Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor

Hosein Yavarzadeh
UC San Diego, USA

Archit Agarwal
UC San Diego, USA

Max Christman
UNC Chapel Hill, USA

Christina Garman
Purdue University, USA

Daniel Genkin
Georgia Tech, USA

Andrew Kwong
UNC Chapel Hill, USA

Daniel Moghimi
Google, USA

Deian Stefan
UC San Diego, USA

Kazem Taram
Purdue University, USA

Dean Tullsen
UC San Diego, USA

Abstract

This paper introduces novel attack primitives that enable adversaries to leak (read) and manipulate (write) the path history register (PHR) and the prediction history tables (PHTs) of the conditional branch predictor in high-performance CPUs. These primitives enable two new classes of attacks: first, it can recover the entire control flow history of a victim program by exploiting read primitives, as demonstrated by a practical secret-image recovery based on capturing the entire control flow of *libjpeg* routines. Second, it can launch extremely high-resolution transient attacks by exploiting write primitives. We demonstrate this with a key recovery attack against AES based on extracting intermediate values.

ACM Reference Format:
Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. 2024. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651382

1 Introduction

Microarchitectural side-channel attacks exploit processor architectural state to leak information from one process or protection domain to another, when there should be no communication between them. Side-channel attacks can be used standalone to leak information, but are also often critical building blocks for more sophisticated attacks. For example, Spectre attacks [18, 21, 35, 75] use side channels both to trigger a controlled misspeculation and to transfer data back to the attacker via transient microarchitectural state. Microarchitectural attacks can exploit various shared units within the processor such as caches [16, 31, 39, 70], branch predictors [23, 26], and address translation buffers [29, 30, 66].

While there have been several control-flow based side channel attacks on the branch predictor [17, 23, 26, 35], they are hampered by limited knowledge of the associated addressing techniques. In particular, attacks targeting the conditional branch predictor (CBP) [26] exclusively target the simplest structure in the predictor, which only enables extracting or injecting coarse control-flow information.

1.1 Our Contributions

In this work, we demonstrate a new class of control flow attacks which exploit detailed knowledge of every aspect of modern CBPs. We show that due to recent research that fully exposed the internal structure of modern Intel branch predictors [71], prior limitations no longer exist, making conditional branch predictors a powerful and dangerous medium for attack. We create primitives that make it as easy (from a programmer's perspective) to read from and write to the tables of the conditional branch predictor, or the path history register (PHR, a precise record of the last taken branches), as it is to read and write memory.

No prior work has used the PHR as an attack vector. We show that the ability to read and write the PHR precisely is a particularly powerful primitive that enables two new attack capabilities.

(1) Full Control-Flow Recovery. Reading the path history register (PHR) provides several advantages over previous side-channel attacks exploiting branch predictors and caches, as it records the complete control flow history of recent branches including branch addresses and precise ordering. We can target each individual execution of a branch

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651382

that is executed many times, where prior CBP attacks could only influence the first few, or capture the bias of the last few instances. In comparison, attacks targeting the instruction cache [10, 11, 15] to recover control flow history can only capture the existence or absence of one outcome and require wide instruction divergence that spans distinct cache lines. Similarly, data cache attacks can only capture control-dependent data accesses that access divergent cache lines. In contrast, we can detect the outcome of every instance of each branch directly, regardless of the cache footprint of the executed code or even whether any memory data was touched.

In this paper, we describe the key primitives that allow us to read and write the PHR, and to read and write the prediction history tables (PHTs) of the CBP. We further describe more advanced techniques. The first of those allows us to go beyond just the recovered PHR and capture control flow history of nearly unlimited length, allowing us to track extremely large swaths of branch history. Previous attacks that can precisely track the entire program's control flow require elevated privilege (i.e., root access), which makes them practical against the specific threat model of untrusted OS (e.g., against Intel SGX [45, 51, 64, 69]), but not valid for other threat models.

Finally, we also describe a tool that transforms the PHR (a series of heavily folded bits of branch address and target history) into a (nearly always) unambiguous control flow graph (CFG) which includes the full history (series of taken/not taken decisions) of every branch. This can be done both for the physical PHR (194 taken branches and all intervening not-taken branches), but also for our extended path history.

This research is distinct from prior work focused on the PHT state. It demonstrates that the PHR is vulnerable to leakage, reveals data unavailable through the PHTs (ordered outcomes of repeated branches, global ordering of all branch outcomes), exposes a far greater set of branching code as potential attack surfaces, and cannot be mitigated (cleared, obfuscated) using techniques proposed for the PHTs.

(2) High-Resolution Spectre Attack. Beyond just capturing a complete control flow history, writing to the PHR gives us the ability to launch extremely high resolution poisoning (e.g., Spectre) attacks. Prior Spectre-style attacks [26, 35] typically only influence the first instance (or the first few, all in the same direction) of a branch at a given address, such as boundary checking 'if' statements. However, as mentioned, with complete control over the PHR and the PHTs, we can now influence an arbitrary instance of a branch executed many times, inducing sophisticated patterns of branch mispredictions, for example, at specific iterations of a 'for' loop. For example, we can cause an iterative encryption or decryption algorithm to complete and return after any number of iterations (both more and less than intended). This allows us, for example, to observe the results of every incremental step in the algorithm, which in many cases also reveals the key.

We demonstrate the implications of these attack primitives with two case studies. We demonstrate a speculative execution attack against AES that returns intermediate values at multiple steps to recover the AES key, which requires both the reading and writing primitives. We also steal secret images by capturing the complete control flow (up to tens of thousands of branches) of *libjpeg* routines. Finally, we demonstrate that our attack primitives, and thus also the attacks built upon them, work across virtually all protection boundaries and in the presence of all recent control-flow mitigations from Intel.

Outline. Section 2 provides background. Section 3 delves into the threat model. Section 4 introduces novel attack primitives that allow adversaries to leak and poison the PHR and PHTs of the conditional branch predictor. Section 5 extends the Read PHR attack to capture control flow history of practically unlimited length. Section 6 introduces *Pathfinder*, a tool that reconstructs the control flow graph of the victim program and identifies the execution path leading to the observed PHR value, thereby unveiling the complete control flow history. Section 7 discusses the attack surface. Section 8 demonstrate the real-world applicability of our attacks by successfully recovering a secret image enabled by capturing the entire control flow of *libjpeg* routines. Section 9 demonstrates successfully key recovery attack against AES in widely used cryptography libraries, like the Intel-IPP [6]. Section 10 discusses potential countermeasures. Section 11 reviews related work and Section 12 concludes the paper.

1.2 Disclosure

We communicated the security findings outlined in the paper to both Intel and AMD in November 2023. Intel has informed other affected hardware/software vendors about the issues. AMD plans to address the concerns raised in the paper through a Security Bulletin, AMD-SB-7016.

2 Background

This section provides relevant information about the branch prediction units (BPUs) inside modern processors. In particular, we focus on conditional branch predictors (CBPs) which provide a prediction as to whether each conditional branch will be taken or not. We summarize recent findings about CBPs [26, 71] and verify that those finding are still valid for more recent Intel processors (Raptor Lake) that were not examined by previous studies.

2.1 Branch Prediction

High-performance processors rely heavily on branch prediction to optimize pipeline utilization. In general, the branch prediction unit (BPU) is responsible for making three critical

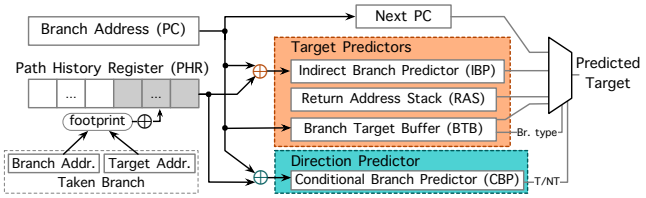


Figure 1. Branch Prediction Unit (BPU) in Modern CPUs.

predictions: identifying branches, predicting their destination addresses, and outcomes (taken or not). In order to make these predictions, the BPU features multiple components. These components consist of the branch target buffer (BTB) and the indirect branch predictor (IBP) for identifying branches and their targets, along with the conditional branch predictor (CBP) for predicting their outcomes. Figure 1 provides an overview of the BPU in modern processors, drawing from previous studies [17, 26, 28, 35, 40, 44, 62, 71].

