagged predictor components, each indexed through independent functions of the global history and the branch address.

Path History Register (PHR): In 2018, Spectre attacks [31] revealed the potential for exploiting CBP or BTB/IBP mispredictions, coupled with speculative execution, for malicious purposes. Fundamentally, Spectre attacks capitalize on branch mispredictions to execute code that can lead to the leakage of sensitive data. Specifically, they reverse-engineered some structures of the IBP designed for Haswell architecture including the global history (PHR). They found that the PHR tracks taken branches and gets updated using the corresponding branch and target addresses, as Figure 3 illustrates. We summarize the latest findings (based on [69, 70]) about the PHR update policy in Appendix A, and verify that these findings still hold true for more recent Intel processors not covered in earlier studies.

It is important to highlight that some earlier studies [14] claimed the existence of only one BTB with different indexing schemes. Nonetheless, our reverse engineering work unveils that the IBP incorporates a distinct and separate structure, a topic we will delve into in Section 3.

2.2 Branch-based Side Channel Attacks

We categorize branch-based side-channel attacks into two main types based on the specific BPU structure they target:

Direction based attacks: These attacks [19, 22, 24, 31, 52, 69] aim to extract or manipulate the control flow of a victim process by analyzing or influencing the CBP state. The most representative attack is Spectre v1 [31]. This can reveal confidential information or steer the victim's execution towards unintended paths.

Target based attacks: These attacks attempt to directly hijack and/or extract the victim's control flow by poisoning entries in the BTB and/or IBP. Early research on branch-based vulnerabilities primarily targeted the BTB [9–11]. These attacks demonstrate a timing side channel based on BTB capacity. Following the initial BTB-focused research, Evtyushkin et al. [23] leveraged BTB collisions to bypass Kernel Address Space Layout Randomization (KASLR) by colliding entries between an unprivileged user process and privileged kernel code. Similarly, Lee et al. [34] exploited BTB collisions to infer the control flow of victims.

More recently, in Spectre v2 [31], also called Branch Target Injection (BTI), the attacker trains the target predictors (BTB and/or IBP) so that the victim program speculatively jumps to a chosen disclosure gadget. Through speculative execution of the instructions within the gadget, the attacker gains access to the victim's confidential data, subsequently transferring it via the data cache [26, 56, 68] or other covert channels [17, 22, 45, 47, 65, 76].

Similar to Spectre v2, Barberis et al. [14] introduce Branch History Injection (BHI), a cross-privilege BTI attack. Since the branch target entries are isolated between kernel and user, Spectre v2 is difficult to directly apply in a cross-privilege scenario, targeting a kernel branch. This work finds that the branch history state (PHR) is still shared across privilege levels. Therefore, the attacker manipulates the branch history in user space and further misleads a kernel indirect branch to a prepared malicious kernel gadget during speculation. In a concurrent study, InSpectre Gadget [64] demonstrates a native Spectre-v2 exploit targeting the Linux kernel on recent-generation Intel CPUs, leveraging the latest BHI variant [14].

2.3 Defenses Against Branch-based Attacks

The broadest defense against contention-based side-channel attacks involves partitioning the targeted unit across various processes or domains [35, 54, 71]. This principle also extends to general defenses against other BPU attacks [25, 42, 62, 70, 75, 78]. Another approach is to simply flush the target predictor state when context switching [7, 53, 62]. An alternative defense utilized by sensitive code, such as cryptographic libraries, is constant-time programming [15, 21]. This strategy removes secret-dependent branches. However, it's rarely adopted in general-purpose code due to its high overhead.

To mitigate Spectre-style attacks targeting either the direction predictor (CBP) or the target predictors (BTB and IBP), several defense strategies have been proposed [3,4,6–8,19,70]. These include inserting fences in programs [30,40,51,61], speculative load hardening [20], or artificial data dependencies [43] near branches, measures to restrict the impact of speculation on structures such as the cache [12,46,66,67], limitations on speculative execution when handling sensitive data [55], and even alterations to CPU design to enable safe speculation [13,36,63,72,73]. Retpoline [4,6] is a mitigation technique designed by Google to counter Spectre v2 attacks. It works by altering how indirect branches are handled in code, introducing a trampoline code sequence that traps any potential mis-speculation into an infinite loop until the actual address is resolved and popped from the return stack buffer (RSB). Intel has proposed several hardware defenses against target injection attacks, including IBRS [8], STIBP [3] and IBPB [7]. We will elaborate on them in Section 4.

3 Analysis of Indirect Branch Prediction

In this section, we present a detailed analysis of our reverse engineering efforts on branch target prediction mechanisms in modern Intel processors. This includes reverse engineering the detailed structures of the Branch Target Buffer (BTB) and the Indirect Branch Predictor (IBP) in multiple high-end Intel CPUs. For brevity, we focus on *Raptor Cove* microarchitecture in the text, and only summarize the important information for other microarchitectures.

3.1 Branch Target Buffer (BTB)

The Branch Target Buffer (BTB) plays a crucial role in steering modern CPUs' front-end by not only identifying the branch instructions but also predicting their target addresses. Earlier research [2,23,27,31,34,58,59,74,76] reverse engineered the BTB structures of several modern CPUs. Zhang et al. [74] revealed the mapping policy and the associativity of the BTB, showing that the instruction address of the branch is divided into tag, index, and offset fields to lookup the BTB. Numerous studies [23,28,31,74,76] confirmed that the BTB only stores lower bits of the target address, rather than the entire 48-bit virtual address. The predicted target address is subsequently formed by concatenating these lower bits with the higher bits of the branch address. We replicate these experiments on the recent *Raptor Cove* and *Gracemont* microarchitectures and find consistent results with previous studies, which is summarized in Table 1. In modern CPUs, the BTB not only stores the target address but also records various metadata pertaining to branch instructions, such as branch type and core ID. In the subsequent discussion, we delve into our new observations regarding this information stored within the BTB entries.

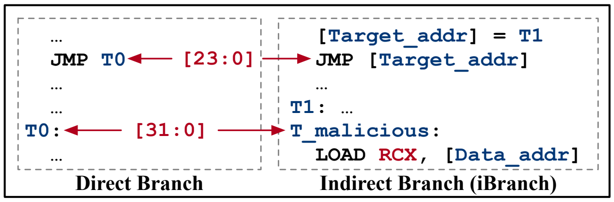
Model	i7-6770HQ	i9-12900 (P-core) i7-13700K (P-core)	i9-12900 (E-core) i7-13700K (E-core)
μArch	Skylake	Raptor (Golden) Cove	Gracemont
Address Type	Virtual	Virtual	Virtual
Index	PC [13:5]	PC [14:5]	PC [14:5]
Tag	PC [29:22] $\oplus$ PC [21:14]	PC [23:15]	PC [24:15]
Associativity	8	12	5
Capacity (Entry)	4K	12K	5K
Target Length (Bit)	32 (Long) / 10 (Short)	32	32 (Long) / 10 (Short)
Predicted Target	{PC, BTB}		

Table 1: Fundamental BTB Details Across Various CPUs.

3.1.1 Branch Type in BTB

In x86 architecture, there are various types of branching instructions including: conditional branch, direct unconditional branch, indirect branch, call, and ret. The BTB needs to record the branch type for each branch instruction since different branch prediction mechanisms (CBP, IBP, or RSB) are used for different branch instruction types. To verify this in Raptor Cove microarchitecture, we create a scenario comprising two branch instructions of different types but sharing the same lower 24 bits of the branch address (thus exhibiting aliasing in the BTB). Subsequently, we measure the branch misprediction rate for both instructions. Our findings reveal that when two branches of different types have distinct target addresses, a misprediction rate of 100% occurs. This suggests that these branches are utilizing the same BTB entry, resulting in inaccurate predictions. However, when they share identical target addresses (lower 32 bits), as expected, no misprediction arises.

