Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution

Hosein Yavarzadeh*, Mohammadkazem Taram†, Shravan Narayan*,§, Deian Stefan*, and Dean Tullsen*

*University of California San Diego, †Purdue University, §University of Texas at Austin

***Abstract*—This paper presents Half&Half, a novel software defense against branch-based side-channel attacks. Half&Half isolates the effects of different protection domains on the conditional branch predictors (CBPs) in modern Intel processors. This work presents the first exhaustive analysis of modern conditional branch prediction structures, and reveals for the first time an unknown opportunity to physically partition all CBP structures and completely prevent leakage between two domains using the shared predictor. Half&Half is a software-only solution to branch predictor isolation that requires no changes to the hardware or ISA, and only requires minor modifications to be supported in existing compilers. We implement Half&Half in the LLVM and WebAssembly compilers and show that it incurs an order of magnitude lower overhead compared to the current state-of-the-art branch-based side-channel defenses.**

1. Introduction

Isolation is a fundamental goal of any secure multi-user system, providing assurance that one domain is in no way influenced by the actions of another. We want the user to be isolated from the kernel, co-running processes (including on a simultaneous multithreaded processor) to be isolated from each other, VMs to be isolated, sandboxed code to be isolated from the host program that runs this sandboxed code, etc. While significant advances have been made in software and language based isolation, recent disclosures have illustrated that it is not possible to fully provide isolation without isolating microarchitectural structures [47], [54].

The primary forms of isolation we want to guarantee are memory isolation and control flow isolation, i.e., memory accesses and control flow paths in one domain should not leak information to, or be maliciously influenced by, another domain. Memory isolation has been well studied and supported at both the software and hardware level; for example, operating systems ensure memory isolation between different processes using virtual memory and page table entries that are either flushed between processes or tagged with process IDs, and processors even have hardware support to isolate cache lines [64].

In contrast, control flow isolation has only very limited support, and as a consequence has been the target of many attacks. For example, the branch prediction based information leakage has been exploited to recover the encryption key from the RSA algorithm [10], [11], [12] and to break Address Space Layout Randomization (ASLR) [26]. More recently, Spectre attacks [21], [47], [17] have demonstrated how an attacker process can infer or manipulate a victim process' control flow during speculative execution in order to disclose memory of this victim process. Preventing such attacks requires robust control flow isolation at the hardware level.

Control flow isolation on modern CPUs typically involves isolating two structures: the Branch Target Buffer (BTB) and the Conditional Branch Predictor (CBP), however, much of the existing practical defenses have been geared towards the BTB. The BTB for instance can be directly flushed by software, which allows partitioning of the BTB in time (rather than in space). Alternately, the BTB can be bypassed altogether for security sensitive programs [89]. Unfortunately, similar mechanisms to partition the CBP do not exist. One of the reasons for this is the sheer complexity of the CBP predictors used in real CPUs. As a consequence, existing software control flow isolation defenses opt to forego the use of the CBP altogether, either eliminating branches [15], [20] or forcing all control flow to go through indirect branches, at high performance cost [51], [62].

At its core, CBP isolation has been challenging for two reasons. First, commercial CPU vendors are notoriously secretive about their predictors, therefore, reverse engineering those structures is difficult; indeed, no prior work has successfully identified all the key predictor structures in recent advanced CPUs. Second, the indexing functions used in the internals of this complex predictor involves hundreds of bits of information folded together; this makes it nearly impossible to track and control all the ways these bits affect predictions from the CBP.

Despite these challenges, we demonstrate that it is in fact possible to automatically partition the CBP in today's CPUs, completely in software, with minimal performance impact. Our approach incurs more than an order of magnitude lesser overhead than other state-of-the-art defenses. This result is enabled by a comprehensive reverse engineering of the branch predictors on three high-end Intel processor families, revealing for the first time the structures and index functions of all tables in the predictor. Our analysis reveals the unexpected result that despite hundreds of bits used to index the branch prediction tables in these processors, a single bit of the branch address is used without modification as an independent bit of the index function of every table in the predictor. Thus, two

DOI 10.1109/SP46215.2023.00134

branches that differ in this single bit can never influence each other. By partitioning the branch predictor on this bit, we can prevent side channel attacks on the CBP [25], [27], [28], [36] and prevent malicious mistraining of the CBP required by transient execution attacks [46], [47].

One limitation of our approach is that while we can partition the CBP into two isolated domains, further partitioning becomes prohibitively expensive. Nevertheless, we believe this two-domain partitioning is still useful in many scenarios. We can, for example, partition the CBP between userspace code and kernel code, partition two threads running on simultaneous multithreaded (SMT) cores, or even partition the CBP between code from untrusted application components and the remainder of the application.

We implement our CBP partitioning defense mechanism on top of a general purpose compiler (LLVM) as well as Swivel [62] – a WebAssembly compiler that sandboxes untrusted code in applications so that it cannot be coerced to corrupt or leak the memory contents from the trusted code in these applications through memory safety attacks or Spectre attacks.

Contributions. The contributions of this work are:

- We reveal, for the first time, a comprehensive picture of the branch predictors in three of the most recent families of Intel processors, including the size, structure, and exact indexing function of each table.
- We propose Half&Half, a novel software-based defense against CBP poisoning/aliasing attacks and against CBP side channels.
- We implement our defense mechanism on top of a general purpose compiler (LLVM) and show execution overhead of 1.3%-6.8%.
- We implement Half&Half in a recent Spectre-hardened WebAssembly compiler, Swivel, replacing their CBP isolation mechanism with ours, and show an order of magnitude reduction in overhead.

2. Background and Related Work

This section discusses relevant background information and prior work. It first provides an overview of the known state-of-the-art branch prediction structures used in modern processors. It then discusses the most relevant research on branch predictor-based attacks and mitigations.

2.1. Branch Prediction

All modern high-performance processors employ dynamic branch prediction [83], [101], [44], [80], [77], [55], a crucial performance optimization which allows them to maintain high pipeline utilization. To continuously fetch and execute instructions after the branch, the processor needs three different predictions from the Branch Prediction Unit (BPU): (1) whether or not the current instruction is a branch, (2) whether the branch is taken or not, and (3) where the target of the branch is. A BPU, therefore, typically features different structures for these different predictions. The Branch Target Buffer (BTB) identifies branches at fetch time and predicts the target, while the Conditional Branch Predictor (CBP) provides predictions for the direction of branches (taken or not-taken). It should be noted that predicting the targets of computed (or indirect) branches is much more difficult than predicting static targets, therefore that part of the BTB that predicts those target addresses is often called the Indirect Branch Predictor (IBP).

Previous research has extensively studied the structure of the BTB and the IBP and exposed BTB-based vulnerabilities for various high performance processors from different vendors including Intel [59], [91], [26], [35] and AMD [9], [90], [106], [5]. However, the structure of the conditional branch predictor in most modern processor designs is still largely unknown.

2.2. Conditional Branch Prediction

Conditional branch predictors proposed in the literature are mostly history-based predictors, i.e., they predict the direction of branches based on previous outcomes [83], [101], [44], [80], [77], [55]. The type of history that these predictors use for each branch can be categorized as local or global. The *local history* captures previous outcomes of the same branch while the *global history* tracks the outcome of any branch that the processor executes. A simple local predictor is the bimodal, a table indexed by low bits of the branch address and composed of two-bit saturating counters [50], [55]. A high counter value predicts the branch (at the address) to be taken, while a low value predicts the converse; when the branch is actually resolved as taken or not, the counter is incremented or decremented respectively. A local predictor (including those that capture patterns rather than just tendencies [101]) lacks sufficient context to accurately predict many branches whose behavior is influenced by the path the code took to reach the branch (global history). These are called correlated branches, and previous research [102], [67] shows they constitute a significant portion of branches in many programs.

In practice, state-of-the-art predictors [80], [75], [77] use a combination of both local and global histories. Global predictors traditionally maintain a global history of past (dynamic) branches in a shift-register called the *Global History Register (GHR)*. When the processor executes a branch, it inserts the outcome of the branch into the GHR, e.g., '0' if it was non-taken and '1' otherwise. This GHR is used in various ways to index into the branch predictor tables. The length of the GHR is a sensitive parameter, as a small GHR fails to capture correlations between branches more separated in the program flow, while a large GHR creates a great deal of noise (for uncorrelated branches) that obscures the few correlations that typically matter. To address this problem, researchers have proposed using multiple tables each indexed by different history lengths [74], [79], [42], [43]. The O-GEHL predictor [73] proposed using multiple tables indexed by history lengths that form a geometric series, e.g., the first table uses the history of the past 5 branches, the second table uses the history of the past 10 branches, the third table uses the history of the past 20 branches and so on.

TAGE In 2006, Seznec [80] proposed a predictor called *Tagged Geometric History Length Predictor*, or *TAGE*. Similar

to the O-GEHL predictor, TAGE uses geometric history lengths. TAGE, however, relies on a hash of the global history bits, combined with tags on table entries, to associate counters with their global histories. Since its introduction, variants of TAGE [80], [76], [77], [78] have won each branch predictor championship [1], [2], [3], [4]. It is known that the TAGE predictor has been implemented in many commercial high-performance processors, e.g., IBM POWER9 [37] and Phytium Mars [105], and due to its accuracy, it is likely employed by other recent commercial high-performance processors. Therefore, this section delves deeper into details of the TAGE predictor.

TAGE (see Figure 1) features a base predictor (Table_0) alongside a set of tagged predictor components (Table_1 to Table_n). The base predictor is a table of 2-bit saturating counters and is simply indexed with the branch address, i.e., the value of the Program Counter (PC) when the processor fetches the branch. The tagged tables are indexed using different hash functions of the branch address and the history lengths. An entry in a tagged component consists of a saturating counter which provides the prediction, but it also has some metadata: (1) a tag value that indicates the history to which this entry belongs, and (2) a usefulness counter that is used for replacement decisions.

At prediction time, the TAGE predictor simultaneously queries the base predictor and the tagged components. The tagged components provide a prediction only on a tag match, in other words, only if we observe a repeated history for a branch. The overall prediction is provided by the tagged predictor component that uses the longest history, or in case of no matching tagged predictor component, TAGE uses the prediction provided by the base predictor.