2.2 Conditional Branch Prediction (CBP)

The conditional branch predictor, also referred to as a directional branch predictor, is responsible for predicting whether the branch instruction will be taken or not. The CBPs discussed in the literature are typically history-based [34, 42, 56, 59, 72], meaning they make predictions about branch outcome by analyzing previous outcomes. These predictors can be further categorized based on the type of history they keep track of, which includes local and global history. Local history records past outcomes specific to the same branch, whereas global history monitors the outcomes of all branches executed by the processor. A simple local predictor uses a table of saturating counters, simply indexed by the branch address [36, 42, 72]. A high saturating counter value predicts a branch to be taken, a low value predicts not taken, and the counter is adjusted upon branch resolution. In contrast, global predictors maintain a history of past branches in a global history register (GHR) and use this history to access branch prediction tables [24, 49, 73].

State-of-the-art branch predictors use a combination of local and global predictors [43, 52–56]. For instance, the **TA**gged **GE**ometric history length predictor (TAGE) uses geometricly increasing history lengths for table look-ups [56]. TAGE has a track record of success in branch predictor championships [3–5] and is found in commercial high-performance processors [33, 71, 77]. TAGE consists of a base predictor, which is indexed by the branch address, and multiple tagged tables that are indexed by hash functions of the branch address and different history lengths. Each entry in the tagged tables includes a saturating counter, a tag value for history association, and a usefulness counter for replacement policy.

BranchScope. Evtyushkin et al. [26] show that in modern Intel processors (Specifically Sandy Bridge, Haswell, and Skylake architectures), the CBP is made of a local predictor which is indexed by the branch address, and a more complicated global predictor that uses global history to make predictions. BranchScope initiates hundreds of thousands of random branches to force the CBP to prioritize the local predictor over the global one. It then generates collisions within the local predictor by aliasing branch addresses, influencing and potentially leaking local predictor entry values. However, BranchScope falls short of providing a comprehensive analysis of the global predictor, and how it could be exploited in side-channel attacks.

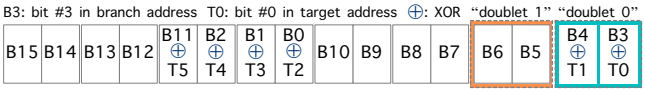


Figure 2. Branch ***footprint*** Used in Updating the PHR.

Half&Half. More recently, Yavarzadeh et al. [71] present a comprehensive reverse engineering analysis of the CBP in Intel processors spanning from Ivy Bridge to Alder Lake. That work sheds light on the intricate workings of the global predictor and proposes a software-based mitigation strategy that effectively partitions CBP structures and prevents leakage between two domains using the shared predictor. As our attacks rely on the inner workings of the CBP, the following subsections provide a more detailed background on the structures used in Intel CBPs. We present a summary of previous findings, but also confirm and extend those results on newer Intel microarchitectures, in particular the latest Intel microarchitecture, *Raptor Lake*.

2.2.1 Path History Register (PHR).

The global history in Intel CBPs, referred to as a path history register (PHR), records the history of the last 194 taken branches in Alder Lake (93 in Skylake), whether they are conditional or unconditional. Branches that are not taken do not affect the PHR. Also, 16 bits from the branch address (bits 15 to 0) and 6 bits from the target address (bits 5 to 0) form a 16 bit number called ***branch footprint*** (depicted in Figure 2), which is used in the PHR's update process. The PHR undergoes a two-step update process upon a branch being taken: first, a leftward shift of two bits, followed by XORing the 16-bit footprint into the PHR. This can be expressed as follows:

$$PHR_{new} = (PHR_{old} << 2bits) \oplus footprint$$

From the PHR update policy, we can conclude that the PHR functions as a $194 * 2$ bits shift-register, with the even and odd bits operating independently as the PHR shifts two bits to the left per taken branch. To simplify the PHR calculations and representation, we introduce a new notation known as the ***doublet***. A *doublet* is a 2-bit value, and within the PHR, it represents pairs of adjacent bits. For instance, in Figure 2, bit 0 and 1 form doublet 0, bit 2 and 3 constitute doublet 1, and so on. With this representation, the PHR functions as a 194-doublet shift register; therefore, when a branch is taken,

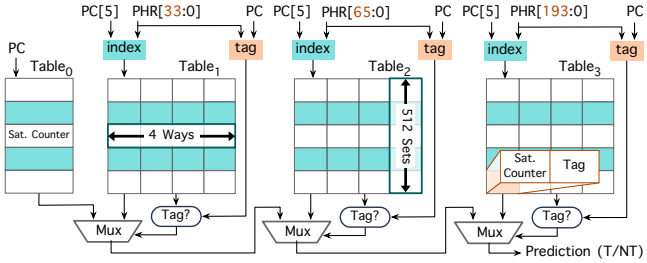


Figure 3. Structure of the CBP in Intel CPUs.

it shifts one doublet to the left, and its 8 lower doublets are XORed with the *footprint*, which also consists of 8 doublets.

Observation 1. The PHR structure in *Raptor Lake* architecture is identical to that of *Alder Lake*.

2.2.2 Pattern History Tables (PHTs). Half&Half reveals the presence of three prediction tables within the CBP, with each table being 4-way set associative indexed by a 9-bit function, composed of eight bits from folded global history (PHR) and a single bit from the PC (PC[5] or PC[4] depending on the architecture). Additionally, they identify a base predictor that works in conjunction with the PHTs and is indexed using the lower 13 bits of the PC. We validate their findings on *Raptor Lake's* CBP, as well.

Figure 3 offers an overview of the CBP in use within modern Intel microarchitectures. Similar to the TAGE predictor, the reconstructed CBP model comprises a base predictor (Table 0) indexed solely by the branch address, serving as a local base predictor alongside three tagged components (Table 1-3) indexed and tagged using the PC and increasing lengths of the PHR: Table 1 uses the 34 lower doublets of the PHR, Table 2 utilizes the 66 lower doublets, and Table 3 leverages the full-length PHR. Each entry in the tagged tables comprises a saturating counter and a tag value for history association that is formed through a combination of PHR and PC.

While previous work [71] did not identify the size of the saturating counters in the PHT entries and the base predictor, we find that information to be important for our attacks. In order to find the bit-width of the saturating counters, we study the misprediction rate of a conditional branch which has a repetitive pattern of directions (e.g. "$T...TN...N$" where the number of 'T's and 'N's are equal to m), and its PHR is fixed to all zeros. We increase m and measure the misprediction rate. The point at which the number of mispredicted branches remains constant after that indicates the bit-width of the saturating counter: $n = \log_2(m+1)$.

Observation 2. We confirm that 3-bit saturating counters are used in the branch predictor [40].

Machine	machine 1	machine 2	machine 3
Model Name	Core i9-13900KS	Core i9-12900	Core i7-6770HQ
µArch.	Raptor Lake	Alder Lake	Skylake

Table 1. Specifications of the Target Processors.

3 Assumptions and Threat Model

The primary focus of this paper is on vulnerabilities arising from the conditional branch prediction unit employed in recent Intel processors. We assume that the victim and the attacker belong to separate security domains but are part of either the same or different processes running on the same physical core – either as co-resident SMT threads, or time-sliced execution flows. In all the attacks that we show in this paper, we assume that the attacker has the ability to invoke the victim code multiple times, e.g., through a system call or a function call. Additionally, we assume that the branches within the victim program behave deterministically with regards to the inputs. In other words, with the same set of inputs the outcome of the victim branches remain constant across different calls. Moreover, we assume the attacker has access to the binary (the addresses of the branches) of the victim function. While this paper specifically explores a subset of processors outlined in Table 1, the generalizability of the reverse engineering efforts [26, 71] extends our attack methodology to a broader range of Intel microarchitectures, encompassing all of Intel's flagship processors since Ivy Bridge (2012), spanning over a decade.