Based on this result, we explore whether a direct branch can inject a malicious target into the BTB entry and mislead an aliased indirect branch to speculatively execute it, because the correct target of the indirect branch will not be resolved until the Execution stage. As Listing 1 shows, the indirect branch has only one target. Single-target indirect branches only use the BTB for prediction, which will be explained in Section 3.3. During normal execution, the malicious target is never reached. In order to measure whether the instructions in the malicious target are executed, we introduce a load instruction within it. Before this, we flush the data associated with this instruction out of the cache. After the victim execution, we reload the data to determine if it has been cached. We also flush the correct target of the indirect branch to create a large-enough speculation window. We find that the data associated with the load instruction never gets cached, indicating that the load instruction is never speculatively executed. However, we observe that both the malicious target and the subsequent instruction are cached in the instruction cache. This suggests that these instructions are speculatively fetched and maybe decoded, but are somehow being prevented from executing speculatively. We also tested other types of branches (e.g. conditional branches) to see if they could inject a malicious target into the BTB entry and mislead the aliased indirect branch, and we confirmed that they cannot. Conversely, if both branches are indirect ones, we see cached data (specu-



```
  ...                            [Target_addr] = T1
  JMP T0 <-- [23:0] -->          JMP [Target_addr]
  ...                            ...
  ...                          T1: ...
T0: <--   [31:0]   -->         T_malicious:
  ...                            LOAD RCX, [Data_addr]
```

Listing 1: Microbenchmark Pseudo-Code for Injecting Targets with Different Branch Types. The dashed frame means these are multiple branches in the same thread. The arrow represents specific bits of their virtual addresses are identical. In a single test, they are executed in order from left to right.

lative execution). Thus, there is at least one bit in the BTB metadata that differentiates indirect branches from other types of branches. We believe that in the former scenario (different branch types), speculative execution does not occur due to the presence of the Branch Address Calculator (BAC) unit in the decode stage, along with the BTB bit identify branch type. This unit ensures that the BTB prediction aligns with the actual decoded branch type. This unit empowers modern CPUs to detect potential branch mismatches earlier in the pipeline, well before the target is resolved in the execution stage.

Observation 1. The BTB features a 'Branch Type' bit to aid branch prediction. Since different branch types may alias in the BTB, this check serves to verify that the indirect branch prediction from the BTB aligns with the actual type.

3.1.2 Core-ID in BTB

Modern Intel CPUs support simultaneous multi-threading (SMT) [57], which virtualizes a single physical core into two logical (SMT) cores. Since there is only one BPU per physical core, we aim to determine whether the BTB is shared or statically partitioned between SMT cores. To do that, we create two SMT threads: $Thread_0$ with M direct branches and $Thread_1$ with N direct branches. All branches share the same BTB index but have different tags. Subsequently, we measure the number of BTB entries that can be allocated to each SMT thread. We observe that when $M+N \leq 12$ (where $M,N \in 0,1,2,\ldots,12$), no branch mispredictions occur. This suggests that there are no constraints on the number of entries per set available to each SMT thread and there is no SMT-based static partitioning in place.

Building upon this, we repeat the experiment in Listing 1 but this time the aliased branches run on different SMT cores and have the same branch type. We find that these two branches do not alias in the BTB and no mispredictions occur. This suggests that the BTB stores some information about the core-ID or thread-ID or process-ID. Through exhaustive testing across various scenarios—cross-SMT context, cross-process, and cross-thread—we ascertain that aliasing is possible only if two branches are in the same SMT virtual core, implying that only the core-ID is recorded for each branch instruction in the BTB. This finding also explains why BTB mistraining disappears in cross-SMT scenarios mentioned in BunnyHop [76].

Observation 2. Core-ID is recorded and used as part of the tag in the BTB. When two aliased branches reside on the same SMT virtual core, they can share the same BTB entry, even in a cross-thread or cross-process scenario.

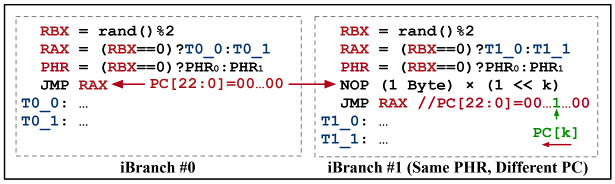
3.1.3 Privilege Level in BTB

In addition to our new findings, we also confirm prior findings, including the fact that there is a *Privilege Level* field in each BTB entry so that the kernel and the user cannot share entries in the BTB [14, 76]. The "Privilege Level" field behaves similarly to the "Branch Type" field. Aliased branches in different privilege levels lead to speculative fetch but not execution.

3.2 Indirect Branch Predictor (IBP)

The Indirect Branch Predictor (IBP) is a dedicated unit designed to predict the target addresses of indirect branches as they can jump to various places within the program. Many recent studies [14, 27, 31, 58, 59, 74] have revealed that indirect branch prediction in modern CPUs relies not only on the branch address but also on the global history, dubbed as Path History Register (PHR) or Branch History Buffer (BHB). Some studies [14, 74] argue that the indirect branch prediction mechanism lacks a distinct structure and is instead implemented through multiple indexing functions to the BTB. Other studies, such as an Intel patent [44] and Uzelac's reverse engineering of the Intel Pentium M processor [58], provide evidence supporting the existence of a separate structure dedicated to predicting indirect branches. Furthermore, the ITTAGE predictor proposed by Seznec et al. [49] in 2011 also features its own distinct structure separate from the BTB, as discussed in Section 2. Since the structure of the IBP within Intel CPUs is completely undocumented, we begin with a small set of assumptions. Specifically, we assume that the IBP adopts a TAGE-like structure, consisting of multiple tagged tables indexed by varying lengths of the global history (PHR). These tables are utilized to store the target addresses of indirect branches.

Prior works [14, 27, 31, 58, 59, 74] show that the IBP is indexed by the global history (PHR). Additionally, recent works [69, 70] have shown complete details of the structure and update policy of the PHR used for the CBP.



```
  RBX = rand()%2
  RAX = (RBX==0)?T0_0:T0_1
  PHR = (RBX==0)?PHR0:PHR1
  JMP RAX <-- PC[22:0]=00…00
T0_0: …
T0_1: …
```

iBranch #0

```
  RBX = rand()%2
  RAX = (RBX==0)?T1_0:T1_1
  PHR = (RBX==0)?PHR0:PHR1
  NOP (1 Byte) × (1 << k)
  JMP RAX //PC[22:0]=00…1…00
T1_0: …
T1_1: …
```

iBranch #1 (Same PHR, Different PC)

Listing 2: Microbenchmark Pseudo-code for Recovering the Number of PC Bits Involved in the IBP.

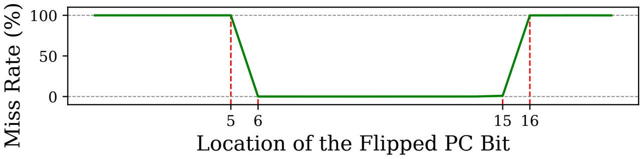


Figure 4: Misprediction Rate of when the Position of Flipped PC bit Varies.

To study the structure, size, and the update policy of the PHR in the IBP, we run a set of similar experiments to those discussed in Half&Half [70]. Our results show that the PHR used in the IBP has the same update policy as the CBP. On *Raptor Cove*, the PHR records the global history of the last 194 taken branches, which means its size is $194 \times 2 = 388$ bits in total because each update shifts it to the left by 2 bits. Building on this, we can set the PHR to any arbitrary value through a carefully aligned series of 194 direct branches, as demonstrated in a recent study [69].

3.2.1 Index and Tag Inputs

First, we explore how many bits of the branch address (PC) are used in the IBP lookup (in addition to the PHR). As shown in Listing 2, there are two indirect branches, each with two targets. The branch targets are determined by a random binary input, each correlated with a specific PHR value. We keep all lower bits of branch addresses identical except for one bit, which we systematically brute force to determine when these two branches collide within the IBP. We measure the misprediction rate for these cases to identify the collision point. Figure 4 illustrates that the miss-rate remains consistently near 0% when the flipped bit pertains to PC[15:6]. However, it increases at other locations. This finding suggests that when PC addresses of the branches differ in PC[15:6], the IBP can effectively distinguish between them and accurately predict their target addresses.

Observation 3. PC[15:6] is used as input (index and/or tag) to the IBP, which differs from the BTB that uses PC[23:0].

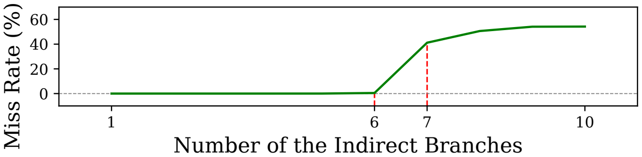


Figure 5: Misprediction Rate when the Number of Indirect Branches with the Same PHR Value Varies.

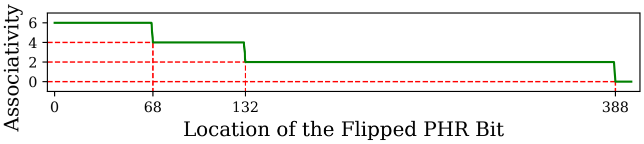


Figure 6: Associativity of the IBP when the Position of Effective PHR bit Varies.