On a correct prediction, TAGE updates the prediction counter. If the prediction was from a tagged component, TAGE also updates the usefulness counter, indicating that the entry has been useful, making it less likely to be replaced. In case of a misprediction where the prediction was provided by the base predictor, TAGE updates the counters and allocates a new entry in the first tagged component (Table_1). In the more general case, TAGE tries to allocate a new entry in Table_{i+1} if the prediction from Table_i is incorrect.

2.3. Branch-based Side-Channel Attacks

We broadly categorize branch based attacks into two categories—attacks that can learn the control flow of an isolated (victim) process, and transient execution attacks that disclose the memory contents of a victim process by leveraging branch mispredictions and speculative execution.

Control flow extraction attacks Prior work has demonstrated a number of attacks on the branch prediction unit by reverse engineering the internal BPU structures. For instance, the first BPU attacks by Acıiçmez et al. [10], [11], [12] targeted the BTB—the component that stores branch targets for a limited number of indirect branches in executing code. These attacks showed that a malicious process could fill the limited BTB target entries with dummy values to cause measurable timing differences in a victim

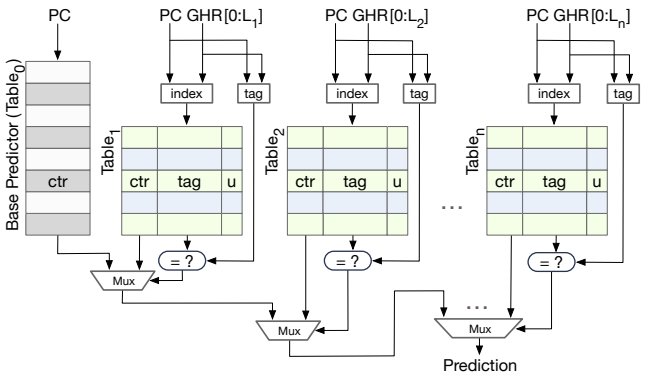


Figure 1: Structure of the TAGE predictor. As proposed in [80], a base predictor is backed with several tagged predictor components ($\mathsf{Table}_{1\ldots n}$) indexed by the program counter (PC) and increasing geometric history (GHR) lengths ($L_1 < L_2 < \cdots < L_n$).

processes. This, in turn, allows the attacker process to infer the control flow and secret data of the victim process. Evtyushkin et al. [26] similarly used BTB collisions between an unprivileged user space process and privileged kernel space code to break the kernel's ASLR. Lee et al. [51] used BTB collisions to infer the control flow of programs running in Intel SGX—a hardware extension that allows secure isolated execution of programs.

BranchScope [28] expanded BPU attacks to target the conditional branch predictor (CBP) by reverse engineering a part of the CBP structure. They observed that the CBP in modern Intel processors was a combination of a local predictor that provides predictions based on the local history, alongside a more complicated predictor that uses global history. BranchScope works by executing many random branches to force the CBP to use the local predictor, and then induces collisions in the CBP entries (similar to the BTB attacks). BranchScope, however, stopped short of a full analysis of the CBP structure, especially the global predictor, and how this may be used in real attacks as well as practical defenses—we cover this in detail in Section 3.

Spectre v1 Recently, Spectre attacks [47] have shown that leveraging CBP or BTB mispredictions in combination with speculative execution can be used for attacks. At their core, Spectre attacks work by leveraging branch mispredictions to execute code that would leak sensitive data. Specifically, they reverse engineer some structures of the indirect branch predictor (target predictor) for Haswell architecture and used this information to do drive spectre-btb attacks. In this paper, we target the CBP which is a completely different structure than the indirect branch predictor.

Listing 1 demonstrates how the out-of-place Spectre-PHT attack [17] can be used to learn the memory contents of a victim process. The attack starts with the attacker process training the branch predictor to predict "Taken" on a branch located at address `0x1234` in the attacker process. When the victim process runs next, the branch predictor will predict "Taken" on the branch in victim code (as the victim branch has the the same address as the attacker process'

```
void attacker_process(){
  //Training branch predictor
  for (i=0; i < INT_MAX; i++){
    //Branch addr: 0x1234
    if (TRUE);
  }
}
```

```
void victim_process(){
  //Branch addr: 0x1234
  if (i < array1_size){
    data = array1[i] * 4096;
    y = array2[data];
  }
}
```

Listing 1: Spectre-PHT gadget. A read from `array2` loads data into the cache at an address dependent on `array1[input]`, using the attacker-controlled `input`.

branch), and speculatively execute the branch body even if "`i`" is very large. This results in a speculative read gadget which can read data from any memory location into the variable `data`. This `data` can then be leaked either through the data cache [87], [87], [98], [30] by leveraging a double index gadget as shown, or via other covert channels [22], [72], [16], [69]. Similar attacks are also possible across the userspace and kernel boundary [84], and even across different components of a single process [62].

Spectre attacks also have in-place variants [47], [46], [21], where attackers invoke victim code with chosen inputs so that the victim branch is trained in-place, i.e., without relying on any collisions in BPU entries. While we mention this for completeness, we consider these in-place attacks beyond the scope of this paper; those attacks do not exploit a contention/isolation based vulnerability.

2.4. Defenses Against Branch-based Attacks

Several defenses have been proposed to stop BPU attacks targeting the CBP or BTB. While many of these BTB defenses have proven effective, with some even being deployed in practice [84], [38], the use of a general, high performance, and backwards-compatible CBP defense has remained elusive; we discuss this in more detail next.

The most general defense against contention-based side-channel attacks is to partition the targeted unit between different processes or domains [52], [99], [107], [85]. This applies to general defenses against BPU attacks as well. Prior work [107] has proposed hardware design changes that implement this change, however, this design has not yet seen mainstream adoption in commercial CPUs. Intel has instead proposed targeted mitigations such as isolating the branch prediction of hyperthreads [38], however, in practice, this simply disables branch prediction in one of the hyperthreads resulting in large overheads [23]. A related but simpler defense, available in Intel CPUs as an optional mitigation, is to simply flush the BTB state when switching contexts [84]; however as we discuss in Section 4.2, employing a similar flush for the CBP in CPUs is challenging.

An alternate defense used by sensitive code such as cryptographic libraries is constant-time programming [15], [20]. This approach eliminates secret-dependent branches from code by implementing algorithms as circuits. However, this approach is not typically employed in general purpose code due to the high overheads.

Spectre defenses Several defenses have been proposed to specifically mitigate Spectre based attacks. These include changes to CPU design to permit safe speculation [104], [103], [95], [14], [53], limiting the effects of speculation on structures such as the cache [97], [13], [70], [96], and even limiting speculative execution when handling sensitive data [86]. Additionally, many designs [32], [49], [71], [108], [29], [63] have combined CPU design changes with software provided hints to speed up enforcement of safe speculative execution. These changes offer a path forward on the next generation of CPU designs to prevent Spectre attacks at the cost of some performance.

To address attacks on current hardware, CPU manufacturers and software vendors have turned to microcode patches and software solutions. The core challenge with these approaches is the security vs. performance trade-off; the more secure options like microcode patches [82] or inserting fences in programs [58], [40], [94], [81] significantly affect performance, while Spectre gadget finders [31], [66] or targeted use of secure coding patterns (retpolines [89], speculative load hardening [19], or artificial data dependencies [65] near branches) are inherently incomplete defenses.

Spectre defenses have also been developed or adapted for specific domains. For example, JavaScript engines in browsers use speculative load hardening for memory operations on JavaScript arrays [24]. Swivel [62] proposes a code pattern called linear blocks in combination with techniques like BTB flushing, and speculative load hardening to secure WebAssembly [33] — a technology that allows sandboxed execution of untrusted code components in a process.

Our work proposes a new general-purpose, low-overhead method of partitioning the CBP in existing CPUs. By leveraging these partitions, we can prevent branch based attacks targeting the CBP. In order to partition the CBP, we start by conducting a series of experiments to uncover the design and inner workings of the BPU in today's CPUs. We explain these experiments next.

3. In-Depth Analysis of Intel CBPs

This section details our in-depth study of Intel's conditional branch predictors and the techniques and the microbenchmarks used to uncover their internal structures. The goal is to discover if it is possible to find a set of transformations that would partition the CBP to provide secure isolation. To do that, we need to know the structure and indexing/access functions for each table in the predictor. We start with analyzing the type and the size of the global history that Intel uses to predict correlated branches, for two reasons – it's the easiest to reason about without knowing other details, and the size and structure of the global history often tells us something about the structure of the predictor itself. We then expand our study to include other undocumented and previously unknown details of the predictor, including the number and the size of the predictor tables and their indexing functions.

3.1. Assumptions

Since the Intel CBP structure is completely undocumented, we must start with only a small set of assumptions: We assume the CBP uses some form of global history, the CBP has a TAGE-like structure, i.e., it has multiple tables to

Table 1: Specifications of the analyzed processors.

Machine	machine 1	machine 2	machine 3	machine 4
Model Name $\mu Arch.$	Core i9-12900K Alder Lake	Core i7-1165G7 Tiger Lake	Xeon Gold 6314U Ice Lake	Xeon Gold 6230 Cascade Lake
Machine	machine 5	machine 6	machine 7	
Model Name $\mu Arch.$	Core i7-6770HQ Skylake	Xeon E5-1650 V3 Haswell	Xeon E3-1230 V2 Ivy Bridge	

store the global predictions and the tables are indexed with global history. In sections 3.3 and 3.4 we will validate these assumptions and recover the detailed structure of the CBP.

3.2. Experimental Setup

We study the conditional branch predictors (CBPs) used in three different Intel microarchitectures, described in Table 1. For brevity, we focus on the Skylake microarchitecture in the text, and only mention other microarchitectures when they differ in important details. We analyze the behavior of the conditional predictor using carefully-crafted microbenchmarks. For simplicity, the microbenchmarks are presented here as a mixture of C and x86 assembly; however, in practice, these are all written directly in x86 assembly for precision. We use the performance monitor counters (PMCs) to measure the outcome of experiments. Specifically, we use performance counters relevant to branch prediction such as the number of taken (or not-taken) branches and the number of mispredicted taken (or not-taken) branches.