4 Attack Primitives

We conceptualize the branch predictor as a read/write scratchpad. A read operation occurs when a branch instruction receives a prediction from the branch predictor. On the other hand, when a branch instruction resolves in the pipeline, typically during the execution stage, it updates the PHTs and/or PHR, resembling a write operation. As discussed in Section 2, the PHR, along with the branch address, is used to index the PHTs for reads (branch predictions) and writes (branch outcomes). Not only do these tables persist across protection boundaries, but so does the PHR itself. By reading the PHR or PHT entry values following a victim program, one can perform a leakage attack. Additionally, writing the PHTs and/or PHR values before calling the victim can enable new Spectre attacks.

However, due to the intricate nature of these structures and their indexing functions, performing reads and writes is significantly more complex than standard memory accesses, making them more challenging. To address this, we build our attacks from a few carefully-crafted primitives that greatly simplify the process of exploiting the PHR, PHT, or both. This section introduces these attack primitives. We introduce

fundamental techniques for manipulating the PHR and then build more complex attack primitives on top of them.

Shift PHR. During the PHR update, the first step involves shifting it one doublet to the left. In the second step, the lower 8 doublets of the PHR are XORed with the footprint, and if the footprint is all zeros, no change occurs. Therefore, the PHR undergoes only a leftward shift of one doublet. A zero footprint is achieved by constructing a branch where the 16 lower bits of the branch address are zero, as are the 6 lower bits of the target address. Using this, we develop a macro called `Shift_PHR[i]` which has `'i'` taken branches with zero footprint, resulting in the PHR shifted by `'i'` doublets to the left. This can be expressed as follows:

$$PHR_{new} = PHR_{old} << i$$

Clear PHR. We know that the PHR has a limited size of 194 doublets, and shifting the PHR value 194 times will result in resetting it to an all-zero state. This operation is represented by `Clear_PHR`, effectively equivalent to `Shift_PHR[194]`.

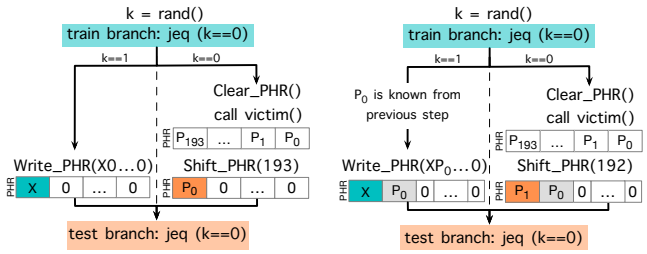
4.1 Write PHR

Given our knowledge of how the branch footprint is generated from the branch and target address bits, we can create a more sophisticated mechanism that allows the attacker to set the PHR to any desired value. Consider a scenario where we initially reset the PHR value using the `Clear_PHR` macro. We introduce a taken branch instruction with zeroed branch and target addresses, except for T0 and T1 (bits 0 and 1 in the target address). By controlling these bits, we can adjust doublet 0 to any desired value, encompassing all four possible options, as it's a 2-bit value. Using this, we develop the `Write_PHR`($P_{193}, P_{192}, P_{191}, ..., P_0$) macro, which takes the desired PHR value as input and, upon execution, sets the PHR value to the specified input value using 194 branches, similar to the one introduced earlier. Each branch assigns the corresponding P_i value in the PHR, where `'i'` ranges from 193 down to 0.

4.2 Read PHR

Reading the PHR can be extremely powerful, because it reveals not just branch outcomes, but also the branch locations and the path that led to each branch. We devise a mechanism for reading the PHR value of a victim program one doublet at a time. In this design, we make an assumption that a victim function with an identical set of inputs that are fixed, but not explicitly known, will exhibit the same behavior (i.e., the same control flow execution) and yield the same consistent expected output.

The key idea here is to create two distinct paths that lead to a *test branch*. One path involves calling the victim function, which sets PHR_0, while the other path directly sets PHR_1. By comparing PHR_0 and PHR_1, we can identify the point at which they become equal, effectively revealing the victim function's PHR value.



(a) Read doublet 0: No correlation captured iff $X == P_0$.

(b) Read doublet 1 once P_0 is known through Figure 4a.

Figure 4. Reading PHR doublets.

Read doublet 0. To read doublet 0 in the PHR, we start with two correlated branches within a for loop. The first branch, called the *train branch*, is conditionally based on a random bit, making its direction unpredictable. The second branch, known as the *test branch*, can be predicted if the branch predictor captures its correlation with the train branch. This prediction is possible due to the two distinct PHR states (or two distinct paths) created by the train branch's behavior, one when it's taken and the other when it's not.

Consider the code snippet in Figure 4a as an example. In this scenario, we want to determine the value of P_0, which represents the least significant doublet in the PHR generated by the `victim()` function call. The attacker begins by executing the *train branch*, based on a randomly generated value. When the *train branch* is taken (k==0), the attacker calls the victim function following the *Clear_PHR* macro, assuming that the PHR value after the `victim()` call would be equivalent to $[P_{193}, P_{192}, P_{191}, ..., P_0]$. Subsequently, the attacker shifts the received PHR value 193 doublets to the left using the *Shift_PHR* macro. This operation results in a PHR with all zeros except for the most significant doublet, which corresponds to the value we aim to recover (P_0). If the branch is not-taken (k==1), the attacker does not invoke the victim function. Instead, we set the PHR to a known value using the *Write_PHR* macro, where all doublets are zeros except for the most significant one, systematically testing all four possible values (00, 01, 10, and 11). If the CBP fails to effectively capture the correlation between the train and test branches, resulting in a 50% misprediction rate for both branches, it indicates that the PHR received from the taken and non-taken paths are identical, i.e., $X == P_0$.

In summary, after testing all four possible X values (00, 01, 10, and 11), we find that in three cases, the misprediction rate (specific to the test branch) is close to 0%, indicating X is not equal to P_0. However, in one specific case, the 50% misprediction rate strongly suggests that X is indeed equal to P_0. Therfore, we successfully leak the value of doublet 0.

Read Doublet 1, and the rest. Once we've obtained the least significant doublet of the PHR (P_0), we can extend the same technique to capture P_1 (as shown in Figure 4b). The process remains similar: in the taken path, we shift the PHR value after the `victim()` function call by 192 doublets, and in the not-taken path, we set the PHR value to $[X, P_0, 0, \ldots, 0]$. With P_0 known from the previous step, we test all four possible options for X to determine P_1. By repeating this process, we can recover all doublets of the PHR following the execution of any victim program.

Evaluation. To evaluate the primitive's effectiveness, we initialized the PHR value to a predetermined state and read it back, verifying that the retrieved value matched the initialized value. We repeated this process with 1000 randomly generated PHR values, and the *Read_PHR* macro successfully retrieved the intended PHR values in all cases.

> **Attack Primitive 1.** *Read_PHR()* retrieves the precise PHR value, capturing the history of the last 194 taken branches, following the execution of a victim program.

4.3 Write PHT

Given our ability to set the PHR to any desired value, we can effectively access any entry in the PHTs or the base predictor. As illustrated in Figure 3, each PHT is indexed by a particular length of the PHR. This allows us to selectively modify a specific entry in any of the three PHTs or the base predictor. To manipulate the 3-bit saturating counter for a specific entry, we execute a branch instruction with the precise values of the PHR and PC eight times in a loop to ensure that the counter becomes saturated, either to *Taken* or *Not-Taken*, depending on our desired outcome.

> **Attack Primitive 2.** *Write_PHT(PC, PHR, value)* sets the PHT entry accessed by *(PC, PHR)* to the provided *value* (taken or not).