3.2.2 Associativity

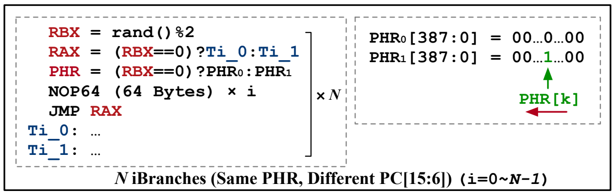
As introduced earlier, the ITTAGE predictor has multiple set-associative tables, each table indexed with a different length of global history. We investigate if the IBP in Intel processors has a similar structure. First, we seek to determine how the PC is used in IBP lookup and hence determine if it is used in tag and/or index. To do that, we create a scenario where there are N indirect branches with the same PHR but different PCs (bits[15:6]). We increase the number of branches and measure the miss-rate for them.

For a K-way associative structure, once the number of branches exceeds K, the miss rate shows a steep rise, as a single set lacks empty entries to capture the $(K+1)^{th}$ branch and must replace an old branch, leading to a misprediction.

We exhaustively test all the combinations of PC[15:6] values and our results (See Figure 5) show that the IBP can predict up to only 6 branches even as we change PC[15:6] values. This suggests that the PC bits are not used in the index but used as **tag** and the IBP has a set-associative structure with a maximum associativity of 6, indexed by the PHR.

Next, we want to find whether the associativity varies with the effective PHR length. In Listing 3, all N indirect branches have the same PHR setup: PHR_0 is set to 0 while PHR_1 is modified during each test by incrementally shifting a single bit from LSb to MSb. Each branch has a unique PC[15:6] (tag) to avoid entry sharing. We test enough permutations of PC [15:6] (giving us same index, different tags) to exhaust the associativity of that set and start seeing mispredicts. We find the associativity does vary, depending on where in the PHR the one-bit difference exists between the PHR_0 and PHR_1. When the difference is in a lower bit, there are six entries available to record a branch target. When it is in the middle of the PHR, there are four; when the difference is in a high bit (distant correlation), there are only two.

These results, then, shown in Figure 6, indicate that the IBP consists of three tables, each a 2-way set associative structure.



```
   RBX = rand()%2
   RAX = (RBX==0)?Ti_0:Ti_1
   PHR = (RBX==0)?PHR0:PHR1
   NOP64 (64 Bytes) × i
   JMP RAX
Ti_0: …
Ti_1: …
```

Listing 3: Microbenchmark Pseudo-Code for Recovering the IBP Associativity.

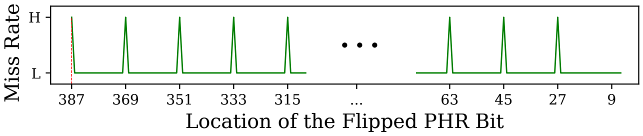


Figure 7: Position of Odd PHR Bits Folded with PHR[387] in the Index Hash Function.

The first table is indexed by the global history of the last 34 branches, i.e. the lowest 68 bits of the PHR. The second table uses the last 66 branches' history (PHR[131:0]). And the last table uses the entire global history (PHR[387:0]).

Observation 4. The IBP on *Raptor Cove*, like ITTAGE, features a set-associative structure with three tables, each being 2-way set associative indexed with different global history lengths.

3.2.3 Index Hash Function

Next, we find the index hash function for each table in the IBP, using the last table as an example. We use two groups of indirect branches. The first group consists of a single branch with the highest bit in the PHR, i.e. PHR[387], set to 1, while the remaining bits are 0. The second group contains $N (N = 2,4,6)$ indirect branches, each using a cleared PHR with a single bit set to 1. Inspired by prior studies [69, 70] on the CBP, the PHR odd and even bits might be folded separately. Therefore, we start by focusing initially on just the odd bits in the PHR. During the test, the location of the set bit of the second group shifts from PHR[387] to PHR[1] with a step of 2. N starts at 2, then changes to 4 when the flipped bit is in PHR[131:68] — this range is within the history span of the second table but not the first. Finally, N increases to 6 when the flipped bit falls within the range of all three tables.

When the bit in the second PHR is not folded with bit 387, we expect a low misprediction rate as the first branch will map to a different set than the second group of branches. When the bits are folded together in the index function, they map to the same index and there will be an eviction, causing a branch misprediction. Figure 7 shows a noticeable spike at some locations. This identifies bits folded with PHR[387]. Moreover, they are spaced uniformly, with an interval of 18 bits. Changing the flipped bit in the first branch maintains this folding pattern, with a fixed distance of 18 bits. However, none of the lowest 8 odd bits are folded with any higher bits. No matter what these 8 bits are, when their higher PHR bits match or differ by an interval of $18 * i (i = 0,1,2,\ldots)$, they map to the same set. This indicates the lowest 8 odd bits are not used in the index. However, the IBP is still able to distinguish the PHR states with just these lowest bits set, because they are used as tag, which will be discussed in Section 3.2.4. Similarly, the even bits also exhibit this folding function with each other.

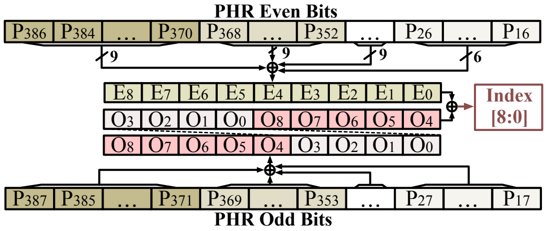


Figure 8: 9-bit Index Hash Function for the Last IBP Table.

Finally, we aim to understand how the odd and even bits are combined to form the final index. We conduct similar experiments to investigate this, with one branch group only setting odd bits and another group only setting even bits. The final index hash function is illustrated in Figure 8. First, all odd and even bits are folded into two 9-bit values respectively. Next, the 9-bit value generated by the odd bits is rotated 5 bits to the left and then folded with another 9-bit value to generate the final index. The remaining two tables in the IBP also employ the same index hash function (but on fewer PHR bits).

Observation 5. Each IBP table is indexed with a 9-bit index, generated from a dedicated hash function folding together the odd and even bits in the PHR.

3.2.4 Tag Hash Function

Because the IBP tables are all set associative, we know they must be tagged structures. We first investigate if, and how, the PHR factors into the tag formation. Take two indirect branches with the same PC[15:6] as an example. Their PHRs are set to all zeroes except PHR[k] and PHR[$k + 18 * i$] respectively. k is an odd number here. Based on the index hash function both branches use the same IBP set. From Section 3.2.2 we already know that PC[15:6] impacts the tag for the IBP. If the PHR is not used as tag, we will observe misprediction because it is a IBP hit scenario for both branches and leads to entry aliasing. However, the result shows that misprediction only occurs when $i = 0$ or 11, indicating the PHR is used as tag but with a different folding distance than the index.

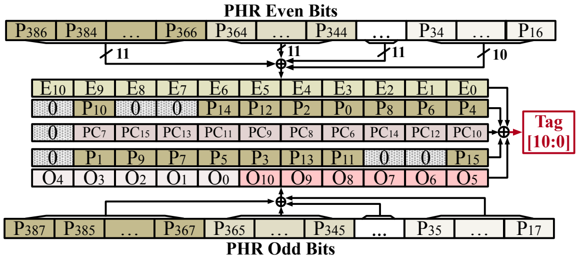

PHR Even Bits
P386 P384 … P366 P364 … P344 … P34 … P16
11
11
11
10
E10 E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
0 P10 0 0 P14 P12 P2 P0 P8 P6 P4
0 PC7 PC15 PC13 PC11 PC9 PC8 PC6 PC14 PC12 PC10
0 P1 P9 P7 P5 P3 P13 P11 0 0 P15
O4 O3 O2 O1 O0 O10 O9 O8 O7 O6 O5
Tag [10:0]
P387 P385 … P367 P365 … P345 … P35 … P17
PHR Odd Bits

Figure 9: 11-bit Tag Hash Function for the Last IBP Table.

Mispredictions only happen when the PHRs are equivalent under a 198-bit distance folding. Further, a 198-bit tag seems unlikely.

Prior works [39] on the predictor tag design offer us some insights. There are indications that the tag hash function uses a different folding distance of 22. The least common multiple of 18 and 22 is 198, in line with the observed folding pattern. We verify this assumption by setting bits PHR[k] and PHR[$k+22*j$] for the first branch and PHR[$k+18*i$] and PHR[$k+18*i+22*j$] ($k=2n+1, n=8,9,10,\ldots$) for the second. A high misprediction rate occurs again under this PHR configuration because the two branches share the same IBP index under a 18-bit distance folding and the same tag under another 22-bit distance folding, causing them to share the same IBP entry. We run similar experiments on the even bits and find the same folding pattern.