It should be noted that the experiments described in this paper are a small fraction of all tests run in our effort to reverse engineer this unit. What is presented here is the minimal subset of those experiments that establish the details necessary to understand the features of the CBP critical to our isolation defense.

3.3. Global History

As discussed in Section 2.2, the CBPs in modern processors use global history to capture the correlation between dynamic branches. The global history is particularly important to this analysis because, of all the features used by modern predictors to access their tables, it is the hardest for the system/user/compiler to control. The fact that it is so tightly integrated into the prediction mechanisms in state-of-the-art predictors is likely, therefore, to create a challenge. Specifically, we want to know the size of the global history (in number of branches recorded), what features are recorded in the global history, and how it is updated at each branch.

Prior work has shown that these processors use global history to access the IBP [47], [35]; however, there is no reason to believe they use the history in similar ways, so we start with a blank slate in this study.

Recovering the Size of the Global History To uncover the maximum size of the global history used in the CBP we start with two correlating branches that appear immediately after each other. We then increase the distance between the correlated branches by inserting additional branches in between. We hypothesize that after a certain distance, due to limited capacity of the global history, the branch predictor will no longer capture the correlation between the two branches (and start mispredicting).

```
macro dummy_branches(n, i = 0, j = 0)
  rep iter in (1..n)   //n = Number of Dummy Branches
    align(i)   jmp label_iter
    align(j)   label_iter:
  endrep
endmacro
```

Listing 2: Helper macro for creating dummy branches.

```
for (iter=0; iter < NUM_TRIES; iter++){
  k = rand();
  if(k);              //Train Branch (Miss Rate = ~50%)
  dummy_branches(n);  //n = Number of Dummy Branches
  if(k);              //Test Branch
}          //Miss Rate = ~0% if correlation captured
```

Listing 3: Microbenchmark pseudo-code for detecting the maximum size of the global history used in CBP.

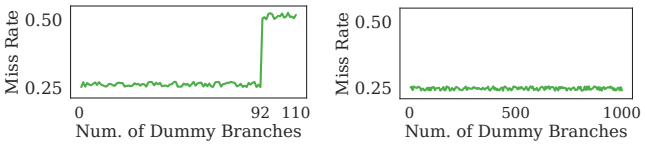
Listing 3 shows the code snippet we use to determine the size of the history. It consists of a branch (test branch) whose outcome is directly correlated with the outcome of another branch (train branch). The train branch is conditioned on a random bit (line 2); this ensures that its direction cannot be predicted locally nor globally. The test branch, however, *can* be predicted if the branch predictor captures its correlation with the train branch. In between these two branches, we insert a variable number of dummy branches until we exhaust the global history.

For this experiment, we use unconditional (and thus taken) branches as dummy branches. While it is not obvious that unconditional branches would even be stored in the global history, our experiments (including this one) confirm that they are. We use unconditional branches instead of conditional always-taken branches because they are simpler and result in measurements that are less noisy.

We increase the number of dummy branches (n) from 0 to 100, and measure the mispredictions rate for the two branches. For the train branch, the predictor has a miss rate of 0.5; this is expected since branch condition is random. For the test branch, we expect to see one of two possible outcomes for each value of n: (1) the predictor captures the correlation to perfectly predict the test branch (miss rate of 0), or (2) the predictor does not capture the correlation, making it unpredictable (miss rate of 0.5). Thus, on average, when we measure the predictor accuracy for the two branches together, we expect to see a miss rate of 0.25 if the predictor captures the correlation between branches, and a miss rate of 0.5 otherwise.

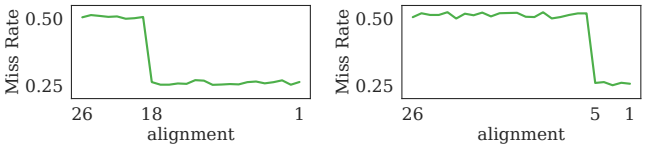
Figure 2a summarizes our results. We find that the CBP can capture the correlation between the train and test branches when there are up to 92 taken dummy branches between them. This implies that global history stores the footprint of the last 93 taken branches (the train branch plus 92 dummy branches).

Since the global history could also be affected by *not-taken* branches, we evaluate the effects of not-taken branches. Specifically, we repeat our experiment but insert not-taken dummy branches branches between the train and test branches. We find that the CBP can predict the test branch regardless of the number of not-taken branches: as Figure 2b shows, even with hundreds of not-taken dummy branches between the correlated branches, the misprediction



(a) Taken dummies. **(b) Not-taken dummies.**

Figure 2: Miss rate of two correlated branches due to variable number of dummies between them. This helps determine the maximum size of global history.



(a) Branch address. **(b) Target address.**

Figure 3: Miss rate of two correlated branches due to variable alignment of the inputs to the PHR.

rate remains constant (at 0.25).

Finally, we repeat these experiments by randomizing both the branch addresses and their targets. We do this to ensure that the addresses of the branches (and their targets), which affect the global history (as we will see shortly), do not invalidate our observation.

> **Observation 1.** Global history records the history footprint of the last 93 taken branches, whether they are conditional or unconditional. Not-taken branches do not affect the history.

This observation indicates that our predictor already deviates from expectations. It does not have the standard *global history register* structure. Global history as used for conditional branch prediction is typically implemented in one of two ways:

- **Global History Register (GHR):** a shift register that inserts "1" when a conditional branch is taken and "0" when it is not taken [55], [80]. The GHR always inserts the direction of the conditional branch regardless of whether it is taken or not taken. The GHR is by far the most common form of global path history in the literature [101], [55], [44], [57], [74], and is used in the original TAGE predictor [80].
- **Path History Register (PHR):** a shift register that inserts a bit (or a few bits) every time a conditional branch is *taken* [61], [74]. The PHR in [73], for example, inserts one address bit per conditional branch.

From Observation 1, it is clear that the Intel CBPs use a PHR for global history. However, this PHR also deviates from PHRs used in the literature, which are only updated for conditional taken branches and ignore unconditional branches. In the rest of this section we describe our work recovering the structure of this PHR.

Recovering inputs to the PHR We start by identifying the inputs to the PHR. As mentioned above, a PHR typically inserts a set of address bits on each taken branch. These bits can be from the address of the branch instruction itself, the address of the target, or a combination of the two.

```
for (iter=0; iter < NUM_TRIES; iter++){
  cmp rand(),0;             //Condition of Branches
  dummy_branches(93,i,j);   //Clearing PHR
  align(i)    je lbl_tr     //Train Branch
  align(j)    lbl_tr:       //Miss Rate = ~50%
  je lbl_ts                 //Test Branch
  lbl_ts:     //Miss Rate = ~0% if correlation captured
}
```

Listing 4: Microbenchmark pseudo-code for recovering inputs (branch and target addresses) to the PHR.

We assume that the i lower bits of the branch address and the j lower bits of the target address are used to update the k lower bits of the PHR when the branch is taken. Our goal is to experimentally validate that the PHR actually has this structure—and to recover the structure details, i.e., i and j (and later k). We do this using an experiment similar to our first (see Listing 4). In this experiment, though, we vary the train branch address (and the target address) to understand which bits affect the test branch prediction rate.

We start by clearing the PHR, i.e., setting the PHR to all-zeros, before the train branch. We do this using 93 taken dummy branches. From Observation 1, we know that only the last 93 branches have an impact on the PHR. But, since we do not know which bits from the addresses are used to update the PHR, we use dummy branches that have an address aligned to i_0 and a target aligned to j_0 bits, so that the lowest i_0 and j_0 bits are all zero respectively. If i_0 and j_0 are large enough, i.e., $i_0 \geq i$ and $j_0 \geq j$, the PHR will (very likely) be zero: each branch would insert at least one zero into the shift register. We test this by aligning our train branch (and its target) similarly and measuring the test branch prediction rate: if the PHR is indeed zero, we expect the misprediction rate for the correlated test branch to be high—we effectively do not have any history from the train branch.

We vary i_0 and j_0 (from 1 to 26) and find that the $i = 19$ lower bits of the branch address and $j = 6$ lower bits of the target address are used to update the PHR. Figure 3a shows our findings for the former: the misprediction rate increases from 0.25 (correlation captured) to 0.5 (both train and test branches mispredicted) when we increase the branch address alignment from 18 to 19, but keep alignment of the target address fixed (to 19). Figure 3b shows our findings for the latter: the misprediction rate jumps when the target address alignment goes from 5 to 6. Therefore, in the Skylake microarchitecture, only the 19 lower bits of branch address and 6 lower bits of target address are used to update the PHR when a branch is taken.

The next step is to recover *which* (of the i and j) bits are used to update PHR. As shown in Listing 5, we do this by first clearing the PHR and then flipping one bit of the branch address or target address at a time. Iff a bit is used to update PHR, we expect the CBP to capture the correlation between the train and test branches. We find that only 16 (of 19) bits of the branch address are used to update the PHR—the three least significant bits are not used. All 6 lower bits of the target address are used to update the PHR.

```
for (i=0; i < NUM_TRIES; i++){
  cmp rand(),0;
  dummy_branches(93,19);  //Clearing PHR
  SET_ADDRESS(variable);  //PC[18:0] = variable
  je lbl_tr               //Train Branch, Miss Rate = ~50%
  SET_ADDRESS(variable);  //PC[5:0] = variable
  lbl_ts:
  je lbl_ts               //Test Branch
  lbl_tr:     //Miss Rate = ~0% if correlation captured
}
```

Listing 5: Microbenchmark pseudo-code for recovering the update function of the PHR.

Table 2: Recovering positions of affected bits in PHR.