4.4 Read PHT

To read the saturating counter value of a particular PHT entry, we use a 'prime+test+probe' mechanism, wherein during the *prime* phase, the counter value is set to all zeros (indicating a prediction of 'Not-Taken'), by setting the PHR and PC appropriately and executing not-taken branches. Subsequently, in the *test* phase, the victim code is executed, leading to updates in the PHTs or the base predictor. In the *probe* phase, we execute a taken branch with precisely the same PHR and low PC bits multiple times while measuring how many times it is mispredicted by the CBP (by measuring the performance counters or the timing difference). This misprediction count reveals information about the value in the PHT entry. For instance, 4 mispredictions indicates the entry remained in the strongly not-taken state, 2 mispredictions indicates it moved two steps away, perhaps due to two taken branch instances.

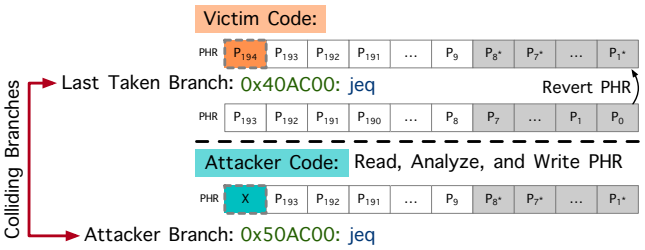


Figure 5. Read doublet 194 of the PHR: The attacker branch misprediction rate is 50% iff $X == P_{194}$.

> **Attack Primitive 3.** *Read_PHT(PC, PHR)* extracts the counter value of the PHT entry indexed by *(PC, PHR)*.

5 Extended Read PHR

One limitation of the *Read PHR* attack is that it only captures the history of the most recent 194 taken branches. While this provides a substantial amount of information, it may fall short for targeted code with more than 194 taken branches. This limitation is particularly significant in older Intel processors (e.g. Skylake) with a smaller PHR size of 93. To address this challenge, we developed a primitive called *Extended Read PHR*, which is not constrained by the PHR size and captures the entire control flow history of a victim program. This primitive leverages the fact that branches within the victim code use the (constantly changing) PHR to access the PHTs, which contain information about far more than the last 194 taken branches. In particular, we exploit the fact that if we know the last 194 taken branches, but we don't know the last doublet (depends on the 195th branch) that was shifted out of the PHR, we can exploit the fact that that doublet, plus 193 known branches, was used to predict the most recent (also known outcome) branch – and recover the unknown doublet.

Read doublet 194. Consider Figure 5, which provides a code snippet of both the victim and the attacker code. For instance, let's focus on the last taken branch within the victim code. After applying *Read PHR*, we obtain the observed PHR value as $[P_{193}, P_{192}, \ldots, P_2, P_1, P_0]$. By leveraging our knowledge of the PHR update policy and the branch and target addresses, we can effectively reverse the PHR update steps to determine the PHR value prior to this branch instruction. We can successfully restore all doublet values except for the most significant doublet. Now, let's consider a not-taken branch on the attacker side with precisely the same lower bits of the branch address. By setting the PHR value before this branch to the value we obtained through reversing the PHR update steps, while brute-forcing the four possible values of the most significant doublet, we create a scenario where

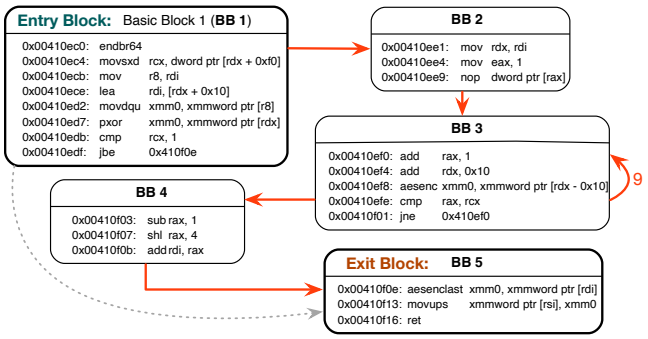


Figure 6. PHR Analysis Output for Looped Implementation of AES ECB Encryption Using AES-NI (Listing 1).

the victim branch and the attacker branch alias with each other in the PHTs. Since their branch addresses are identical, we can infer that they will collide in PHTs only if their PHRs match. This collision or non-collision is reflected in the misprediction rate for the attacker branch. A matching PHR leads to a 50% misprediction rate, while different PHRs result in a 0% misprediction rate. Thus, we can deduce the value of the most significant doublet of the PHR (P_{194}), effectively extending the *Read PHR* to retrieve one more doublet.

We further improve our method by incrementally recovering one additional doublet at a time. This entails orchestrating collisions between the conditional branches in the victim and attacker code. Essentially, we leverage the pattern history tables (PHTs) as a side channel to extract more PHR doublets.

Evaluation. In an extensive series of tests encompassing 1000 cases with varying numbers of taken branches (ranging from 194 to 1000), our experiments consistently demonstrated that the *Extended_Read_PHR* primitive successfully reads the entire control flow history. This holds true unless there are more than 194 consecutive unconditional taken branches, an occurrence we observed to be exceptionally rare in real-world programs. Consecutive unconditional taken branches pose a challenge since they do not interact with PHTs at all, preventing the attacker from exploiting the PHTs as a side-channel to infer the PHR values.

Attack Primitive 4. *Extended_Read_PHR()* retrieves the PHR value after the execution of a victim program. It is capable of capturing the entire control flow history without being constrained by the size of the PHR.

6 Pathfinder

Capturing the PHR is distinct from capturing the runtime control flow of a target program, as the PHR is a complex combination of multiple addresses and target bits for each taken branch. Consequently, we introduce the *Pathfinder* tool, which, given a PHR outcome and an executable code

```
1  void looped(uint8_t *plaintext, uint8_t *ciphertext, AES_KEY *key)
2  {
3      __m128i state = _mm_loadu_si128((__m128i *)plaintext);
4      __m128i *rd_key = (__m128i *)key->rd_key;
5      state = _mm_xor_si128(state, *(rd_key++));
6      for (size_t i = 1; i < key->rounds; i++) {
7          state = _mm_aesenc_si128(state, *(rd_key++));
8      }
9      state = _mm_aesenclast_si128(state, *rd_key);
10     _mm_storeu_si128((__m128i *)ciphertext, state);
11 }
```

Listing 1. Pseudo Code for Looped Implementation of AES ECB Encryption Using AES-NI.

(containing the victim function), constructs the runtime control flow graph of the victim function. Pathfinder uses the *angr* binary analysis tool [58] and an algorithm to identify all potential control flow paths matching the observed PHR values. While it's not guaranteed that there will always be a single path leading to the specific PHR, our extensive analysis has shown that ambiguous results are exceedingly rare due to the PHR's size and complex update function. It should be noted that binary analysis tools, such as *angr* [58], have certain limitations in reconstructing control-flow graphs. For instance, they may miss edges for indirect branches (with multiple targets) in complex binaries. However, in practice, we have found this limitation to have little impact on the *Pathfinder* tool.

Pathfinder Algorithm. After constructing the control flow graph (CFG) from the victim binary, *Pathfinder* takes this CFG and the observed PHR value as inputs and provides a list of all possible paths that could yield the same PHR as output. This algorithm starts from the exit block of the victim's CFG and systematically explores predecessor nodes in a recursive manner, focusing specifically on those capable of fulfilling the conditions for the lowest doublet of the PHR. Given that this doublet is exclusively updated by the last taken branch, we can eliminate predecessors that do not match that doublet. The algorithm then continues recursively, thoroughly evaluating each viable candidate among the predecessor nodes. If, at any point, it encounters a node where no predecessors meet the PHR criteria and analysis is still incomplete, it discards that option and proceeds to the next candidate. This process persists until it identifies all the potential paths matching the observed PHR, with the majority of cases yielding only a single path.