It remains possible that PC[15:6] is also folded with the PHR to generate the final tag. Consider two branches with their PHR states set at PHR[k] and PHR[$k+18$] ($k=2n+1$). These two branches map to the same IBP set but differ in their PHR tag folding pattern. During testing, PC[15:6] of one branch is kept fixed at 0. In the other branch, it is placed at an address where a single bit is set (and that bit is varied in subsequent experiments). The misprediction rate spikes in a specific case, indicating both branches have the same IBP tag. This confirms that PC[15:6] is folded with the PHR, rather than concatenated to the PHR tag to form the final tag. PHR even bits are also folded with PC[15:6]. By comparing the patterns of how odd and even bits are folded with PC[15:6] respectively, through exhaustive search, it can be inferred that there is also a rotation-shift between the odd and even bits.

The final tag hash function is shown in Figure 9. The other two tables also utilize this same folding algorithm. Thus, for the lower 16 PHR bits that are not used in the IBP index, we find that they are used in the IBP tag but not involved in the 22-bit distance folding with the higher PHR bits. There is a different folding pattern for them as shown in the figure.

Observation 6. PHR is used to calculate the tag for IBP entries, using a different folding distance with the index hash function. For generating the final tag, PC[15:6] is also involved (See Figure 9).

3.2.5 Target Address in IBP

The final step is to recover how the target address is stored in the IBP. To do that, we create a scenario with two indirect branches with identical PC[15:6] and PHR state, plus aliased targets in lower bits. The first branch, i.e. the attacker branch, has a load after each of its targets.

We begin by executing the attacker indirect branch multiple times to ensure that the poisoned target addresses are stored in the related IBP entries. Before the victim indirect branch, we flush its target address from the cache to create a large speculation window. The measurement shows that the victim branch has a near 100% misprediction rate and the target data of the loads are cached, indicating that it speculatively jumps to the attacker branch's target. This result persists even when the number of identical bits in their targets increase to 47. This observation suggests that the IBP stores absolute 48-bit target addresses (because the poisoned branch target matches all 48 bits of the aliased branch). This is in contrast to the BTB which constructs full addresses from 32-bit partial stored addresses.

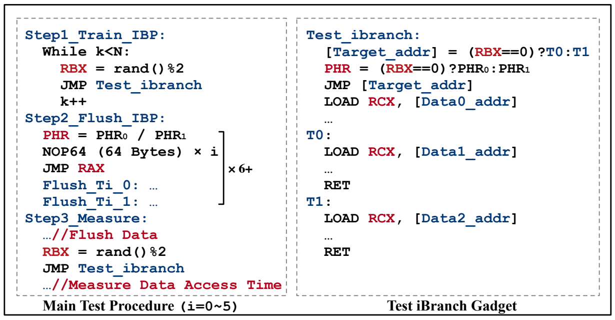
Observation 7. Unlike the BTB, the IBP stores the full 48-bit absolute target address.

3.2.6 Core-ID in IBP

From Section 3.1.2, we already know that the BTB records Core-ID on both *Skylake* and *Raptor Cove* architectures. We conduct a series of similar experiments, employing indirect branches to ascertain the presence or absence of a *Core-ID* field in IBP entries. In the *Skylake* architecture, we find that indirect branches across SMT cores can share the same entry in the IBP, which suggests that there is no *Core-ID* field in IBP entries. In contrast, in the *Raptor Cove* architecture, akin to the BTB, all entries in the IBP are tagged with the SMT core-ID. Our findings indicate that there is not a distinct partitioning between the cores (each can fill a set or be fully evicted from a set). Hence, contention-based side channel attacks (on IBP) remain feasible across SMT cores, whereas direct injection attacks are not viable within the IBP between two SMT cores (except on Skylake). However, all threads that run on the same logical (SMT) core have the same core-ID. As long as the attacker and the victim share the same SMT core, the attacker can still inject malicious targets into the victim's IBP entries and mislead its indirect branches to a disclosure gadget.

3.2.7 Privilege Level in IBP

In order to study the indirect branch prediction in higher privilege levels, e.g. kernel code, we create a custom *system call (syscall)*, which includes a single indirect branch. In the user space, we create an aliased indirect branch with identical PC[15:6] and PHR states as the one in the kernel. In the



```
Step1_Train_IBP:
  While k<N:
    RBX = rand()%2
    JMP Test_ibranch
    k++
Step2_Flush_IBP:
  PHR = PHR0 / PHR1               ┐
  NOP64 (64 Bytes) × i            │
  JMP RAX                         │× 6+
  Flush_Ti_0: …                   │
  Flush_Ti_1: …                   ┘
Step3_Measure:
  …//Flush Data
  RBX = rand()%2
  JMP Test_ibranch
  …//Measure Data Access Time
```

Main Test Procedure (i=0~5)

```
Test_ibranch:
  [Target_addr] = (RBX==0)?T0:T1
  PHR = (RBX==0)?PHR0:PHR1
  JMP [Target_addr]
  LOAD RCX, [Data0_addr]
  …
T0:
  LOAD RCX, [Data1_addr]
  …
  RET
T1:
  LOAD RCX, [Data2_addr]
  …
  RET
```

Test iBranch Gadget

Listing 4: Microbenchmark Pseudo-code for Testing Prediction under BTB Hit + IBP Miss. The test iBranch gadget is called at step 1 and step 3 in the main test.

Skylake architecture, we observe that there is no *Privilege Level* field in IBP entries, enabling the potential for user-level target injection attacks into the kernel. In contrast, in the *Raptor Cove* microarchitecture, we observe that these two indirect branches cannot occupy the same entry in the IBP, even if they have identical PC and PHR values. This suggests the presence of a *Privilege Level* field in IBP entries, which prevents aliasing across different privilege levels. However, there remains a potential attack surface when both the attacker and the victim run on the same SMT core and share the same privilege level.

3.3 IBP and BTB Interactions

As discussed above, both BTB and IBP record information about indirect branches. The question arises: which makes the final prediction for an indirect branch? Intel's patent on indirect branch prediction [44] offers us some hints. As listed in the following, there are four combinations of BTB and IBP outcomes in total. We will disclose the prediction under each scenario.

BTB Hit + IBP Hit: To investigate whether the target prediction originates from the BTB or IBP, we deliberately poison both structures for a specific victim indirect branch and observe side effects for the potential misdirected paths. As expected, our findings indicate that the prediction indeed comes from the IBP for multi-target indirect branches in this case. For single-target indirect branches, the prediction originates from the BTB.

BTB Hit + IBP Miss: We explore this situation with one indirect branch, as shown in Listing 4. After training the IBP to record its targets, we issue more than 6 indirect branches in the same set to evict it from the IBP, creating an IBP miss upon next execution. Finally, we execute the test branch again and measure its misprediction rate. With random binary input, we observe a 50% miss rate instead of 100%, indicating the BPU still makes predictions despite the IBP miss. Moreover, for each misprediction, both *Data1* and *Data2* are cached, but *Data0* is not. Therefore, we infer that the prediction comes from the BTB. Since the BTB entry is only able to hold one target, we observe an overall 50% miss rate when the correct target alternates between T0 and T1 under a random input.

BTB State	IBP State	BPU Prediction	Predicted Target	Speculation?
Hit	Hit	IBP (MT) / BTB (ST)	IBP[47:0] / {PC, BTB}	Y
Hit	Miss	BTB	{PC, BTB}	Y
Miss	Hit	None	Next Instruction	N (Only Fetching)
Miss	Miss	None	Next Instruction	N (Only Fetching)

Table 2: Indirect Branch Prediction Policy under Different BTB/IBP States (MT: Multi-Target indirect branch; ST: Single-Target indirect branch).

BTB Miss + IBP Hit: To test this scenario, step 2 is replaced with 12+ direct branches in the same BTB set to evict the test branch, resulting in a BTB miss upon next execution. The misprediction rate is always 100% in step 3, indicating instructions at some other locations might be speculatively fetched or even executed. The Intel developer manual [1] suggests that the next instruction right after the branch might be speculatively executed if the BTB lacks information about the branch. However, measurement shows that *Data0* is never cached, but the first load instruction is always brought into I-Cache. This indicates that instructions after the branch enter the front-end, but likely are flushed when the decoder recognizes a branch not in the BTB. There is no indication that the IBP is accessed in this case.

BTB Miss + IBP Miss: In this case, the result is the same as above. The execution of the indirect branch will be stalled until the target is resolved. These results confirm that the CPU only identifies the existence of branch instructions based on their BTB states.