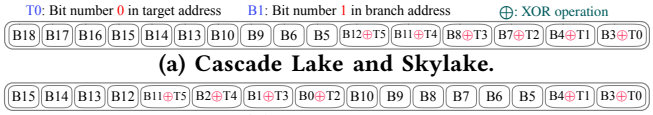
Flipped bit	dummy branches	Flipped bit	dummy branches
T0, B3, T1, B4	92	B9, B10	88
T2, B7, T3, B7	91	B13, B14	87
T4, B11, T5, B12	90	B15, B16	86
B5, B6	89	B17, B18	85

Observation 2. In the Skylake microarchitecture, the following bits are used to update the PHR:
- ▶ Branch Address [18:3] : 16 bits
- ▶ Branch Target Address [5:0] : 6 bits

Recovering the PHR update function Our next step is to understand how the PHR is precisely updated to account for new branches. We assume the PHR is not $(16+6)\times 93$ bits deep, but rather these bits are folded into the PHR via XOR (since both AND and OR are biased) and shifted before the next branch is included, following the design of PHRs from the literature [61], [73]. To understand which bits are folded in this *update function*, we use an experiment similar to that of Listing 5: we first clear the PHR, then we flip a pair of bits from branch address and target address (including all permutations of 2 bits out of the $(16+6)$ bits) to figure out if they are combined in the PHR update function (or not). Figure 4 summarizes our findings – which bits of the target address are XORed with which bits of the branch address. In the Skylake microarchitecture (Figure 4a), for example, we find that bit 0 of the target address is XORed with bit 3 of the branch address: when we flip both of these bits, the branch predictor cannot capture correlation between train and test branches, i.e., they cancel each other's effect in the PHR update function.

To recover the exact PHR update function, we also need to figure out how many bits the PHR is shifted by on each update and precisely which bits of the PHR are affected on each update. We do this with several experiments. First, we repeat the history length experiment (Listing 3) more precisely: we clear the PHR, then we flip only one bit of the branch address or the target address to figure out the maximum number of dummy branches that can be inserted in between train and test branches while still capturing the correlation between them. Our insight is that the dummy branches shift the *footprint* of the train branch through the PHR, and thus how long a particular bit remains depends on the position where it was originally inserted.

Table 2 summarizes our findings. We find, for example, that when B5 (bit number 5 in the branch address) is flipped, the CBP can capture correlation between train and test branches with up to 89 dummy branches in between. In other words, 89 dummy branches place the affected bit by B5 in the most significant bit (MSB) of the PHR, and the 90th dummy branch will toss it out. By examining the number of dummy branches that place the affected bit of the PHR in the MSB position, we recover the PHR update function. Our exhaustive experiments reveal that each taken branch clears two bits from the PHR and affects the 16 lower bits of the PHR. We call these 16 bits the branch *footprint*. From these findings we draw two conclusions: (1) the PHR is a shift-register that always shifts two bits per taken branch, and its size is 93×2 bits, and (2) the branch address and target address footprint (given in Figure 4) is XORed with the PHR after it is shifted.



(a) Cascade Lake and Skylake.

(b) Alder Lake.

Figure 4: *Footprint* in Alder Lake and Skylake.

Table 3: PHR structure in Intel microarchitectures.

Microarchitecture	PHR Size	Branch Addr.	Target Addr.	Footprint
Alder Lake/Tiger Lake/Ice Lake	194×2	$B[15:0]$	$T[5:0]$	Figure 4b
Cascade Lake/Skylake	93×2	$B[18:3]$	$T[5:0]$	Figure 4a
Haswell/Ivy Bridge	93×2	$B[19:4]$	$T[5:0]$	Figure 14

Observation 3. The PHR is updated in two steps whenever a branch is taken by:
1) Shift two bits to the left: `PHR = PHR << 2`
2) XOR the 16-bit footprint into the PHR: `PHR[15:0] = PHR[15:0] ⊕ footprint`

Table 3 summarizes the fully deconstructed structure of the global history for various Intel microarchitectures, which only differ in minor ways. Previous work [47], [35] has employed similar approaches to find the global history used for indirect branch prediction. While CBP's PHR update function is interestingly similar to what has been found for indirect branch prediction [47], [35], they differ in size and how they are used to access the structures they respectively target.

3.4. Pattern History Tables (PHTs)

In this section we use our new understanding of the PHR to recover the structure of the CBP tables. This is critical for building defenses against branch-based attacks; simply put, we cannot reason about defenses that attempt to partition these CBP tables without knowing the structure of the tables and how they are indexed.

The CBPs have a TAGE-like structure [80], [76], [77]. In a TAGE-like predictor, tagged components, or PHTs, are the main data structures that store the actual predictions for correlating branches. Hence, to recover the CBP tables, we first need to know their inputs, i.e., what information the CBP uses to look up a table and retrieve a prediction. These components, as described in Section 2.2, are typically indexed using a combination of the global history (PHR) and the branch address (PC). We start by discovering which

```
for (i=0; i < NUM_TRIES; i++){
  k = rand();
  SET_PHR(k0..0);    //PHR = k0...0
  align(alignment)   //PC[alignment:0] = 0
  if(k);             //Test Branch 1
  SET_PHR(k0..0);    //PHR = k0...0
  align(alignment)   //PC[alignment:0] = 0
  if(!k);            //Test Branch 2
}
```

Listing 6: Microbenchmark pseudo-code for recovering which PC bits are used in the PHT look-up process.

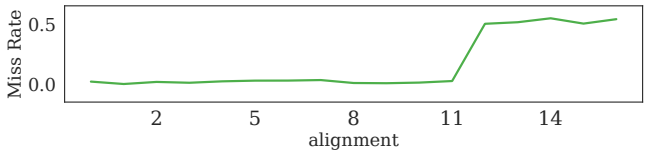


Figure 5: Recovering PC bits used in PHT look-ups.

bits of the PC are used in the PHT look-up process. We then build on this to recover the precise structure of the PHTs.

Recovering the PC Input Bits In order to determine how many bits of the PC are involved in the index and tag hash functions, we intentionally force *aliasing* between two branches. Aliasing occurs when the CBP provides the prediction for two (or more) branches with a single entry from the PHTs or the base predictor – more specifically, when the indexing function (and tag) for the two branches are indistinguishable from each other. Our ability (or not) to force aliasing, then, tells us which bits are used in the index or tag.

For two branches to alias, the PC bits used in the index or tag should be the same. To recover which bits are used, we use two test branches that are aligned but whose directions are mutually exclusive at each iteration, as Listing 6 shows. When the first branch is taken, the second branch is not taken, and vice versa. To ensure the PHR does not impact our measurements, for both branches we set the PHRs to be the same: all zeros except for the most significant bit, equal to k, which is the direction of the first branch.

Since these branches are correlated, the CBP should capture the correlation between them by means of the PHTs *unless* they alias. In other words, the misprediction rate should increase as we increase the alignment. Figure 5 shows our results: when both branches have an alignment of up to 11, the miss rate is low, but when it goes up to 12, the miss rate increases to 0.5 indicating aliasing between the branches.

> **Observation 4.** The 12 lower bits of the PC are used as input to the index and/or tag hash functions. Other microbenchmarks confirmed that all of these 12 lower bits of the PC are involved in either the index or tag hash functions.

Recovering the Associativity Pattern history tables contain tagged entries and are roughly organized as a cache. To recover the structure of the PHTs, then, we need to

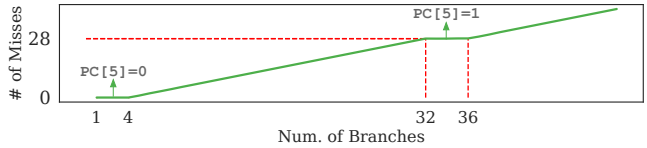


Figure 6: Associativity of the last PHT. The predictor is able to predict 4 unique PCs with PC[5]=0 and 4 unique PCs with PC[5]=1.

recover the associativity – the number of entries that can share the same index – of the different tables. We do this by determining how many ⟨PHR, PC⟩ combinations can be predicted for the same PHRs but similar (i.e., not the same) PCs.* For this experiment, we need the branches (i.e., the ⟨PHR, PC⟩) to lead to the same index and different tags, to ultimately create conflict misses. This is difficult without knowing the index function and the tag function. But as we see in the next section, knowing the associativity is key to finding the index function, so we tackle the associativity first.

We start with the last PHT – the table that uses the longest history length. We target this PHT by placing a single correlating bit in the highest bit of the PHR. Listing 7 shows how we find its associativity. Here, we use a variable number of branches with the same PHR but different PC. We increase the number of branches and measure the misprediction rate. In this experiment (with various permutations, starting addresses, etc.), we find that we largely cannot capture more than 4 unique PCs, i.e., the associativity is at most 4.

These results are robust even when we vary the branch addresses. This suggests that the PC is largely not used in the PHT index function.

The one exception is bit `PC[5]` on Alder Lake/Skylake. If we spread the branch addresses so that `PC[5]` is both 0 and 1 for different branches, we can predict as many as 8 branches accurately. In other words, changing `PC[5]` causes the index function to select a different set. We can see an instance of this in Figure 6, where we get conflict misses with more than 4 unique PCs, but as soon as we advance far enough in the code so that `PC[5]` flips, we then can capture 4 more branches accurately. We observe this same behavior for Cascade Lake and the Ivy Bridge microarchitecture (though for Ivy Bridge, `PC[4]` is used as part of the PHT index rather than `PC[5]`).

We recover the other PHTs similarly. Specifically, we run versions of this microbenchmark, moving the one bit of correlation to different positions in the PHR. That is, we change the position of k in `PHR = k0...0` from `PHR[185]` to `PHR[0]` and measure how many branches can be correctly predicted (the associativity) for each case, keeping `PC[5]=0`.

Figure 7 shows our measurements. We have an associativity of 4 when using history bits `PHR[185:58]`, 8 when using `PHR[57:22]`, and 12 when using `PHR[21:0]`. These results reveal quite a bit about the underlying tables. Higher tables in the TAGE predictor use longer histories (more

*If they have the *same* lower bits, they would just alias and use the same entry.

```
for (i=0; i < NUM_TRIES; i++){
  k = rand();
  SET_PHR(k0..0);    //PHR = k0...0
  SET_ADDRESS(1);    //PC[11:0] = 0
  if(k);             //Test Branch 1
   ...
  SET_PHR(k0..0);    //PHR = k0...0
  SET_ADDRESS(n);    //PC[11:0] = n-1
  if(k);             //Test Branch n
}
```

Listing 7: Microbenchmark pseudo-code for recovering the associativity of the last PHT.

of the PHR), overlapping the histories captured by smaller tables. The size of the steps in the graph suggest that there are three PHTs, each of which is 4-way set associative. When using the lower bits of the global history, all three tables can each hold four distinct PCs. But when using bits 22 and higher, the lowest table is of no use, and only the eight entries in tables 2 and 3 are useful. When using bit 58 or higher, only the associativity in the highest table is of use.