For example, consider the code snippet presented in Listing 1, which illustrates a looped implementation of AES encryption using AES-NI instructions. Let's assume we run this code with a 128-bit key length, which should involve 10 encryption rounds. Afterward, we read the PHR and provide both the PHR value and the binary to the *Pathfinder Tool*. Figure 6 illustrates the tool's output, displaying the runtime control flow graph (CFG), with red edges indicating the executed paths. As illustrated in the graph, the execution starts

at basic block 1 (BB 1), proceeds to BB 2, and subsequently to BB 3, where it iterates nine times. Then, it advances to BB 4 before reaching the exit point at BB 5.

This algorithm not only identifies the path that generates the observed PHR but also provides information about the victim's execution, including (1) the branches taken or not within the victim's code, (2) the number of iterations within each loop, and (3) the PHR values at each basic block.

Evaluation. We evaluate the accuracy of Pathfinder by (1) rigorously testing well-designed microbenchmarks, including challenging scenarios such as varying loop iterations, nested loops, and complex control flow graphs, and (2) validating real-world applications like AES encryption (depicted in Figure 6). In all cases, Pathfinder accurately identifies the precise path leading to the observed PHR value. While most cases exhibit a single path generating the observed PHR, intentionally crafted microbenchmarks may feature multiple paths. However, the tool identifies all possible paths, typically differing in just one CFG node, which allows the attacker to infer the majority of the victim's control flow. A limitation arises with loops exceeding 194 iterations (with invariant control flow between loop branches), where the tool output indicates more than 194 iterations but cannot specify the exact count. Nevertheless, this limitation does not impede the tool from inferring the PHR at the loop entrance and continuing to capture all other control flow in the code. We also expect the tool to begin to break down if the target program reaches far enough back in the branch history to cause evictions in the set associative PHT tables. However, due to the size of the tables and the entropy of the PHR we have not actually seen this in practice, even in our large case study in Section 8 that recovers tens of thousands of branches.

7 Attack Surface Analysis

In this section, we evaluate the effectiveness of our attack primitives in breaching various security isolation boundaries, including those between userspace and kernel code, concurrent threads on simultaneous multithreaded (SMT) cores, and SGX enclaves. Additionally, we examine the effect of Intel's IBPB/IBRS countermeasures on our attack primitives.

7.1 Kernel vs. Userspace

To test if the PHR is affected during the kernel-to-user transition, we perform a read operation on the PHR value immediately after initiating a syscall. We analyze various *syscalls*, including *getppid*, *geteuid*, and our own customized syscalls, and conduct our experiments on kernel version 6.3.0-generic. Our experiments demonstrate that the syscall entrance and exit introduce approximately 23 and 7 branch outcomes into the PHR, respectively. As a result, we can capture over 160 unique branch histories related to those specific system calls using the *Read PHR* macro. In the reverse scenario, where the kernel is entered, we also confirm that the PHR is not flushed, allowing the user program to set a specific PHR value upon entry that will impact kernel predictions. To further evaluate the effectiveness of the *Read/Write PHT* macros, we designed scenarios where user programs attempt to leak and/or poison PHT entries used by conditional branches within system calls. Our experiments confirm that the *Read/Write PHT* macros function as intended, and there is no flushing and/or partitioning of CBP entries across the user/kernel boundary.

	User/Kernel		SGX Enclave			Intel Defenses	
	Enter	Exit	Enter	Exit	SMT	IBPB	IBRS
Read PHR	✓	✓	✓	✓	✗	✓	✓
Write PHR	✓	✓	✓	✓	✗	✓	✓
Read PHT	✓	✓	✓	✓	✓	✓	✓
Write PHT	✓	✓	✓	✓	✓	✓	✓

Table 2. Attack Primitives Practicality.

7.2 Intel SGX

Intel Software Guard Extensions (SGX) [9] is a hardware-based security extension that creates isolated, secure enclaves for running applications, protecting sensitive data and code from unauthorized access or modification. This technology is widely used to secure data and applications in various contexts, offering a secure execution environment on Intel processors. We investigate the behavior of the PHR and PHT entries during SGX enclave enter and exit operations. We conducted experiments similar to those discussed in the preceding subsection to assess the functionality of Read/Write PHR/PHT macros during enclave enter and exit. The results are summarized in Table 2. We confirm that all of our attack primitives work consistently across SGX enclave boundaries.

7.3 Simultaneous Multithreading (SMT)

Nearly all modern high-performance CPUs implement simultaneous multithreading (SMT) [61], which shares the branch prediction unit, including the CBP. We conduct an investigation to assess the effectiveness of our Read/Write PHR/PHT macros in SMT mode, specifically to determine if one co-resident thread can potentially leak or influence the PHR and/or PHT in the other co-resident thread. Our findings show that the PHR (as expected) is not shared between two SMT threads and each logical SMT core has its own dedicated PHR. On the other hand, we found that the PHTs are indeed shared and one malicious co-resident thread can potentially leak or influence the branch direction prediction in the other co-resident thread.

7.4 Intel IBPB/IBRS

The Indirect Branch Predictor (IBP) predicts indirect branch targets using both branch address and the PHR [35, 71]. Previous studies have demonstrated that attackers exploit the

```
GLOBAL(void) _jpeg_idct(...)
{
  // Pass 1: process columns
  for (ctr = DCTSIZE; ctr > 0; ctr--) {
    if (colptr[1] == 0 && colptr[2] == 0 && colptr[3] == 0 &&
    ↪  colptr[4] == 0 && colptr[5] == 0 && colptr[6] == 0 &&
    ↪  colptr[7] == 0) {
      /* Simple Computation */
    } else {
    /* Complex Computation */
    }
  }
  // Pass 2: process rows
  for (ctr = 0; ctr < DCTSIZE; ctr++) {
    if (rowptr[1] == 0 && rowptr[2] == 0 && rowptr[3] == 0 &&
    ↪  rowptr[4] == 0 && rowptr[5] == 0 && rowptr[6] == 0 &&
    ↪  rowptr[7] == 0) {
      /* Simple Computation */
    } else {
    /* Complex Computation */
    }
  }
}
```

Listing 2. Overall Structure of the IDCT Function in *libjpeg*.

IBP [17, 20, 35, 76] in side channel attacks to gain control over the execution flow. In response to these vulnerabilities, Intel introduced two defensive measures: the Indirect Branch Predictor Barrier (IBPB) [7] and the Indirect Branch Restricted Speculation (IBRS) [8]. IBPB establishes a barrier to prevent pre-barrier software from affecting post-barrier indirect branch predictions and the IBRS restricts speculative execution of indirect branches. Our findings (given in Table 2) demonstrate that activating either or both of these mitigations does not flush or partition the PHT or the PHR, and consequently does not mitigate any of our attacks.

8 Image Recovery Attack: *libjpeg*

JPEG is a widely used standard for lossy image compression. The *libjpeg* library [1], and its derivatives [2], are used widely for JPEG image encoding/decoding.

The JPEG encoding process begins by breaking down an original bitmap image into 8∗8 pixels. It employs a lossy compression technique based on the discrete cosine transform (DCT), which transforms the image from the spatial domain to the frequency domain. Following this, each element of the DCT output undergoes quantization, which is the only lossy step in the process, where high-frequency coefficients, less impactful on the overall image, are compressed into smaller values, often rounding them to zeros. The quantized coefficients are then organized and losslessly packed into the output bitstream. The decoder reverses the encoding process, beginning with dequantization and then performing an inverse cosine discrete transform (IDCT) on each coefficient matrix to generate $8 * 8$ bitmap blocks. Combining these blocks regenerates the complete bitmap image.

The *libjpeg* software offers multiple IDCT implementations, all of which follow a shared structure outlined in Listing 2. This structure has two `'for'` loops that iterate over the 8 columns and 8 rows in the coefficient matrix, reflecting the two-dimensional nature of the IDCT operation. The standard processing of a row/column in IDCT involves complex computations. Yet, if all elements in a row/column are zeroes except the first one (called a constant row/column), the computation becomes much simpler. The IDCT implementations in *libjpeg* take advantage of this optimization by checking for this condition and performing simplified computations when applicable.