3.4 IBP Overall Structure

So far, we have successfully reverse-engineered most of the structural details of the IBP in Intel's recent processors, as well as the prediction mechanism for indirect branches. Figure 10 depicts the ITTAGE structure of the IBP. The indirect branch prediction policy is summarized in Table 2. We believe these analyses will aid the security community in better understanding how and why prior injection attacks work, at the microarchitectural level.

4 Analyzing Intel Hardware Security Measures Against BTI Attacks

Intel has introduced various defense mechanisms aimed at safeguarding processors against malicious exploitation of BPU structures, particularly speculative execution attacks. Although official websites and documentation sources such as [3, 7, 8] outline the usage and applicable scenarios of these defenses, they offer limited insight into their implementation

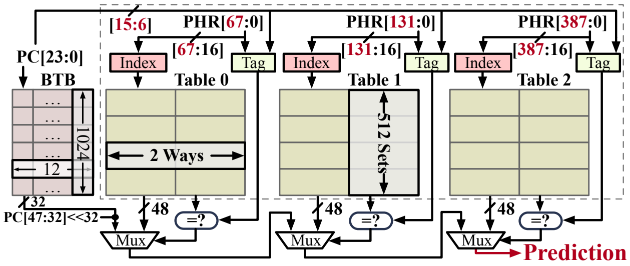


Figure 10: Overall IBP Structure in *Raptor Cove*.

and the specific microarchitectural states affected by them. In this section, our goal is to delve into the inner workings of these defense mechanisms, focusing particularly on Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Predictors (STIBP), and Indirect Branch Predictor Barrier (IBPB). Previous studies [16, 19] have analyzed the overall performance overhead of these defenses. However, none have revealed the specific changes that occur in the BTB and IBP states upon activating these defenses. We examine these defenses in two microarchitectures: *Skylake* (2015) and *Raptor Cove* (2022). Comparing these two architectures can also illustrate the evolution of Intel processors in terms of hardware security, as the former was launched before the Spectre attack, whereas the latter was introduced well after.

To uncover the changes to the BTB and IBP for each defense mechanism, we carefully design microbenchmarks that allow us to track the presence and absence of particular BTB and/or IBP entries across a series of events.

IBRS: We first start with IBRS, which protects against branch injection by lower-privileged attackers. On *Raptor Cove*, our tests show that IBRS is a singular event, in which indirect branch entries are flushed from the BTB. Direct and conditional branches are not flushed. The IBP microbenchmark further shows that the IBP is not cleared and retains all branch targets. Interestingly, then, IBRS has no special function which addresses cross-privilege sharing. This is consistent with Section 3.2.7 that shows that the privilege-level identifier has already prevented cross-privilege IBP injection in the general case, without any special mitigation. Intel seems to describe this as the “Enhanced IBRS mode (eIBRS)” [8], which is an "always on" mitigation.

On the Skylake architecture, our experiments yield similar results for direct and conditional branches, which are not flushed. However, for indirect branches, the outcomes differ significantly. Upon issuing the IBRS, every indirect branch incurs both BTB and IBP misses, even upon repeated execution. This suggests that, on Skylake, the IBRS behaves more like a “mode” than an “event” – once activated, both the BTB and the IBP are entirely disabled, providing no target prediction for indirect branches. After IBRS is disabled, we are able to continue to successfully access prior IBP entries, indicating neither the BTB or IBP are flushed when IBRS is invoked.

Defense μArch		IBRS	STIBP	IBPB
Skylake	BTB	Disable iBranch	None	Flush All Types
	IBP	Disable	Disable	Flush
	Behavior	Mode	Mode	Event
	Granularity	Physical Core	Physical Core	Physical Core
Raptor (Golden) Cove	BTB	Flush iBranch	Flush iBranch	Flush iBranch
	IBP	None	None	Flush
	Behavior	Event	Event	Event
	Granularity	SMT Core	SMT Core	SMT Core

Table 3: Impacts of Intel Hardware Defenses on BTB/IBP.

STIBP: STIBP was introduced with a goal of preventing victim poisoning by attackers on a co-located SMT logical core. On *Raptor Cove*, the results are identical to the IBRS: only BTB entries for indirect branches are flushed, without flushing the IBP. This also aligns with Section 3.2.6 showing that cross-SMT protections are in place by default.

On *Skylake*, STIBP does not flush the BTB entries at all. For indirect branches, we find some unexpected behavior. We can successfully predict single-target indirect branches, but not multiple-target branches. This indicates that the IBP is disabled, but the BTB indirect branch entries are not flushed and the BTB is used to generate indirect branch predictions. Thus, STIBP operates as a special mode of operation (that gets turned on and off) and not a distinct flush event – and in fact the IBP is not flushed.

IBPB: IBPB is intended to prevent attackers from poisoning indirect branches on the same logical core. On *Raptor Cove*, we find that the IBPB flushes all the BTB entries of indirect branches, similar to the IBRS and STIBP. However, IBPB also flushes the IBP entries. On *Skylake*, we find that IBPB flushes the entire BTB and the entire IBP. In summary, IBPB is a stricter and more thorough defense than the other two, and can be applied in a wider range of attack scenarios. However, this comes at the cost of the highest performance overhead (of the *Raptor Cove* mitigations).

Furthermore, we find that *Raptor Cove* allows each of these defenses to be applied per SMT core. When defenses are activated on one SMT context, the branch prediction for the other context remains unchanged. However, on *Skylake*, both SMT contexts are affected. All our discoveries regarding Intel hardware defenses are summarized in Table 3.

5 iBranch Locator

Previous branch target injection (BTI) attacks require extensive efforts to create branch history aliasing between the attacker and the victim branch. For instance, Spectre v2 requires the attacker to execute multiple aliased branches alongside the victim to replicate identical history patterns [31]. BHI uses a few branches to brute-force the history, requiring thousands of attempts to achieve aliasing [14]. Consequently, in this section, building upon our new, much deeper understanding

```
iBranch_index_locator:
  RBX = rand()%2                         ┐
  RAX = (RBX==0)?Ti_0:Ti_1               │
  PHR = (RBX==0)?PHRa_0:PHRa_1           │
  NOP64 (64 Bytes) × i                   │ ×6
  JMP RAX                                │
Ti_0: …                                  │
Ti_1: …                                  ┘
```

iBranch Index Locator (i=0~5)

```
Victim_branch:
  Input = k
  RAX = Target_v_k
  PHR = PHRv_k
  JMP RAX
Target_v_0: …
Target_v_1: …
Target_v_(N-1): …
```

Victim

Listing 5: Pseudo-code of iBranch Index Locator for Extracting the IBP Index of the Victim Indirect Branch. The solid frame means the branches are in separate processes. They are launched simultaneously.

of the IBP structure, we propose iBranch Locator, an efficient and high-resolution tool to locate any indirect branch within the IBP, without the need to know any prior history information before the branch. In this tool, the locating process is divided into two steps: first finding the IBP index and then searching for tag aliasing. Utilizing this tool, two injection attacks can be easily mounted with precision, which will be presented in Section 6.

5.1 iBranch Index Locator

The first step is to find the IBP set where the victim's indirect branch is located. Namely, we need to extract the index of the victim entry. To distinguish which set the victim branch uses, iBranch Index Locator conducts an associativity test during the victim's execution, using an eviction-based contention technique.

As Listing 5 shows, it has 6 indirect branches with the same PHR setup and different PC[15:6] to fully use an IBP set. The victim contains an indirect branch with multiple targets, each correlated to a PHR state, meaning that each target is stored in a different IBP entry. iBranch Index Locator runs on a different SMT core so the real associativity can be measured more accurately without the possibility of entry sharing with the victim; however this can be easily adapted to other sharing scenarios. PHR_{a_0} is fixed to 0 while PHR_{a_1} is changed during each trial. Since the PHR is folded into a 9-bit index, 512 attempts are needed at most. The misprediction rate of the 6 locator branches is measured. If the victim branch uses a specific IBP entry, an increase in the misprediction rate under a certain PHR_{a_1} will be observed. On some older machines like *Skylake*, the IBP is indexed by an 8-bit value, so we only need 256 attempts.

As the example in Figure 11 illustrates, during the 512 trials, the misprediction rate rises at 184 and 384. This result indicates that the victim indirect branch is recorded in two different IBP sets, each corresponding to a target. In this example, `Target_v_0` is stored in the 184th set, and `Target_v_1` is in the 384th set.

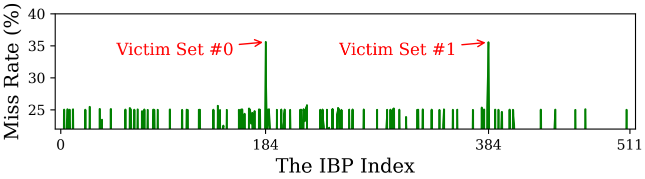


Figure 11: Misprediction Rate of the Indirect Branches in iBranch Index Locator. Spikes corresponds to the victim sets.