> **Observation 5.** There are 3 PHTs in the CBP, each of which is a 4-way set associative table. Only one bit of the PC is used to index the PHTs (`PC[5]` on Alder Lake, Tiger Lake, Ice Lake, Skylake and Cascade Lake, `PC[4]` on Haswell and Ivy Bridge).

Recovering the Index Hash Functions The last piece of our PHT puzzle is the index functions of each PHT. We assume that index functions use folded instances of the global history (much like the PHR update function). We start with the index function of the last PHT, which uses the longest history length. Assume that bit number `n` and bit number `m` of the global history are combined together to make a single bit in the index hash function of the last PHT ($58 \leq n, m \leq 185$). Therefore, if there is a conflict in the PHT when both bit `n` and bit `m` of the PHR differ, then these two branch/history pairs are mapped to the same set in the last PHT, and thus the two bits are folded together in the index function. This happens when the index function cannot distinguish between `n=0,m=1` and `n=1,m=0`, resulting in the same index and thus conflict misses.

Similar to Listing 7, we design an experiment which includes two groups of branches. The first group has four branches with PHR = `0k0...0` (position of `k` is fixed and equal to 184 for the single experiment shown) and the second group is comprised of four additional branches with PHR = `0...0k0...0` (position of `k` is variable in the experiment and varies from 185 to 0). With no conflicts (different index), we should be able to predict both groups of branches after the first few iterations since our tables are 4-way associative. However, if the branches in the first and the second group both map to the same set in PHT (same index), we will observe an eviction (by measuring a loss in accuracy). Figure 8 shows the number of correctly predicted branches, depending on the position of the `k` bit in the second group of branches. It is clear that for certain `k` the number of correctly predicted branches drops from 8 to 4, indicating that these bits are combined together (in this case, with bit 184) and map to the same set in the last PHT.

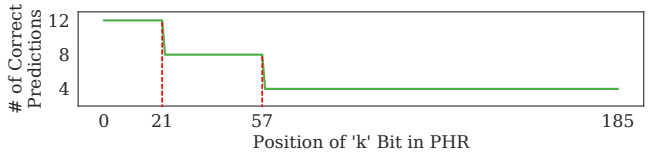


Figure 7: Associativity of all PHTs. We have an associativity of 4 when using history bits PHR[185:58], 8 when using PHR[57:22], and 12 when using PHR[21:0].

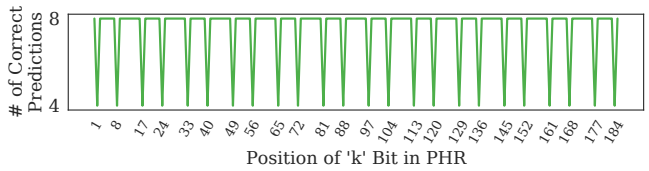


Figure 8: PHR bits that are folded into the index.

We observe that bits $8, 24, 40, \ldots, 168, 184$ from the even PHR bits and $1, 17, 33, \ldots, 161, 177$ from the odd PHR bits form an arithmetic series with the constant difference of 16. This indicates that the index function uses a straightforward, regular folding function (i.e., sequences of sixteen bits are repeatedly folded with other sequences of sixteen bits, with even and odd bits handled differently). Building on this, we exhaustively consider all pairs of bit positions to recover the exact hash function used to index each PHT. A generalized formulation of index functions is presented below.

Hint: `PHR[16i+8,16i-6,2]` means `PHR[start:end:step]`.

- **PHT #3:** `Index[7:0] = PHR[16i+8,16i-6,2]` $\oplus$ `PHR[16j+1,16j-13,2] : i ∈ [11,1] , j ∈ [11,0]`
- **PHT #2:** `Index[7:0] = PHR[16i+8,16i-6,2]` $\oplus$ `PHR[16j+1,16j-13,2] : i ∈ [3,1] , j ∈ [3,0]`
- **PHT #1:** `Index[7:0] = PHR[20,6,2]` $\oplus$ `PHR[15,1,2]`

From prior experiments, we know that `PC[5]` is used in the index function too. To understand how this bit is used, we perform similar experiments (which create deterministic conflicts). These experiments indicate that each PHT index function is index by 9 bits in total: the first 8 bits come from the folded global history, the ninth bit is `PC[5]`, i.e., for all PHTs `Index[8] = PC[5]` (`PC[4]` on Haswell/Ivy Bridge).

> **Observation 6.** Each PHT is indexed by a 9-bit index function, using eight bits derived from global history, and a single bit of the PC which is not combined with any other bits. This same bit is also used (without being combined) for the base predictor.

Implication: Partitioned Branch Predictor From a security standpoint, this final observation has enormous implications. It means that no branch for which `PC[5]` is `0` can possibly be influenced by branches for which `PC[5]` is `1`, in any of the PHTs. They cannot cause evictions to reduce branch accuracy. They cannot detect evictions, eliminating side channels. They cannot mistrain branches because they cannot induce aliasing across this partition.

Table 4: PHTs in Intel microarchitectures.

Microarchitecture	Num. of PHTs	Index Length	Size of Each PHT
Alder Lake/Tiger Lake/Ice Lake Cascade Lake/ Skylake	3	$9 = 8 + \texttt{PC[5]}$	$2048 = 2^9 \times 4$
Haswell/Ivy Bridge	3	$9 = 8 + \texttt{PC[4]}$	$2048 = 2^9 \times 4$

To ensure that the Base Predictor does not invalidate our assumption – that we can partition the entire branch predictor – we perform a last set of experiments. These experiments are very straightforward since the base predictor is simple and described in Appendix A.3. We find that the base predictor is indexed directly by 13 lower bits of the branch address (`PC[12:0]`). Thus, *every* table in the CBP is fully partitioned by `PC[5]` (`PC[4]` in Haswell/Ivy Bridge).

Table 4 summaries the PHTs for different Intel microarchitectures. All tables are 4-way set associative with 9 bits of index (8 bits from folded PHR plus 1 bit from PC). Note that Half&Half works on the primary server-level CPUs produced by Intel spanning more than a decade, during which time Intel has averaged over 95% of the server/datacenter CPU market share [68]. In the next section we use these microarchitectural details to build more secure systems.

4. Partitioning the CBP with Half&Half

This section introduces our CBP partitioning mechanism, dubbed Half&Half, a fast, software-only defense against branch-based side channel and mistraining attacks that is enabled by the in-depth knowledge we gained from our reverse-engineering analysis.

4.1. Assumptions and Threat Model

Our goal is to isolate the conditional branch behavior of two mutually distrusting code components that are executed on a single CPU. One or both of these code components may be malicious, and seek to infer or affect the control-flow of the other component by creating collisions (either aliasing or conflicts) in the entries of the conditional branch predictor. The two code components may be run on the same physical processor core – either as co-resident SMT threads, or time-separated execution flows. We additionally make no assumptions about how the two components are separated at the process level — they may be run in separate OS processes, or in separate security domains (such as userspace and kernel code), or even in a single process sandboxed by WebAssembly — a technique to sandbox untrusted code components within a single process. We assume that the test machine employs mitigations against other attacks on the branch predictor, such as attacks that target the branch target buffer [47] or the return stack buffer [48] and consider these out of scope. We also consider microarchitectural attacks that exploit the memory subsystem (e.g., Meltdown [54], MDS [93], [18], and LVI [92]) out of scope for this paper. Finally, since the two components are isolated and do not communicate, we consider the in-place Spectre-PHT attack [47], [46], [21] (an attack which requires the malicious component to directly invoke the other component with carefully chosen inputs in order to mistrain branches) out of scope, because it does not exploit contention or aliasing.

Our defense partitions the CBP of different code components by carefully picking alignments of branches in the binaries of these components. This assumes that we can control the location of branches in the compiled binary versions of the code components (which is possible either with a small modification during compilation or by directly modifying binaries). Importantly, we do not require any changes to existing source code or existing hardware.

4.2. Overview

As discussed in Section 3, the base predictor is indexed using the 13 lower bits of the branch address while the PHTs are accessed via a 9-bit index, of which 8 bits come from the folded path history register (PHR) and 1 bit comes from the PC – `PC[5]` (the sixth bit of the PC) for Alder Lake, Tiger Lake, Ice Lake, Cascade Lake, and Skylake and `PC[4]` for Haswell and Ivy Bridge. Using this information, we partition the pattern history tables (PHTs) and the base predictor by partitioning their index function, which is done most easily by exploiting the PC bit. That is, by forcing every branch of a thread to be at an address with `PC[5]==0` on (Alder Lake-Skylake), that thread has access to exactly half the entries in each of the four tables, and will share zero entries with another thread where all branches have `PC[5]==1`. Note that because this is a symmetric division of the address space, we can even compile a single binary, and only at load time decide which of the two regions will be used, simply by varying the start address.

With two isolated CBP partitions, we can secure a large class of possible applications, including isolation of user code from kernel code (described in appendix A.5), isolation of co-resident SMT threads from each other (in the most common case of 2 hardware contexts), and isolation of the trusted application code from any untrusted code sandboxed by WebAssembly. We can, in theory, extend this to more than two partitions, however doing so would impose very significant performance penalties due to the additional restrictions on branch placement, since we now have to control bits of the PHR to do so.

Support for two partitions is particularly powerful in the SMT (hyperthreading) case, as these Intel processors all have a thread limit of two. In an SMT processor, the BPU tables and the BTB are typically shared [88]. When paired with Intel's STIBP [39], which isolates the BTB but does nothing for the shared CBP, we can for the first time provide a complete solution for control flow isolation between co-executing SMT threads on the same core.