From a security standpoint, this performance optimization has substantial implications. Gaining insight into the runtime control flow of the IDCT function would unveil the constancy of specific rows/columns, ultimately leading to the exposure of information about the original image.

Attack Scenario. We use the *Unlimited Read PHR* attack primitive in conjunction with Pathfinder to uncover the precise runtime control flow of the image decompression process, particularly within the IDCT function. This enables us to identify which rows/columns are constant within each block. Additionally, we know that the number of constant rows/columns in a block corresponds to the block's relative complexity. We use this insight to recover the original image, as we assign each 8∗8 block a value based on the normalized number of constant rows/columns within it. Consequently, the recovered image frequently exhibits a high similarity to the results of edge detection, particularly in blocks positioned along the image's edges, which tend to be more complex than others.

Evaluation. We conducted an evaluation using a test set of 15 JPEG images. To create a diverse collection, we included a range of images, including high-resolution photographs, simpler logo-style images, QR codes, captchas, and more. Figure 7 presents three examples of successful image recovery using our attack method. The first example demonstrates the recovery of a QR code from the ASPLOS website, which remains scannable despite some noise. The second and third examples showcase the recovery of the ASPLOS logo and website background image, respectively. The number of recovered branches roughly ranges from 1000 for simple logo-style images to 20k for high-resolution images.

Comparison to Prior Works. Prior attacks on *libjpeg* primarily exploited the page fault channel [69] to track the number of constant rows/columns by distinguishing the number of page faults, while cache side channel attacks [32] were employed to monitor cache line access during array look-ups for image recovery. However, these methods have inherent limitations. Those attacks require elevated privilege (i.e., root access), which makes them practical against the specific threat model of Intel SGX, but not valid for other threat models. Additionally, tight loops, small array indexing, and data accesses within code page or page boundaries escaped detection, as the page fault channel's granularity

Figure 7. Examples of Recovered Images by *Pathfinder*.

is limited to the 4KB page size, constraining the temporal and spatial resolution of such attacks. Furthermore, distinguishing between adjacent array indices that fall within a single cache line remains challenging, and numerous mitigations have been introduced against cache side-channel attacks [22, 47, 48, 50, 67].

Alternatively, an attacker can use the instruction cache channel to determine which instructions were fetched, thereby inferring whether a branch instruction was taken or not. However, granularity of such an attack is still limited to the cache-line size (64 bytes). In addition, such an attack still requires privilege elevation and the ability to constantly interrupt (single-stepping), or to probe the instruction cache from a co-located SMT thread; otherwise it can only determine the outcome of a single execution instance of each branch (best case). In contrast, Pathfinder can capture the entire control-flow of the victim, even in scenarios where branch instructions are repeatedly executed in a tight loop, without relying on elevated privileges or SMT co-residency, making it applicable to a significantly broader range of scenarios. Furthermore, in the specific case of the *libjpeg* attack, Pathfinder not only quantifies the number of constant rows/columns but also precisely identifies which rows and columns were constant. This capability allows for the extraction of additional features from a secret image, such as new insights into the frequency domain.

9 Leaking AES Keys

In this section we develop a new Spectre-style attack on the widely used AES algorithm to extract *encryption keys*. We show that reduced-round ciphertexts can be captured via speculative execution channels, and used in conjunction with the full-round ciphertext to recover the AES key, even from constant-time implementations that leverage AES-NI hardware extension. Before diving into the details of our attack scenario, we start with a brief refresher on AES.

```
function encryption_oracle()
{
    plaintext   = random_bytes();
    ciphertext  = encrypt(plaintext);
    encodedtext = base64_encode(ciphertext);
    sidechannel_send(encodedtext);
    return encodedtext;
}
```

Listing 3. An encryption oracle based on AES. The attack leverages this oracle to obtain the encoded ciphertext and leak some information about it via a side channel.

9.1 AES Background

AES operates on fixed-size blocks of data, typically 128 bits, and employs several rounds of operations to transform plaintext into ciphertext. AES offers varying key lengths, including 128, 192, and 256 bits, each corresponding to 10, 12, or 14 rounds of operations, respectively. AES follows four main operations in each regular round: SubBytes, ShiftRows, MixColumns, and AddRoundKey, with the final round differing by excluding the MixColumns operation. SubBytes replaces bytes using a lookup table, ShiftRows rotates rows, MixColumns linearly transforms columns, and AddRoundKey XORs with a round key. AES-NI (Advanced Encryption Standard New Instructions) is an x86 instruction set extension designed to accelerate AES encryption and decryption for enhanced speed and security on Intel processors.

9.2 Attack Overview

The attacker repeatedly queries an AES encryption oracle holding a secret key, which the attacker seeks to recover (see Listing 3). The oracle outputs the ciphertext and, during its post-processing, unintentionally exposes side-channel information about it, such as whether a specific byte is zero or not. This scenario often arises when encoding encrypted data, e.g., during the transfer of encrypted data in JSON text format or the transmission of images [19, 41]. For example, an oracle encrypts the data and then applies post-processing text encoding, such as `base64_encode(aes_encrypt(data))`, before outputting the encoded data [27, 46]. The above attack scenario also applies to decryption where a random ciphertext is processed by the oracle and the attacker can observe the plaintext. Our encryption oracle is the looped implementation of AES encryption using AES-NI within the Intel-IPP library [6], found in Listing 1. Intel-IPP offers an assembly implementation that uses unrolled AES when the plaintext size is less than 64 bytes, employing the looped version otherwise.

We poison the PHTs to induce the victim to speculatively exit the loop early (e.g., in the 2^{nd} iteration instead of 9^{th} iteration when a 128-bit key is being used), returning the reduced-round ciphertext to the attacker through the side-channel. At some later stage in the pipeline, when the processor finally executes the branch instruction within the

pipeline and recognizes the prior prediction as incorrect, it squashes the instructions that were speculatively executed, including those initiated by potential attackers. During this speculative window, however, the microarchitectural artifact of the ciphertext can be leaked through various covert channels like the data cache [31, 60, 70]. In the following, we provide a comprehensive breakdown of the attack steps.

(Mis)Training the Branch Predictor. Our objective is to induce a misprediction in the i^{th} iteration of the '`for`' loop in Listing 1 (line 5). For this, we manipulate specific entries of the PHTs that will be looked up when the processor predicts the victim branch direction using the CBP. This CBP lookup operation occurs based on the <PHR, PC> combination. Utilizing the *(Extended) Read PHR* attack primitive, we retrieve the PHR value after the victim function and using the *Pathfinder Tool*, we determine the exact PHR value in the 'i-th' iteration. Subsequently, we employ the *Write PHT* attack primitive with the observed PHR value to modify the specific entry in the PHTs corresponding to the <PHR, PC> combination, marking it as 'not-taken.' Consequently, when the processor looks up the PHT, it will retrieve a 'not-taken' prediction and exit the loop early in a speculative manner.

To extend the speculation window, we flush the variable that stores the number of encryption rounds (which is 10 for 128-bit key) from the cache, which results in a delay of a couple hundred cycles until the correct number of rounds is retrieved from memory and the mispredicted branch is resolved in the pipeline. By the time the branch mispredict is detected and resolved, the reduced-round ciphertext has already been leaked through the side-channel gadget.

Recovering the Ciphertext. We employ the Flush+Reload technique [70], which allows us to determine through a cache side-channel whether a specific memory address has been accessed. Our approach depends on how the ciphertext has been leaked through the side channel. We can either rely on the gradual leakage of the reduced-round ciphertext, one byte at a time, akin to previous work [57, 63] that employs similar assumptions regarding the encoding of transient ciphertext/plaintext. We achieve this by selecting a byte and using it as an access index into a 256-page array, and examining which of the 256 pages has been accessed, thus retrieving one byte of the reduced-round ciphertext. Alternatively, we can also assume a side-channel oracle that only leaks whether a byte of the ciphertext equals a predefined value. In this case, we only need to check if a single cache line has been accessed or not, while repeating the attack several times with different random inputs until we detect that the transient ciphertext includes the expected byte.