5.2 iBranch Tag Locator

iBranch Index Locator does not extract the entire PHR, only the index (the 9-bit value derived from the 388-bit PHR). Locating the victim entry requires not only the index but also the tag, making it more challenging, particularly since we have shown the tag hash function differs from the index hash.

So the second step of iBranch Locator is to search for tag aliasing. We can again do a brute force search for the 11 bits by only controlling the PHR, but it requires a large effort to create index and tag aliasing at the same time. Based on the tag hash function revealed in Section 3.2.4, it can be largely simplified by the fact that 10 of the 11 bits are xored with PC bits to create the tag. Thus, iBranch Tag Locator can permute the relevant PC bits (with 1024 branches) and need only toggle two PHR bits (which create the same index but a different Tag[10]) to complete the search of all 2048 tag permutations. For instance, to toggle Tag[10], PHR[34] can be toggled. To maintain the same IBP index as the victim, $PHR[34+18=52]$ should also be flipped. On *Skylake*, we find the IBP tag is 7 bits, requiring much fewer attempts to successfully find the tag.

Different from iBranch Index Locator, iBranch Tag Locator needs to run on the same SMT core with the victim, because entry sharing will only occur when they share core-ID. Figure 12, shows a related experiment, where we seek to create IBP entry collision between the victim and locator branches. It only succeeds when we permute the PC bits to replicate the tag of the victim. We again see two hits, at two PC values, indicating the victim branch has two targets and occupies two IBP entries.

Therefore, iBranch Locator empowers us to successfully uncover the IBP entry of the victim branch, with at most 512+2048 trials on *Raptor Cove* and 256+128 trials on *Skylake*. It can also disclose other exploitable information about the victim branch, such as the number of targets it may jump to. Moreover, it is immune to address randomization techniques, such as ASLR and KASLR. iBranch Locator is still capable of extracting the location when the victim is running with a randomized branch address and targets. Last but not least, it supports cross-privilege detection starting from user space: iBranch Index Locator can be launched in user space to extract a kernel indirect branch's IBP index, then iBranch Tag Locator might run in an unprivileged eBPF program to

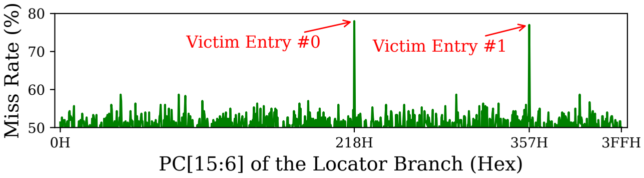


Figure 12: Misprediction Rate of the Locator Branch under Different PC[15:6]. Spikes indicate that tag aliasing is found.

recover the tag.

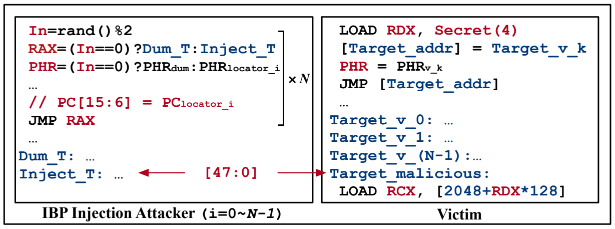
Despite the advantages offered by iBranch Locator, it has limitations. In this form, it cannot distinguish a target indirect branch from other indirect branches in the victim; however, indirect branches are far less common in many executables and identifying a few (or even many) false positives is still far better than prior brute force methods.

6 High-Precision Target Injection Attack

Using iBranch Locator, we introduce two distinct types of high-precision branch target injection attacks directed at indirect branches. The first type of attack poisons the victim's IBP entry. This attack offers a very wide range of attack possibilities due to the full-length target being stored within the IBP. The second type of attack directly injects malicious targets into the victim's BTB entry, which can be done more quickly once iBranch Index Locator has successfully recovered the IBP index, without the need for tag searching. In contrast to previous target injection attacks [14], our attacks do not require any prior history information of the victim. Additionally, it is efficient in identifying aliasing, which is a common challenge in this domain. Moreover, even with ASLR enabled, we propose a new method to break the randomized bits of the victim branch address. This makes injection attacks still feasible in the cross-process scenario.

Attack Model: The attacker and the victim are two processes on the same SMT (logical) core. The victim contains an indirect branch with multiple targets, each correlated to a specific PHR state. We assume that there is a secret-dependent malicious load which is never reached by the victim during normal execution, similar to the gadget exploited by Spectre v1 [31]. The main goal of the attacker is to leverage iBranch Locator to inject malicious targets and mislead the victim to speculatively leak confidential information in microarchitectural states, e.g. cache. The attacker uses methods like Flush+Reload [68] to extract the cache state. Therefore, we assume that the attacker and the victim use a shared memory.

Experimental Setup: All the experiments described in this section are conducted on an Intel i7-13700K performance core (Raptor Cove) with IBRS and STIBP enabled. The operating system used is Ubuntu 20.04.6 LTS, running an unmodified Linux kernel version 5.15.89.



```
 In=rand()%2
 RAX=(In==0)?Dum_T:Inject_T
 PHR=(In==0)?PHRdum:PHRlocator_i
 ...
 // PC[15:6] = PClocator_i
 JMP RAX
 ...
Dum_T: ...
Inject_T: ...
```

```
 LOAD RDX, Secret(4)
 [Target_addr] = Target_v_k
 PHR = PHRv_k
 JMP [Target_addr]
 ...
Target_v_0: ...
Target_v_1: ...
Target_v_(N-1):...
Target_malicious:
 LOAD RCX, [2048+RDX*128]
```

Listing 6: Proof-of-concept Pseudo-code of the IBP Injection Attack. Two processes run on the same SMT core.

6.1 IBP Injection Attack

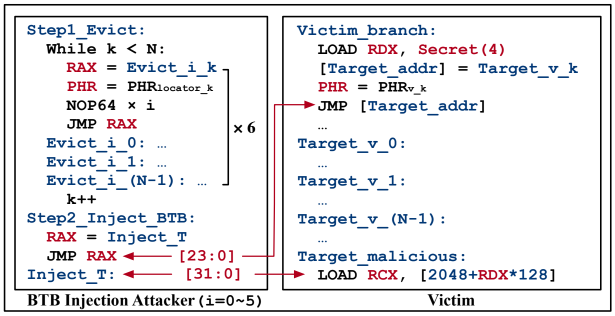
Equipped with iBranch Locator, the attacker is able to locate any victim IBP entry and inject an arbitrary target address. Our attack requires the following steps.

Step 1. Locate victim entries by iBranch Locator: The attacker first launches iBranch Locator to extract both the IBP index and tag of the victim entries, detailed in Section 5. For an N-target victim, iBranch Locator extracts $PHR_{locator\_i}$ as index and $PC_{locator\_i}$ as the corresponding tag $(i = 0, 1, \ldots, N-1)$.

Step 2. Inject IBP with multi-target indirect branch: Listing 6 shows the attacker process, including N indirect branches and 2 targets in total. For each branch, the first target is the Dum_T correlated with a dummy PHR state, which generates a different index from the victim. The second is Inject_T which is identical with the malicious target in the virtual address, correlated with $PHR_{locator\_i}$. The PC[15:6] of the i-th branch is also set to $PC_{locator\_i}$, causing entry sharing with the victim's i-th target. The multi-target branches (even if the first target is only seen once) ensure the attacker uses IBP prediction and records targets into the related IBP entries. As a result, the attacker successfully injects the malicious target into all N victim entries.

Result analysis: We use Flush+Reload [68] to detect load execution and thereby extract the secret value. The result shows shared data at address 2560 has a noticeably higher hit rate than other locations in the Reload stage, indicating the secret is $(2560-2048)/128 = 4$. The injection success rate achieves 97.4% on our *Raptor Cove* machine.