4.3. Implementation

In order to partition the CBP into two isolated domains, we need to adjust the addresses of all conditional branches of each program such that their `PC[5]` bit (`PC[4]` on some microarchitectures) is constant. We automate this process by modifying two existing compilers to compile code such that it meets our partitioning requirements. First, we implemented our partitioning scheme in LLVM to demonstrate how we can compile existing C/C++ code

without modification. Next, we implemented our scheme in Swivel [62]—a WebAssembly compiler that sandboxes untrusted code to prevent memory safety attacks (buffer overflows, user-after-frees etc), as well as Spectre style attacks; in particular, we replaced some of Swivel's slower mitigations that prevent Spectre-PHT attacks with our light-weight CBP partitioning scheme. Pairing Half&Half with Swivel-SFI (BTB, RSB) results in a comprehensive Spectre solution that is much faster than anything prior.

Implementing the partitioning scheme In order to implement the address adjustment modifications on top of these compilers, we added code to adjust the addresses of conditional branches before emitting binary. This can be done by inserting sufficient NOP instructions to meet our alignment requirements. However, this naive approach has higher overheads than necessary; for example, a single branch may require as many as 32 NOP instructions for suitable alignment[†]. To address this issue, we apply three simple optimizations in our compiler modifications:

- **Using Multi-Byte NOPs:** The x86 architecture allows construction of NOP instructions of different lengths [41]. Multi-byte NOPs have the advantage of requiring less instruction decoding bandwidth and being quicker to translate than a sequence of one byte NOPs. We take advantage of this support, to use NOP instructions as large as 15 bytes.
- **Jumping over NOPs:** We observed that it was simply more efficient to jump over the sequence of NOP bytes for sequences of NOPs larger than 15 bytes. This is possible as the NOP instructions are only present to align conditional branch instructions and do not actually have to be run[‡].
- **Alignment-Invariant NOP Motion:** We found that inserting the required NOPs right before the branch instruction exacerbates the execution time overhead since they are often placed inside backward loops. We instead inject the NOPs right after the previous branch instruction. By doing this, we are less likely to fill inner loops with NOPs, reducing lost performance due to fetch and decode overhead.

Figure 9 shows an example code alongside its translation to domain A. For domain A, we need to ensure that the fifth bit of the address of all conditional branches is set to zero, i.e., `PC[5]=0`. Therefore, in the example code, the second and the third branch need to be adjusted. The second branch is 23 bytes from the next boundary. Since 23 is more than the threshold (16 bytes), the compiler inserts an unconditional branch before the NOPs to jump over them. Note that the unconditional jump is a 2-byte instruction and the compiler only inserts 23-2=21 bytes of NOPs (a 15-byte NOP and a 6-byte NOP) before the conditional branch to set `PC[5]` to zero. In this example, after inserting NOPs for the second branch, the third branch does not require any extra NOPs as its new address already satisfies our goal (`PC[5]=0`).

[†]NOP instructions do not use ALU resources, but still require CPU resources to fetch, decode, and convert to micro-ops. Thus, use of NOP instructions should be minimized where possible.

[‡]The extra jump instruction added is an unconditional and therefore does not lookup or update the CBP tables.

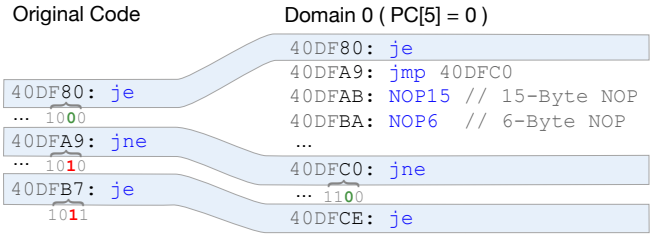


Figure 9: Translating regular code to an isolated partition of the CBP (domain A).

Architecture independent partitioning In order to make Half&Half compatible with various microarchitectures, we designed a new configuration where we partition both `PC[4]` and `PC[5]` bits. In this case, Domain A is where both bits are 0, and Domain B has both bits 1. Because each architecture only partitions on one of the bits, each domain still has access to half the predictor (despite using 1/4 of the address space), but the required NOPs for padding is increased. With this configuration, the partitioning mechanism will work across different processors and it will not need processor-based compilation.

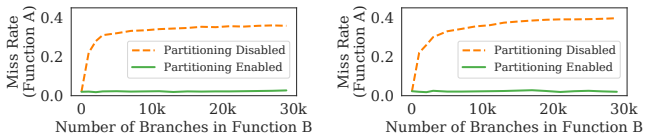
5. Evaluation

This section evaluates our approach of partitioning the CBP (Half&Half) into two isolated domains to prevent branch based attacks. We start with a security evaluation of Half&Half, followed by a performance evaluation.

5.1. Security Evaluation

To investigate the security of Half&Half, we first show that the branch prediction of two partitioned CBP domains, domain "A" and domain "B", are isolated, i.e., branch prediction of code running in domain "A" is unaffected by code running in domain "B". After this, we demonstrate Half&Half prevents the proof-of-concept Spectre-PHT attacks from Google's SafeSide suite.

Our experiment to investigate the effectiveness of Half&Half's CBP partitioning scheme consists of two functions—Function A and Function B, which are run in two partitioned domains of the CBP. Function A is considered benign, and runs a sequence of 1024 conditional branches. Function B is considered malicious, and aims to change the branch prediction accuracy of Function A by running a sequence of N branches, where N is configurable. Branches' directions within a function are correlated and correlation distances are widely varied. CBP will therefore utilize all PHTs together with the base predictor in order to maximize its prediction accuracy. As we have partitioned the CBP, we expect that the branch prediction rate of Function A is the same whether it is executed in isolation or it is executed alongside Function B (for any value of N). In our experiment, we evaluate this statement in both sequential and parallel settings: we first run Function A and Function B interleaved in a single thread; next, we run Function A and Function B in parallel on one CPU core.



(a) Single-Thread **(b) Multi-Thread**

Figure 10: Misprediction rate of Function "A" (Y-axis), when Function "B" is run on either the same or a concurrent thread on a single CPU core. The X-axis varies the number of branches (N) executed in Function B.

Sequential setting To collect our baseline numbers, we first run Function A in a loop 100k times to measure the branch prediction accuracy of this function in isolation. We then repeat this procedure, but this time we follow each call to Function A with a call to Function B, configured with N=1 branch. This results in 100k calls to Function A interleaved with 100k calls to Function B, for which we once again measure the branch prediction accuracy of Function A. We then repeat this procedure for different values of N (the number of branches in Function B), up to N=30000. The results are shown in Figure 10a which graphs the misprediction rate of Function A, for different values of N. We see that the misprediction rate of Function A when partitioned is completely flat for any value of N, i.e., it is completely isolated from the control flow behavior of Function B. Additionally, when we repeat the experiment without partitioning, we see in Figure 10a that Function A's branch prediction behavior is significantly affected by Function B's activity, causing an order of magnitude more misses.

Parallel setting To ensure Half&Half is secure even in parallel (SMT) settings, we repeat the prior experiment with two threads—Thread A and Thread B, both pinned to a single core. Specifically, we pin these threads to different virtual or hyperthreaded cores that map to a single physical core to ensure that the threads share a branch prediction unit. In the experiment, Thread A runs Function A in parallel with Thread B running Function B, and we show the results in Figure 10b. Similar to the prior experiment, we observe that the misprediction rate of Function A is independent of Function B and the value of N when partitioning is enabled.

Testing the out-of-place Spectre-PHT attack To examine the effectiveness of Half&Half against the out-of-place Spectre-PHT attack [106], [8], [17], we run the proof-of-concept implementation of this attack from Google's SafeSide suite [8] with and without partitioning. We find that, when partitioning is enabled, an attacker cannot poison the CBP to make mispredictions in the victim code.

5.2. Performance

To evaluate the performance overhead of Half&Half, we modify two existing compilers to output binaries that are restricted to one of two partitions of the CBP. First, we modify LLVM so we can measure the overhead on existing native (C/C++) code. Next, we modify Swivel [62] (a WebAssembly compiler that inserts mitigations for Spectre style attacks) to leverage CBP partitioning for its Spectre mitigations, and measure speedups. We use these compilers to evaluate the performance of standard benchmarks such as SPEC CPU 2006 [34] benchmarks and Sightglass [6]–a benchmark suite which includes cryptographic primitives, mathematical functions, and common programming utilities such as `heapsort`, `strcat`, and `strtok`.

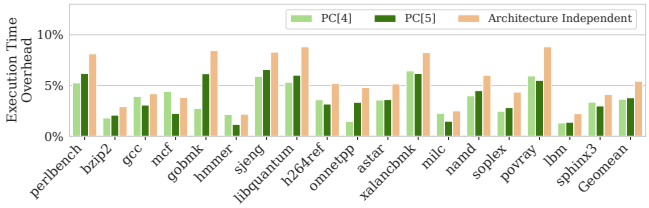


Figure 11: Performance overhead of Half&Half (LLVM) on SPEC 2006 benchmarks.

Benchmark setup To measure the performance overhead of native code, all benchmarks are run on three machines: (1) A 16-core, 24-thread Alder Lake CPU (3.20-5.20GHz) running Ubuntu 22.04 LTS with the generic Linux kernel version 6.0.9-060009, (2) a 4-core, 8-thread Skylake CPU (2.60-3.50GHz) running Ubuntu 20.04 LTS with the generic Linux kernel version 4.4.0-210, and (3) a 4-core, 8-thread Ivy Bridge CPU (3.20-3.60 GHz) running Ubuntu 20.04 LTS with the generic Linux kernel version 5.16.10. To measure the speedup of WebAssembly spectre mitigations, we run the corresponding benchmarks on machines 2 and 3. We run our experiments on these three microarchitectures as the partitioning approach differs slightly across them (See Section 4.2 for details). Unless otherwise specified, all benchmarks are pinned to a single (physical) CPU core. All benchmarks used in this section are compiled with statically linked libraries, in order to ensure that libraries are also partitioned.

Overhead of partitioning native code To measure the impact of partitioning on native code, we use our modified LLVM compiler to compile programs from SPEC 2006 in three settings: (1) using the full CBP, (2) using only the first domain of the CBP (Domain A), and (3) using only the second domain of the CBP (Domain B). We show the performance of the programs in Figure 11 for three configurations including the Alder Lake/Skylake (Only `PC[5]`), Ivy Bridge (Only `PC[4]`), and architecture independent (`PC[5:4]`). In all cases, the results are the average of running in Domain A and B (generally, the performance varies little between the two).