Key Extraction Algorithm. Once we have the ability to obtain reduced-round AES ciphertexts (in particular an AES ciphertext that has two rounds), we can follow a similar key recovery strategy to that of [57]. We give a brief overview here but for brevity refer to [57] for the full cryptographic details. A two round ciphertext is computed as $RRC = k^2 \oplus SR(SB(k^1 \oplus MC(SR(SB(k^0 \oplus P)))))$, where k^i is the round key for round i. Since a two round AES ciphertext only contains one MixColumns operation, there is very little diffusion throughout. This allows for a relatively straightforward correspondence between the reduce-round ciphertext and plaintext which we can exploit to recover the round key and then the encryption key. Building upon this, we have developed a software algorithm capable of processing both reduced-round and full-round AES ciphertexts to extract the AES key, which significantly improves the practicality of our proof-of-concept.

Evaluation. We use the Intel-IPP [6] AES ECB Encryption function as our victim model, utilizing a 128-bit key for encrypting a 128-byte data block. As previously mentioned, our attack is capable of speculatively terminating the victim loop at any iteration, in this case ranging from the first to one less than the total number of rounds. We rigorously test all of these, leaking the ciphertext and subsequently verifying the correctness of the recovered reduced-round ciphertext. To establish a ground truth for comparison, we call the victim function with the reduced number of rounds and record the results. We determine the success rate by comparing the number of stolen bytes by the attacker that match the ground truth. We repeat this process 1000 times and calculate the average success rate. On average, the attack succeeds with a probability of 98.43%. It is noteworthy that our attack scenario extends beyond AES ECB encryption and is applicable to other cryptographic functions, including various AES modes (CBC, CFB, CTR, etc.), as they also employ a looped implementation susceptible to our attack strategy.

Comparison to Prior Works. Previous Spectre attacks on AES [57] manipulated the base predictor, affecting only the first or a few instances of a branch at a specific address. However, with complete control over the PHR and PHTs, we can influence an arbitrary instance of a branch executed many times with just one call. This becomes particularly crucial when the attacker aims to manipulate branch prediction in target code featuring loops or branch instructions executed many times with varying directions, or where a mispredict on a particular iteration would be helpful, or multiple mispredicts in different iterations.

10 Mitigation Strategies

This section discusses possible mitigation strategies.

10.1 Mitigating PHR Attacks

This research represents a major departure from prior research, which predominantly concentrated on the PHT state, by shedding light on the vulnerability of the PHR state to leakage, thereby exposing crucial information regarding the runtime control flow, and in particular the global ordering of

all control flow. The heightened sensitivity of fine-grained control-flow-reads underscores the increased importance of avoiding secret-dependent control flow, as these vulnerabilities become more readily exploitable. The most straightforward software-based solution for mitigating the *(Unlimited) Read PHR* is to flush the PHR using ***194 unconditional direct branches*** during context switching between different security domains. Because *unconditional direct branches* do not interact with the PHTs at all, this prevents the attacker from exploiting the PHTs as a side-channel to reconstruct the PHR beyond 194. Less costly, we could add a small, non-deterministic number of random branches into the PHR during context switching. This randomization of the PHR value would prevent attackers from obtaining the same PHR upon repeated calls to the victim, significantly reducing the attacker's ability to read the *PHR*. However, the proposed mitigation strategies are not without their limitations:

1. The full reverse engineering effort would have to be repeated for each new generation of processors from different vendors to confirm that these mitigations remain valid.
2. The randomization approach may still be vulnerable to a brute force attack (but likely requiring orders of magnitude more time).

Further study is needed to evaluate the effectiveness and efficiency of these approaches.

10.2 Mitigating PHT Attacks

In contrast to prior attacks [26, 35, 57], which induce mispredictions at simple conditional branches (e.g., bounds check 'if' statement) primarily through base predictor poisoning, our Read/Write PHT attacks enable highly precise PHT poisoning, allowing for the targeting of specific iterations within a loop. The introduction of fine-grained control-flow-writes makes it more critical to consider non-deterministic speculative control-flow when mitigating Spectre attacks, and significantly increases the set of dynamic branches that can be targeted effectively with a poisoning attack. To effectively counter the Read/Write PHT attacks, we propose two mitigation strategies: (1) flushing the PHTs during context switches or across security boundaries, and/or (2) physically partitioning the CBP into distinct security domains. For instance, the Half&Half defense mechanism [71] can be used to partition the PHTs between two SMT threads or between the userspace and kernel boundaries. However, this method is limited to partitioning the CBP into two parts and may not be suitable for complex scenarios involving the multiplexing of partitions across various security boundaries. Flushing the PHTs in software requires around 100k instructions (mostly branches) – we have run this. This is prohibitively expensive for all but the most security-critical scenarios. Better would be hardware support for flushing.

Additionally, several previous works have proposed novel designs for Branch Prediction Units to mitigate branch-based side-channel attacks, which, if implemented, could also mitigate Read/Write PHT attacks. These approaches can be broadly categorized into two categories: (1) partitioning-based designs [65] and (2) encryption-based methods [37, 79–82]. The Branch Retention Buffer (BRB) [65] partitions the most useful predictor components to isolate separate contexts. Lee et al. [37] proposed a two-level encryption method: randomizing the set-index by encrypting the Program Counter (PC) with a per-context secret key and encrypting data within each branch predictor entry. In STBPU [79], each software entity receives a unique, randomly-generated secret token (ST) that customizes the data representations, thereby enhancing security against potential attacks.

While each of these can be effective at isolating the PHT, they all fail to isolate the PHR. Thus, they are all susceptible to PHR Read/Write attacks. In particular, the PHR Read attack only makes use of the PHR and in no way depends on victim PHT entries. Thus, all attacks based on PHR Read would still work on these proposed secure predictors, with no modification. The Extended Read PHR attack does rely on victim PHT data, and would not work in its current form, and would likely be fully mitigated. To mitigate the PHR Read/Write attacks in hardware, an effective approach could be to implement a dedicated table of global histories (PHRs), with each security domain having its own designated PHR. This prevents the sharing of PHRs among different security domains.

11 Related Work

We can classify branch prediction based side-channel attacks based on the specific component they target. A complex, modern BPU features multiple components (e.g., BTB, IBP, RSB, and CBP) for making various predictions. The initial research on branch-based vulnerabilities primarily focused on exploiting the BTB [12–14, 25, 38, 68, 74, 78]. For instance, Acıiçmez et al. [12–14] introduced four attacks on BTB, demonstrating them against RSA implementations.

More recently, Spectre attacks [35] demonstrate that combining CBP/BTB/IBP mispredictions with speculative execution can be leveraged for security attacks. Spectre V1 targets the CBP, and exploits speculative execution to access data outside the bounds of an array, despite the presence of a bounds check (by forcing a misprediction of the checking branch). In Spectre V2, the attacker trains the target predictors (BTB and/or IBP) such that the victim speculatively jumps to a chosen disclosure gadget. Like Spectre V2, Barberis et al. [17] introduce cross-privilege Branch Target Injection (BTI) attacks [20, 21, 35, 78] on systems that employ isolation-based hardware defenses [7, 8]. This technique, known as Branch History Injection (BHI), manipulates branch history in the IBP to influence it to select a specific entry when transitioning from user to supervisor or guest

to VMX root mode. This, in turn, triggers the transient execution of a disclosure gadget at the predicted target.