Comparison to Prior Works: In summary, the IBP injection can mislead an indirect branch to any address in the address space. Compared to Spectre v2 [27, 31], our attack bypasses the need for prior knowledge of victim branch history. Compared to Branch History Injection (BHI) [14], our injection, equipped with better knowledge of the IBP implementation, is more precise as it can locate the victim branch's entries accurately at a microarchitectural level. So while we also use brute force methods, we are precisely searching the exact structures rather than the entire possible branch history; thus requiring, in many cases, orders of magnitude fewer trials.



```
Step1_Evict:
  While k < N:
    RAX = Evict_i_k
    PHR = PHRlocator_k
    NOP64 × i
    JMP RAX
  Evict_i_0: …
  Evict_i_1: …
  Evict_i_(N-1): …
    k++
Step2_Inject_BTB:
  RAX = Inject_T
  JMP RAX <---- [23:0]
Inject_T: <---- [31:0]
```

```
Victim_branch:
  LOAD RDX, Secret(4)
  [Target_addr] = Target_v_k
  PHR = PHRv_k
  JMP [Target_addr]
  …
Target_v_0:
  …
Target_v_1:
  …
Target_v_(N-1):
  …
Target_malicious:
  LOAD RCX, [2048+RDX*128]
```

Listing 7: Proof-of-concept Pseudo-code of the BTB Injection Attack. Two processes run on the same SMT core.

6.2 BTB Injection Attack

Next, we introduce a more efficient target injection attack, with only iBranch Index Locator needed. In this attack, the attacker indirect branch can inject malicious targets into the victim's BTB entry and mislead it via BTB prediction. Listing 7 and the following description demonstrate the attack steps.

Step 1. Evict the victim from IBP: First, the attacker uses 6 or more indirect branches to evict the victim branch from the IBP. This can be easily achieved by setting the eviction branches' PHR states to $PHR_{locator}$, as extracted from iBranch Index Locator. This creates a BTB Hit + IBP Miss scenario for the victim, ensuring the BTB will source its next target prediction.

Step 2. Inject BTB with single-target indirect branch: As observed in Section 3.1.1, branches with the same PC[23:0] share the same BTB entry. Further, if they have identical branch types, speculative fetching and execution will occur. Therefore, our attacker leverages this feature to inject and mislead the victim branch. As step 2 in Listing 7 shows, the aliased attacker branch has only one target, aliased (low 32 bits) with the malicious load. Therefore, after IBP eviction, the attacker injects the malicious target into the victim's BTB entry. A single-target indirect branch gets the BTB prediction and records its target in the BTB, so it is used here to ensure the victim's BTB entry gets overwritten each time.

Result analysis: Figure 13 shows the average cache hit rate of all shared memory locations in the Reload stage for each of three scenarios. With both IBP eviction and BTB injection, the secret '4' can be successfully extracted. However, with only IBP eviction, the cached data disappears, indicating the BTB entry has not been injected by the attacker. BTB injection alone also fails, because the victim's prediction comes from the IBP rather than the BTB. The injection success rate is 97.2% on our *Raptor Cove* machine.

Comparison to Prior Works: This BTB injection attack against the IBP is more efficient than previous injection attacks [31], requiring at most 512 trials in iBranch Index Locator. The injection can be carried out instantly after the attacker gets the IBP index of the victim. Moreover, this attack can also be applied in a cross-privilege scenario: iBranch Index Locator and IBP eviction can be launched in user space, and then an unprivileged eBPF program that includes an aliased indirect branch can inject the malicious target into the BTB, similar to the method described in Branch History Injection (BHI) [14].

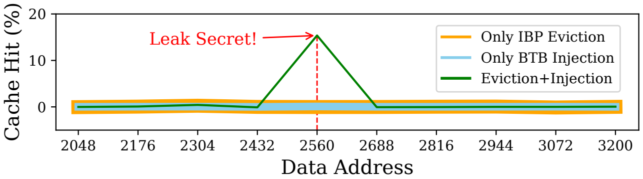


Figure 13: Cache Hit Rate of All the Shared Memory Locations in the Reload Stage.

6.3 Breaking ASLR and Enabling Injection

Most target injection attacks are ineffective under a cross-process scenario with ASLR enabled, as address randomization makes it challenging for attackers to create aliasing with malicious gadgets in the victim process, especially for IBP injection that requires full 48-bit target aliasing with the gadget. Several prior works [18, 23, 76] proposed methods to break ASLR. In this section, we develop a new method to break it inside the BTB, enabling BTB injection in the presence of ASLR, with relatively few attempts.

Step 1. Recover PC[23:12] from BTB associativity: In the first step, we aim to break PC[14:12] of the victim indirect branch. We use a similar idea to iBranch Index Locator, which extracts information from associativity tests. We prepare 12 direct branches with the same BTB index, i.e. PC[14:5], to measure BTB associativity. Because the ASLR only randomizes bits higher than the 12th, we need at most 8 trials to locate which BTB set the victim is in. This successfully uncovers PC[14:12], which is all we need to create index aliasing.

To create the BTB injection attack, we also need to establish tag aliasing. To break the BTB tag (PC[23:15]), we use a test direct branch with the same PC[14:0] as the victim, then brute-force the tag. We measure the misprediction of another 11 eviction direct branches mapping to the same set. Based on Observation 1, when the test direct branch has the same PC[23:0] as the victim, they share a BTB entry and thus the misprediction of the 11 eviction branches is low. Otherwise, the test branch uses a different entry from the victim, causing evictions and thus mispredictions within the 11 branches. In our experiments, we have managed to locate the BTB entry of the victim consistently and recover PC[23:12].

Step 2. Recover PC[31:24] from BTB injection: BTB injection attacks require aliasing in the lower 32 bits of targets. As the offset between the malicious gadget and victim is fixed, the lower 23 bits of the gadget are known. We only need to brute-force bits 31-24, requiring 256 total trials. iBranch Index Locator is immune to ASLR as mentioned before, so IBP eviction remains effective. Successful BTB injection will occur for a certain Target[31:24], breaking partial ASLR (bits 31-12) and enabling effective target injection against indirect branches at the same time.

The results show that we can achieve a 96.5% success rate of breaking ASLR on *Raptor Cove*. To sum up, we need 8+512+256=776 attempts at most to break partial ASLR. Previous works inject targets into the IBP, which stores the entire 48-bit target and this makes injection difficult with ASLR enabled, because breaking all the randomized bits (from the 13th to 48th bit) needs tremendous effort (over 10M brute-forcing attempts). However, with our method and BTB injection, the attack is still practical with ASLR enabled.

7 Mitigation Strategies

In this section, we discuss the potential mitigation strategies for Branch Target Injection (BTI) attacks.

Mitigations from CPU Vendors: As discussed in Section 4, the Indirect Branch Prediction Barrier (IBPB) is the most strict defense available on the newest CPUs and mitigates any out-of-place target injection attack if used properly. To the best of our understanding, Linux opts to automatically activate the IBPB during context switches between different users. The default policy in the latest Linux version, termed "IBPB: conditional," only activates IBPB during transitions to *SECCOMP* mode or tasks with restricted indirect branches in the kernel. Consequently, IBPB activation is infrequent in both user and kernel spaces due to the significant performance overhead (up to 50% [19]). It is not a viable mitigation for frequent domain crossings (browsers, sandboxes, and even kernel/user) – plus the fact that the OS does not use it in the most frequent domain transitions by default.

However, even with this measure in place, when an attacker exploits intra-process poisoning, injection opportunities persist. Therefore, a potential mitigation strategy could involve triggering the IBPB during context switches between all processes – actually in all transitions between different security domains, where it is anticipated that those security domains may influence and/or leak information to another. This could represent a very high overhead for systems with low-level management of short threads, such as microservices or FaaS, if they even had the opportunity to employ IBPB (since they carefully avoid kernel calls between user-level switches). In general, relying on this mitigation would indeed necessitate the operating system triggering frequent IBPBs, resulting in significant overhead.

Secure BPU Design: Section 3 highlights that Intel has already integrated new fields such as *Core-ID* and *Privilege Level* into their recent Indirect Branch Predictor (IBP) design. This inclusion aims to prevent aliasing between indirect branches originating from different SMT cores and different privilege levels, respectively. However, this still leaves potential attack surfaces within the same-core and/or same-privilege scenarios. For future BPU designs, a more complicated tag should be considered to provide more fine-grained isolation across security domains [42,62,77]. Additional measures, such as employing encryption or randomization in entry allocation and index generation policy [25,33,75,77,78], are also suggested.

8 Conclusion

This paper presents the first comprehensive analysis of the Indirect Branch Predictor (IBP) and the Branch Target Buffer (BTB) structures in high-end Intel CPUs. This work also analyzes the inner workings of Intel hardware defenses against Branch Target Injection (BTI) attacks, including IBPB, IBRS, and STIBP, shedding light on their mechanisms and the affected structures upon activation. Expanding on these findings, it demonstrates precise Branch Target Injection (BTI) attacks, showing the continued feasibility of BTI attacks in specific scenarios such as cross-process situations. Leveraging this, it also exploits the IBP and BTB to break Address Space Layout Randomization (ASLR).

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful suggestions.