From our experiments, we highlight two observations. First, we see that partitioning programs to one domain has low overhead, imposing an overhead of 1.4%-6.8%, 1.2%-6.6%, and 2.2%-8.8% in the Alder Lake/Skylake, Ivy Bridge, and Portable architectures respectively. Second, as expected, we see marginally lower overheads on Ivy Bridge as this microarchitecture requires less padding with NOP instructions to partition the CBP.

The overhead of CBP primarily comes from two sources. (1) Reduced branch prediction accuracy due to only being

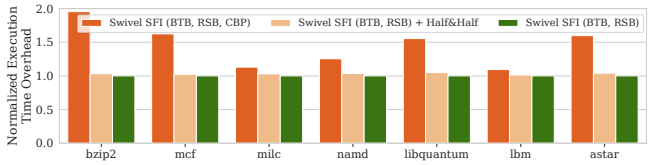


Figure 12: Performance overhead of Half&Half (Swivel) on SPEC 2006 benchmarks.

able to access half the predictor, and (2) the cost of extra instructions (NOPs) in the instruction stream (which can include both pipeline and Icache costs). We measure an overall 2.8% increase in branch misses in partitioned mode, with *gobmk* and *sjeng* seeing the highest increases in misses/instruction.

The average increase in dynamic instructions due to NOPs is 9%, varies from 1% to 16%, and is highest in *xalancbmk*, *bzip2*, *mcf*, and *gobmk*. Overall performance cost does not fully scale with the expansion in instructions, however, since NOPs do not execute and do not have dependencies. It will hurt applications that are bottlenecked in the front-end (instruction delivery), and have little or no impact on applications bottlenecked in the back end (instruction dependencies) such as memory-intensive *mcf*.

Speedup of WebAssembly Spectre mitigations We use the modified Swivel compiler to compile Wasm versions of the Sightglass benchmarks and the SPEC CPU 2006 benchmarks[§]. Wasm versions of SPEC and Sightglass benchmarks are generated by compiling the C/C++ code of these benchmarks using Clang's Wasm backend [7] to produce Wasm binaries. These Wasm binaries are then compiled using Swivel. We note that our modified Swivel compiler retains Swivel's BTB and RSB defenses, but uses Half&Half to defend the CBP (in place of Swivel's default CBP defenses such as eliminating direct branches) as discussed in Section 4.3. We compare the performance of Half&Half paired with Swivel-SFI (BTB, RSB) with Swivel-SFI (BTB, RSB, CBP) and show the results in Figure 12 and Figure 15 (See Appendix A.4) for the SPEC and Sightglass benchmarks respectively. For each bar, the result is normalized to Swivel-SFI with no CBP defense, but all other defenses active (BTB, RSB). For Sightglass, we see that Half&Half based defense allows Swivel to reduce its conditional branch isolation overhead to 3.9% as compared to 55.2% for Swivel-SFI's conditional branch solution. Similarly for SPEC, the overhead reduces to (1.4%-4.9%) as compared to (9.4%-95.4%) for Swivel-SFI. This shows that Half&Half is a viable defense for preventing Spectre attacks in WebAssembly settings.

Partitioning the CBP across SMT threads As discussed in Section 4.2, Half&Half allows the OS to automatically isolate the CBP of two OS processes/threads that are run on one physical core (as SMT contexts, or hyperthreads). In this scenario, the OS scheduler ensures these two threads are always assigned different CBP partitions.

[§]Wasm only supports a subset of the SPEC 2006 benchmarks [62], so we restrict our benchmarks to the supported programs.

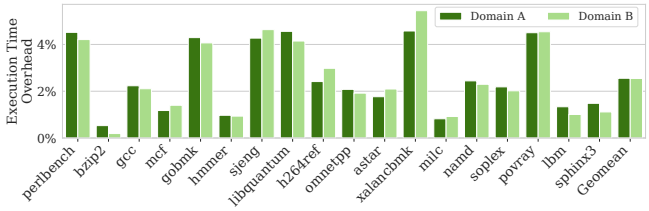


Figure 13: Performance overhead of Half&Half (LLVM) on SPEC 2006 with SMT enabled.

We benchmark this exact scenario, and measure the execution overhead of Half&Half on multithreaded execution. Concretely, we create two threads—Thread A and Thread B—pinned to a single core and configured to use Domain A and B, respectively. We run all permutations of 2 out of 18 SPEC benchmarks in Domain A and B (if B finishes first, it restarts until A finishes). We measure the average execution time of A, both with conventional execution, and with A and B isolated. Half&Half overhead, then, is the relative increase in execution time with partitioning. We show the results in Figure 13. We see that partitioning the CBP for hyperthreads only adds an overhead of 0.5%-5.4%.

The two overheads of CBP Partitioning should have opposite (relative) effects on SMT execution. The loss in branch prediction accuracy is significantly lowered, as SMT threads must share the CBP anyway (but they share it dynamically instead of statically). However, SMT puts more pressure on the front-end of the pipeline to deliver instructions at a high rate, so some configurations will be more sensitive to the NOP expansion. Overall, we see a small lowering in the cost to use CBP partitioning for SMT (compared to the single-thread Skylake overhead in Figure 11). These results demonstrate that our approach can be used to partition the branch predictors of existing SMT processors with only minor performance overheads.

6. Conclusion

This paper presents the first comprehensive analysis of the structure of the conditional branch predictors in modern Intel CPUs. This analysis uncovers a unique opportunity to partition the predictor into two parts based on the branch address. The resulting technique, Half&Half, enables the system to provide complete CBP isolation between two domains, disabling CBP side channel and mistraining attacks. The technique works on current systems with no hardware changes. Performance overhead of Half&Half is at most 4.9%, dramatically lower than the prior state-of-the-art software solution. The ability to partition the CBP into two mutually separated regions enables opportunities such as user-kernel isolation, isolation of co-resident SMT threads, and separation of sandboxes from the calling process.

Acknowledgments

Thanks to Craig Disselkoen and Evan Johnson for their insightful discussions, and the anonymous reviewers for their helpful suggestions. This work was supported, in part, by NSF/Intel Foundational Microarchitecture Research Grant CCF-1823444, as well as gifts from Intel.

References

[1] The 2nd jilp championship branch prediction competition (cbp-2). http://www.jilp.org/cbp2006, 2006. [Online].

[2] The 3rd jilp championship branch prediction competition (cbp-3). http://www.jilp.org/cbp2011, 2011. [Online].

[3] The 4th jilp championship branch prediction competition (cbp-4). http://www.jilp.org/cbp2014, 2014. [Online].

[4] The 5th jilp championship branch prediction competition (cbp-5). http://www.jilp.org/cbp2016, 2016. [Online].

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

[6] Bytecode Alliance. sightglass: a benchmark suite and tool to compare different implementations of the same primitives. https://github.com/bytecodealliance/sightglass, 2019. [Online].

[7] Dan Gohman. WASI: The WebAssembly system interface. online:. https://wasi.dev/, 2019. [Online].

[8] Google. https://github.com/google/safeside, 2019. [Online].

[9] 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. [Online].

[10] 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.

[11] 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.

[12] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *RSA Conference*, pages 225–242, 2007.

[13] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture (ISCA)*, pages 132–144, 2020.

[14] 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)*, pages 151–164, 2019.

[15] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[16] 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.

[17] 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)*, pages 249–266, 2019.

[18] Canella, Claudio and Genkin, Daniel and Giner, Lukas and Gruss, Daniel and Lipp, Moritz and Minkin, Marina and Moghimi, Daniel and Piessens, Frank and Schwarz, Michael and Sunar, Berk and Van Bulck, Jo, and Yarom, Yuval. Fallout: Leaking data on meltdown-resistant cpus. In *Computer and Communications Security (CCS)*, 2019.

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

[20] 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)*, pages 69–76, 2017.

[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 *European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019.

[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)*, pages 529–536. IEEE, 2020.

[23] Jonathan Corbet. Taming STIBP. https://lwn.net/Articles/773118/, 2018. [Online].

[24] Jan de Mooij. Enable index masking by default. https://bugzilla.mozilla.org/show_bug.cgi?id=1435266, 2018. [Online].

[25] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In *Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, 2015.

[26] 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.

[27] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[28] 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.

[29] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *Design Automation Conference (DAC)*, pages 1–6, 2019.

[30] 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.

[31] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2020.

[32] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883, 2021.

[33] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[34] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[35] Jann Horn et al. Reading privileged memory with a side-channel. *Project Zero*, 39, 2018.

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

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

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

[39] Intel. Single thread indirect branch predictors. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html, 2018. [Online].

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

[41] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2011.

[42] Daniel A Jiménez. Fast path-based neural branch prediction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 243–252, 2003.

[43] Daniel A Jiménez. Piecewise linear branch prediction. In *International Symposium on Computer Architecture (ISCA)*, pages 382–393, 2005.

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

[45] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверяй, но проверяй: Sfi safety for native-compiled wasm. In *Network and Distributed Systems Security (NDSS) Symposium*, 2021.

[46] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[47] 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)*, pages 1–19, 2019.

[48] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Security Symposium (USENIX Security)*, 2018.

[49] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *IEEE Symposium on Security and Privacy (SP)*, pages 39–53, 2020.

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

[51] 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.

[52] Haifeng Li, Tianyue Lu, Yuhang Liu, and Mingyu Chen. Make page coloring more efficient on slice-based three-level cache. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 310–317. IEEE, 2019.

[53] 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)*, pages 264–276, 2019.

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

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

[56] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

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

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

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

[60] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 395–404, 2012.

[61] Ravi Nair. Dynamic path-based branch correlation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–23, 1995.

[62] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In *USENIX Security Symposium (USENIX Security)*, pages 1433–1450, 2021.

[63] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[64] 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].

[65] 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.

[66] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security Symposium (USENIX Security)*, pages 1481–1498, 2020.

[67] 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.

[68] Passmark. Amd vs intel market share. https://www.cpubenchmark.net/market_share.html, 2022. [Online].

[69] 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.