BranchScope [26] targets the CBP via the base predictor structure. BranchScope fires off hundreds of thousands of random branches to make the CBP use the basic predictor instead of the complex global one (PHTs). It then creates collisions within the base predictor by carefully manipulating branch addresses, allowing it to potentially leak or modify the values stored in the base predictor entries. Similarly, BranchSpec [23] demonstrates that the branch predictor state undergoes updates in the speculative path, potentially leading to information leakage. These methods both bypass the complex structures of the CBP, because those details were unknown, but instead heavily leaned on the base predictor. In contrast, our approach leverages recently disclosed details [71] about the intricate workings of the entire CBP in Intel CPUs, making it a powerful and dangerous medium for side-channel attack. While that work used the reverse engineering information to devise a new isolation mechanism for the branch predictor, we exploit this knowledge to craft novel side-channel attacks.

12 Conclusion

This paper demonstrates that conditional branch predictors on recent machines, which are shared across all security domains, can be exploited for new microarchitectural attacks. With precise control of both the global history (path history register) and contents of prediction tables (base predictor and PHTs), attacks can be launched with an entirely new level of precision. Rather than being able to capture the biases (or most recent outcomes) of individual branches, this work captures a complete control flow history of all branch behavior going back at least tens of thousands of branches. While previous poisoning attacks typically target the first instance of a branch, this work can create a misprediction of any instance of a branch even if it is executed thousands of times.

These capabilities are demonstrated on two use case attacks. An attack on libjpeg captures all interesting features of an image by perfectly capturing the complete control flow of the routine. An attack on a constant-time AES algorithm exploits the ability to induce the main loop to exit and return intermediate results at any loop iteration.

Acknowledgments

The authors would like to thank to Shravan Narayan for his insightful discussions, and the anonymous reviewers for their helpful suggestions.

This work was partially supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022 and HR00112390029; the National Science Foundation (NSF) under grant numbers CNS-2155235, CNS-1954712, and CAREER CNS-2048262; the Alfred P. Sloan Research Fellowship; and gifts from Intel, Qualcomm, and Cisco.

References

[1] libjpeg. https://libjpeg.sourceforge.net/. [Online].
[2] libjpeg-turbo. https://libjpeg-turbo.org/. [Online].
[3] The 2nd jilp championship branch prediction competition (cbp-2). http://www.jilp.org/cbp2006, 2006. [Online].
[4] The 3rd jilp championship branch prediction competition (cbp-3). http://www.jilp.org/cbp2011, 2011. [Online].
[5] The 4th jilp championship branch prediction competition (cbp-4). http://www.jilp.org/cbp2014, 2014. [Online].
[6] Intel ipp, intel integrated performance primitives, 2023.
[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] Intel® software guard extensions (intel® sgx). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2023.
[10] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.
[11] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 110–124. Springer, 2010.
[12] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Cryptography and Coding*, pages 185–203, 2007.
[13] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Computer and Communications Security (CCS)*, pages 312–320, 2007.
[14] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *RSA Conference*, pages 225–242, 2007.
[15] Onur Acıiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Topics in Cryptology–CT-RSA 2008: The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pages 256–273. Springer, 2008.
[16] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache games - bringing access based cache attacks on aes to practice. Cryptology ePrint Archive, Paper 2010/594, 2010. https://eprint.iacr.org/2010/594.
[17] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, 2022.
[18] 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)*, pages 785–800, 2019.
[19] Romie Oktovianus Bura and Hanum Shirotu Nida. Image transmission using base64 encoding and advanced encryption standard algorithm based on socket programming. In *2021 6th International Workshop on*

Big Data and Information Security (IWBIS), pages 115–120, 2021.

[20] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[21] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[22] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[23] 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)*, pages 529–536. IEEE, 2020.

[24] Avinoam N Eden and Trevor Mudge. The yags branch prediction scheme. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 69–77, 1998.

[25] 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)*, pages 1–13, 2016.

[26] 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.

[27] F Fathurrahmad and E Ester. Development and implementation of the rijndael algorithm and base-64 advanced encryption standard (aes) for website data security. *International Journal of Scientific & Technology Research*, 9(11):6–11, 2020.

[28] Agner Fog. The microarchitecture of intel, amd and via cpus. https://www.agner.org/optimize/microarchitecture.pdf, 2023. [Online].

[29] B Gras, KAVEH Razavi, H Bos, and C Giuffrida. Tlbleed: When protecting your cpu caches is not enough. *Black Hat*, 2018.

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

[31] 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)*, pages 279–299, 2016.

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

[33] IBM. Power9 processor user's manual. Technical report, IBM, 2019.

[34] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[35] 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. *Communications of the ACM*, 63(7):93–101, 2020.

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

[37] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–25, 2020.

[38] 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)*, pages 557–574, 2017.

[39] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

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

[41] Muhammad Wito Malik, Diyanatul Husna, I Ketut Eddy Purnama, Ingrid Nurtanio, Afif Nurul Hidayati, and Anak Agung Putri Ratna. Development of medical image encryption system using byte-level base-64 encoding and aes encryption method. In *Proceedings of the 6th International Conference on Communication and Information Processing*, pages 153–158, 2020.

[42] Scott McFarling. Combining branch predictors. Technical report, Citeseer, 1993.

[43] Pierre Michaud. A ppm-like, tag-based branch predictor. *JILP-Championship Branch Prediction*, 7:10, 2005.

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

[45] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. {CopyCat}: Controlled {Instruction-Level} attacks on enclaves. In *29th USENIX security symposium (USENIX security 20)*, pages 469–486, 2020.

[46] Muhammad Farras Muttaqin and Silvester Dian Handy Permana Yaddarabullah. Implementation of aes-128 and token-base64 to prevent sql injection attacks via http. *International Journal*, 9(3), 2020.

[47] Khang T Nguyen. Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2016. [Online].

[48] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *Cryptology ePrint Archive*, 2005.

[49] Shien-Tai Pan, Kimming So, and Joseph T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–84, 1992.

[50] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings 8*, pages 138–154. Springer, 2016.

[51] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. Frontal attack: Leaking {Control-Flow} in {SGX} via the {CPU} frontend. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 663–680, 2021.

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

[53] André Seznec. A 256 kbits l-tage branch predictor. *JILP-Championship Branch Prediction*, 9:1–6, 2007.

[54] André Seznec. A new case for the tage branch predictor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 117–127, 2011.

[55] André Seznec. Tage-sc-l branch predictors again. In *JILP-Championship Branch Prediction*, 2016.

[56] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *JILP-Championship Branch Prediction*, 8:23, 2006.

[57] 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 *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1753–1770, 2023.

[58] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[59] James E. Smith. A study of branch prediction strategies. In *International Symposium on Computer Architecture (ISCA)*, page 135–148, 1981.

[60] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[61] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture (ISCA)*, pages 392–403, 1995.

[62] 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)*, pages 207–217, 2009.

[63] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[65] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Brb: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477. IEEE, 2019.

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

[67] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[68] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO Conference 2023*, 2023.

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

[70] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *USENIX Security Symposium (USENIX Security)*, pages 719–732, 2014.

[71] 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 *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237. IEEE Computer Society, 2023.

[72] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 51–61, 1991.

[73] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20(2):124–134, 1992.

[74] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. All your pc are belong to us: Exploiting non-control-transfer instruction btb updates for dynamic pc extraction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023.

[75] Jann Horn Google Project Zero. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[76] Jann Horn Google Project Zero. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2019.

[77] Charles Zhang. Mars: A 64-core armv8 processor. In *Hot Chips*, pages 1–23, 2015.

[78] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 667–682, 2020.

[79] Tao Zhang, Timothy Lesch, Kenneth Koltermann, and Dmitry Evtyushkin. Stbpu: A reasonably secure branch prediction unit. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–123. IEEE, 2022.

[80] Lu-Tan Zhao, Rui Hou, Kai Wang, Yu-Lan Su, Pei-Nan Li, and Dan Meng. A novel probabilistic saturating counter design for secure branch predictor. *Journal of Computer Science and Technology*, 36:1022–1036, 2021.

[81] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1267–1272. IEEE, 2021.

[82] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. Hybp: Hybrid isolation-randomization secure branch predictor. In *HPCA*, pages 346–359, 2022.