References

[1] Intel® 64 and ia-32 architectures optimization reference manual volume 1. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[2] Agner Fog. "the microarchitecture of intel, amd and via cpus". http://www.agner.org/optimize/microarchitecture.pdf, 2017.

[3] Intel® single thread indirect branch predictors (stibp). https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html, 2018.

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

[5] Open Source Security Inc. the amd branch (mis)predictor: Just set it and forget it! https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it, 2022.

[6] Retpoline: A branch target injection mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html, 2022.

[7] Intel® indirect branch predictor barrier (ibpb). https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html, 2023.

[8] Intel® indirect branch restricted speculation (ibrs). https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2023.

[9] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Cryptography and Coding*, 2007.

[10] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Computer and Communications Security (CCS)*, 2007.

[11] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *RSA Conference*, 2007.

[12] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[13] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[14] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks. In *USENIX Security Symposium (USENIX Security)*, 2022.

[15] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 2006.

[16] Jonathan Behrens, Adam Belay, and M Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In *European Conference on Computer Systems*, 2022.

[17] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Computer and Communications Security (CCS)*, 2019.

[18] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Asia Conference on Computer and Communications Security*, 2020.

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

[20] Chandler Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[21] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *Cybersecurity Development (SecDev)*, 2017.

[22] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. Branchspec: Information leakage attacks exploiting speculative branch instruction executions. In *International Conference on Computer Design (ICCD)*. IEEE, 2020.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[25] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. Evolution of the samsung exynos cpu microarchitecture. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[27] Jann Horn et al. Reading privileged memory with a side-channel. *Project Zero*, 2018.

[28] Bradley D. Hoyt, Glenn J. Hinton, David B. Papworth, Ashwani K. Gupta, Michael A. Fetterman, Subramanian Natarajan, Sunil Shenoy, and Reynold V. D'Sa. Method and apparatus for implementing a set-associative branch target buffer, U.S. Patent US5574871A, Nov. 1996.

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

[30] Intel® C++ Compiler 19.1 Developer Guide and Reference, 2020.

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

[32] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[33] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[34] 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 (USENIX Security)*, 2017.

[35] Haifeng Li, Tianyue Lu, Yuhang Liu, and Mingyu Chen. Make page coloring more efficient on slice-based three-level cache. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019.

[36] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[38] Nick Mahling. Reverse engineering of Intel's branch prediction. https://www.its.uni-luebeck.de/fileadmin/files/theses/BA_NickMahling_ReverseEngineeringIntelsBranchPrediction.pdf, 2023.

[39] Pierre Michaud. A ppm-like, tag-based branch predictor. *JILP-Championship Branch Prediction*, 2005.

[40] Microsoft. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020.

[41] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Demystifying intel branch predictors. In *Workshop on Duplicating, Deconstructing and Debunking*, 2002.

[42] Steven Myers, Jeffry Gonion, Yannick Sierra, and Thomas Icart. Indirect branch predictor security protection, U.S. Patent 20200192673A1, Jun. 2020.

[43] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[44] Lihu Rappoport, Ronny Ronen, Nicolas Kacevas, and Oded Lempel. Method and system for branch target prediction using path information, U.S. Patent 6 601 161 B2, Jul. 2003.

[45] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead µops: Leaking secrets via intel/amd micro-op caches. In *International Symposium on Computer Architecture (ISCA)*, June 2021.

[46] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An" undo" approach to safe speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[47] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 2019.

[48] Andre Seznec. The o-gehl branch predictor. *JILP-Championship Branch Prediction*, 2004.

[49] André Seznec. A 64-kbytes ittage indirect branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[50] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *JILP-Championship Branch Prediction*, 2006.

[51] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. *arXiv preprint arXiv:1903.10651*, 2019.

[52] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[53] Spectre side channels. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html, 2019.

[54] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. Secsmt: Securing SMT processors against contention-based covert channels. In *USENIX Security Symposium (USENIX Security)*, 2022.

[55] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 2010.

[57] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[58] Vladimir Uzelac. Microbenchmarks and mechanisms for reverse engineering of modern branch predictor units. *A Masters Thesis submitted to the University of Alabama*, 2008.

[59] Vladimir Uzelac and Aleksandar Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[60] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *USENIX Security Symposium (USENIX Security)*, 2018.

[61] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauilgi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. In *Principles of Programming Languages (POPL)*, 2021.

[62] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Brb: Mitigating branch predictor side-channels. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.

[63] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[64] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2. In *USENIX Security*, August 2024.

[65] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[66] You Wu and Xuehai Qian. A case for reversible coherence protocol. *arXiv preprint arXiv:2006.16535*, 2020.

[67] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[68] Yuval Yarom and Katrina Falkner. {FLUSH+RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *USENIX Security Symposium (USENIX Security)*, 2014.

[69] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 770–784, 2024.

[70] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023.

[71] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014.

[72] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[73] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[74] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[75] Tao Zhang, Timothy Lesch, Kenneth Koltermann, and Dmitry Evtyushkin. Stbpu: A reasonably secure branch prediction unit. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022.

[76] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. {BunnyHop}: Exploiting the instruction prefetcher. In *USENIX Security Symposium (USENIX Security)*, 2023.

[77] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *Design Automation Conference (DAC)*, 2021.

[78] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. Hybp: Hybrid isolation-randomization secure branch predictor. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

Appendix

A Path History Register (PHR)

As described in Section 2.1, the Path History Register (PHR) keeps track of past taken branches and gets updated with both branch source and target addresses. The general PHR update process is:

- **Step 1:** The old PHR value is left-shifted by 2 bits, with the lowest bits padded with zeros.
- **Step 2:** The left-shifted PHR is further combined with the latest taken branch's source and target addresses through a certain function (depicted in Figure 14) to form the new PHR value.

Table 4 summarizes basic PHR information for *Skylake*, *Raptor (Golden) Cove* and *Gracemont* microarchitectures. On *Skylake* and *Gracemont*, the IBP uses the past 29 branches' history for prediction while the CBP uses 93 branches [70]. In contrast, *Raptor (Golden) Cove* uses the same PHR length for both IBP and CBP, which is 194 branches. As shown in Figure 3, each update left-shifts the PHR by 2 bits. Therefore, the total bit size of the PHR is the product of the number of branches and 2. The bits of the branch source and target addresses used for update are also summarized in Table 4.

μArch	PHR size (IBP)	PHR size (CBP)	Source Addr. Input	Target Addr. Input
Skylake	29 * 2	93 * 2	[18:3]	[5:0]
Raptor (Golden) Cove	194 * 2	194 * 2	[15:0]	[5:0]
Gracemont	29 * 2	93 * 2	[11:0]	[3:0]

Table 4: PHR size used for IBP and CBP prediction, and branch source and target bits involved in PHR update.

Figure 14 illustrates the detailed PHR update function. First, specific bits of the branch and target addresses are XORed together to form a branch trace. This XOR pattern differs based on microarchitecture. The branch trace is then XORed with the left-shifted PHR to form the new PHR.

These results are consistent with the results shown in [70] and [69], but extended to also include the branch history length used for the indirect branch prediction.

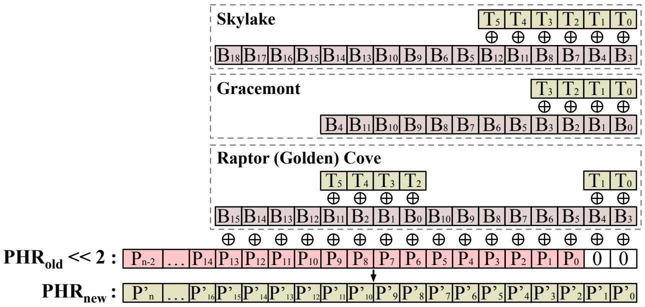

Skylake
T5 T4 T3 T2 T1 T0
B18 B17 B16 B15 B14 B13 B10 B9 B6 B5 B12 B11 B8 B7 B4 B3
Gracemont
T3 T2 T1 T0
B4 B11 B10 B9 B8 B7 B6 B5 B3 B2 B1 B0
Raptor (Golden) Cove
T5 T4 T3 T2
T1 T0
B15 B14 B13 B12 B11 B2 B1 B0 B10 B9 B8 B7 B6 B5 B4 B3
PHR_old << 2 :
Pn-2 … P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0 0 0
PHR_new :
P'n … P'16 P'15 P'14 P'13 P'12 P'11 P'10 P'9 P'8 P'7 P'6 P'5 P'4 P'3 P'2 P'1 P'0

Figure 14: PHR Update Function for *Skylake*, *Raptor (Golden) Cove* and *Gracemont*. B_i represents the i-th bit of the branch source address. T_i is i-th bit of the target address.