[70] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An" undo" approach to safe speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 73–86, 2019.

[71] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *arXiv preprint arXiv:1905.09100*, 2019.

[72] 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*, pages 279–299. Springer, 2019.

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

[74] André Seznec. Analysis of the o-geometric history length branch predictor. In *International Symposium on Computer Architecture (ISCA)*, pages 394–405, 2005.

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

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

[77] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

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

[79] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. *International Symposium on Computer Architecture (ISCA)*, pages 295–306, 2002.

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

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

[82] Navin Shenoy. Latest intel security news: Updated firmware available for 6th, 7th and 8th generation intel core processors, intel xeon scalable processors and more. https://newsroom.intel.com/news/latest-intel-security-news-updated-firmware-available/, 2018. [Online].

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

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

[85] 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.

[86] 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)*, pages 395–410, 2019.

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

[88] 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.

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

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

[91] 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.

[92] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[93] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[94] 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.

[95] 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)*, pages 572–586, 2019.

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

[97] 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)*, pages 428–441, 2018.

[98] 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.

[99] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014.

[100] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[101] 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.

[102] 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.

[103] 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)*, pages 707–720, 2020.

[104] 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)*, pages 954–968, 2019.

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

[106] 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)*, pages 667–682, 2020.

[107] 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)*, pages 1267–1272, 2021.

[108] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1138–1152, 2020.

Appendix A. Additional Results, and Discussion

A.1. PHR Update Policy: Earlier Intel Processors

Figure 14 shows the PHR update policy for the Haswell and Ivy Bridge microarchitectures. Both microarchitectures have the same PHR update policy, and `PC[4]` directly provides one bit of the index to PHTs; thus it partitions the tables.

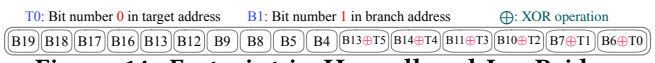


Figure 14: *Footprint* in Haswell and Ivy Bridge.

A.2. Branch Instructions: Influencing the CBP

As discussed in section 3.3, the global history register (PHR) can be influenced by unconditional branch instructions despite the fact that they do not use predictions from the CBP. Therefore, for completeness, we examine all possible instructions that could influence the state of CBP, such as updating the PHR and PHT entries. Our results show that all conditional branch instructions and loop instructions update the PHR (when taken) and use the predictions from the CBP. Additionally, all unconditional branches (including indirect branches) update the PHR but do not affect the PHTs. Table 5 shows the list of all possible instructions that could influence the PHR or PHT entries.

Table 5: Instructions influencing the CBP elements

Affected structure	Instructions
Both PHR and PHTs	jz/jnz, je/jne, js/jns, jo/jno, jpe/jpo, jp/jnp, jb/jnb, jae/jnae, jc/jnc, jbe/jnbe ja/jna, jl/jnl, jge/jnge, jle/jnle, jg/jng, loop, loope/loopne, loopz/loopnz, jcxz/jecxz
Only PHR	jmp (relative/absolute/register/memory), call/ret

A.3. Base Predictor

As discussed in Section 2.2, state-of-the-art directional branch predictors in the literature [80], [77], as shown in Figure 1, employ a base predictor ($Table_0$) alongside a set of tagged components (PHTs). The base predictor is a bimodal-like predictor [55] indexed directly by branch address (PC) to record the local history of each branch. A base predictor is crucial to the performance of the CBP in modern processors because: (1) In case of no matching tagged component, CBP uses the prediction provided by base predictor, and (2) Base predictor performs better than the PHTs when a branch is strongly biased in a particular direction since its warm-up time is short. Also, Evtyushkin et al. [28] showed that the CBP of Haswell and Sandy Bridge microarchitectures features a base predictor in charge of providing a basic prediction. In order to recover how many, and which, bits of branch address are used to index base predictor, we coerce the CBP to provide the prediction using base predictor instead of PHTs. We do this using an experiment shown in Listing 8.

- ► We try to alias two test branches in which one of them is always taken and the other one is not-taken.
- ► Prior to each test branch, we clear the PHR (PHR = 0...0) to ensure that the CBP can only access a set of entries within PHTs indexed by zero. Beforehand, we fill those PHT entries with some correlating branches that will cause no matching tagged component while predicting directions of test branches.

Therefore, the CBP provides the prediction for test branches using only the base predictor, allowing us to study its index. By aligning addresses of test branches we try to alias them in the base predictor, thus a single entry provides prediction for both of them. Since the first test branch is always taken and the second one is always not-taken, aliasing will cause mispredictions.

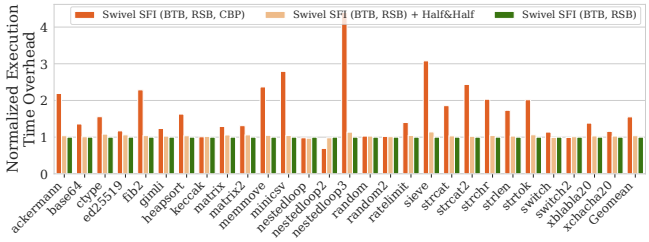


Figure 15: Performance overhead of Half&Half (Swivel) on Sightglass benchmarks.

```
for (i=0; i < NUM_TRIES; i++){
  Fill_PHTS(idx=0); //Fill PHT entries
  SET_PHR(0..0);    //PHR = 0...0
  align(alignment)  //PC[alignment:0] = 0
  if(TRUE);         //Test Branch 1

  SET_PHR(0..0);    //PHR = 0...0
  align(alignment)  //PC[alignment:0] = 0
  if(FALSE);        //Test Branch 2
}
```

Listing 8: Microbenchmark pseudo-code for recovering the index of the base predictor

> **Observation 7.** We found that 13 lower bits of the branch address (`PC[12:0]`) are used to index to the base predictor.

A.4. Speedup of WebAssembly Spectre mitigations

We use the modified Swivel compiler to compile Wasm versions of the Sightglass benchmarks. We compare the performance of this defense with Swivel's two baseline defenses (called Swivel-SFI and Swivel-CET) and show the results in Figure 15 for Sightglass benchmarks. For each bar, the result is normalized to Swivel with no CBP defense, but all other defenses active.

A.5. Partitioning the CBP across Kernel and User programs

As discussed in Section 4.2, Half&Half can be used to partition the CBP between userspace code and kernel code. This prevents malicious users from leveraging out-of-place Spectre-CBP attacks to learn information from kernel code. In the case of arbitrary code, we must either use trusted compilers, trusted binary translators, or verify the binaries to ensure that all conditional branches follow the partitioning policy. A number of prior works demonstrate the potential for verifying binaries [45], [60], [100].

We measure the performance overhead of Half&Half, with the LMbench [56] benchmark suite. We measure performance overheads of LMBench when it is run isolated from the kernel code, and show the results in Figure 16. From the experiments, we see that using Half&Half to isolate Kernel code from userspace code only adds an overhead of 5.9%.

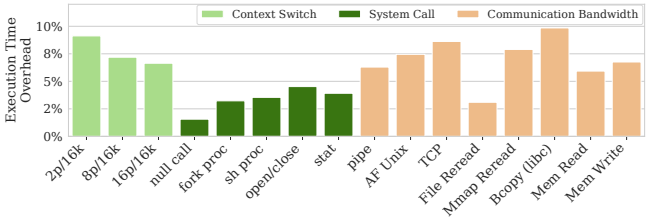


Figure 16: Performance overhead of Half&Half while isolating the LMbench from Kernel code.

A.6. Swivel Protections vs Half&Half Protections

In order to ensure our comparisons with Swivel performance in Section 5.2 are fair, we need to confirm that the compared solutions – specifically Swivel-SFI (PHT, BTB, RSB) vs. Swivel-SFI (BTB, RSB) – provide the same protections. We analyze those protections more deeply here.

According to the Swivel paper, either in-place or out-of-place Spectre can be used by a malicious Wasm instance "A" to:

1) breakout of it's sandbox and directly access memory of Wasm instance "B". (breakout attack)
2) poison the branch predictors so that Wasm instance "B" self-exfiltrates it's own sensitive data. (poisoning attack)

Breakout attacks in Swivel are protected using *linear blocks* and the *BTB flush* which we leave unchanged in our hybrid Swivel + Half&Half implementation.

Poisoning attacks in Swivel are protected with distinct protections: PHT (eliminate direct branches), BTB (flush BTB), and RSB (eliminate returns). In our hybrid Swivel + Half&Half, we only remove Swivel's poisoning PHT protections (i.e., direct branch elimination) in favour of direct branch partitioning which will provide equivalent protections. The table below clarifies our changes to Swivel, and how each protects against the full permutation of attacks.

Attack	Swivel-SFI (PHT,BTB,RSB)	Swivel-SFI (BTB, RSB) + Half&Half
Breakout PHT	LB	LB
Breakout BTB	BTB flush + LB	BTB flush + LB
Breakout RSB	LB + separate control stack + no return inst.	LB + separate control stack + no return inst.
Poisoning PHT	no direct branch	Half&Half direct branch partitioning
Poisoning BTB	BTB flush	BTB flush
Poisoning RSB	separate control stack + no return inst.	separate control stack + no return inst.

A.7. Half&Half Integration with OS Kernels

This section details the process by which the kernel can successfully run an application compiled with Half&Half, targeting SMT execution. Our integration scheme involves the following steps:

First, during the OS Boot setup, each SMT core in a physical core is assigned a unique domain A or B. Depending on the microarchitecture, domain A may have `PC[4/5] = 0`, while domain B may have `PC[4/5] = 1`. This ensures that each core executes only one domain at a time.

When compiling an application, the Half&Half Compiler should, by default, compile all branches with `PC[4/5] = 0` and add a flag to all applications compiled with Half&Half. It should be noted that handwritten assembly codes will require modifications to ensure that all branches within them have appropriate addresses.

When executing an application, the OS checks if it supports Half&Half, and if so, associates the process with one of the domains A or B. If the OS assigns the process domain B, the elf loader offsets the code pages by 16/32 bytes during load.

We believe that this integration scheme only requires minor changes to the existing OS components and will enable the OS to utilize Half&Half's isolation mechanisms between user/kernel and SMT threads.