Exploiting Inaccurate Branch History in Side-Channel Attacks

Yuhui Zhu, *Scuola Superiore Sant'Anna and Scuola IMT Alti Studi Lucca;*
Alessandro Biondi, *Scuola Superiore Sant'Anna*

https://www.usenix.org/conference/usenixsecurity25/presentation/zhu-yuhui

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

Exploiting Inaccurate Branch History in Side-Channel Attacks

Yuhui Zhu[1,2] and Alessandro Biondi[1]

[1]*Scuola Superiore Sant'Anna*
[2]*Scuola IMT Alti Studi Lucca*

Abstract

Modern out-of-order CPUs heavily rely on speculative execution for performance optimization, with branch prediction serving as a cornerstone to minimize stalls and maximize efficiency. Whenever shared branch prediction resources lack proper isolation and sanitization methods, they may originate security vulnerabilities that expose sensitive data across different software contexts.

This paper examines the fundamental components of modern Branch Prediction Units (BPUs) and investigates how resource sharing and contention affect two widely implemented but underdocumented features: *Bias-Free Branch Prediction* and *Branch History Speculation*. Our analysis demonstrates that these BPU features, while designed to enhance speculative execution efficiency through more accurate branch histories, can also introduce significant security risks. We show that these features can inadvertently modify the Branch History Buffer (BHB) update behavior and create new primitives that trigger malicious mis-speculations.

This discovery exposes previously unknown cross-privilege attack surfaces for Branch History Injection (BHI). Based on these findings, we present three novel attack primitives: two Spectre attacks, namely **Spectre-BSE** and **Spectre-BHS**, and a cross-privilege control flow side-channel attack called **BiasScope**. Our research identifies corresponding patterns of vulnerable control flows and demonstrates exploitation on multiple processors. Finally, **Chimera** is presented: an attack demonstrator based on eBPF for a variant of Spectre-BHS that is capable of leaking kernel memory contents at 24,628 bit/s.

1 Introduction

Microarchitectural vulnerabilities pose serious and evolving threats to modern out-of-order CPUs. Research over the past few years has demonstrated how performance-oriented optimizations, designed to maximize pipeline efficiency, can be exploited to create a malicious transient execution environment capable of leaking sensitive data across different security contexts [19, 22, 25, 33, 35, 38, 41, 42, 45, 52, 55, 71]. Some findings have also revealed how micro-architectural side effects can influence the behavior of shared hardware resources, enabling data disclosure attacks in concurrent execution environments [15, 34, 40, 43, 44, 56, 58, 65, 70, 73].

Despite extensive efforts to address these issues at the hardware-software interface, our research reveals that certain underdocumented micro-architectural features, albeit intended to manage resource sharing and contention, can create new attack surfaces when interacting with other micro-architectural behaviors.

Exploitations. This paper examines *history-based branch prediction* and assesses potentially vulnerable behaviors inadvertently implemented in processors. Through extensive empirical analyses of multiple processors, we reveal how micro-architectural handling of resource contention creates distinct vulnerabilities across different processor designs.

Building on our findings, we present a series of novel attack flows that can extract secret data from separate software contexts. First, we present **BiasScope**, a coarse-grained control flow side-channel that exploits the BPU's bias-free behavior to leak branch outcomes. Second, we propose two Spectre-variant attacks, **Branch Status Eviction (Spectre-BSE)**, and **Branch Hisotory Speculation (Spectre-BHS)**. These attacks build upon the concept of *Branch History Injection* (BHI) [8], but achieve malicious manipulation of the Branch History Buffer (BHB) *indirectly* by controlling the branch history updating mechanisms we investigated. Since they avoid explicit branch history injection through adversary-controlled branches, these attacks naturally circumvent existing BHI mitigations on ARM processors.

While the effectiveness of our attacks heavily depends on the structure of victim code, we demonstrate vulnerable code patterns and analyze their relationship to hardware implementation characteristics. Finally, we present **Chimera**, a demonstrator based on eBPF, representative of an *end-to-end attack* with Spectre-BHS, capable of leaking kernel memory at 24,628 bit/s. In the light of recent work [59] that unveiled a large residual attack surface for known Spectre attacks in the Linux kernel, the availability of gadgets that can enable native Spectre-BSE and Spectre-BHS attacks should certainly not be underestimated. Research on *speculative trojans* [72] further exacerbates the risks associated with these attacks.

Contribution. In summary, this paper makes the following contribution:

- We systematically analyze resource sharing and contention in modern branch prediction units, identifying mechanisms that can lead to exploitable behaviors.
- We reveal and evaluate undocumented features in modern BPUs, introducing new techniques for implicit BHB manipulation. We exploit these primitives to present novel side-channel attacks: Spectre-BSE, Spectre-BHS, and BiasScope, enabling both speculative execution attacks and control flow monitoring across privilege boundaries with all Spectre mitigations enabled.
- Through the development of exploitable program patterns and the Chimera end-to-end attack demonstrator based on eBPF, we highlight the importance of systematic analysis in uncovering potential security vulnerabilities in hardware and software designs.

2 Background and Related Work

2.1 Branch Prediction

To minimize branch resolution latency and determine the next fetch address before branch resolution, processors employ a **Branch Prediction Unit (BPU)** that makes educated guesses based on the historical behavior of branches. The BPU primarily predicts two critical properties of a branch: the *target address* (indirect branches) and the *taken/not-taken direction* (conditional branches). To enable predictions, the BPU implements dedicated caches to store learned branch behaviors. The **Branch Target Buffer (BTB)** [10, 11, 13, 30] caches target addresses, enabling early instruction fetch redirection even before branch decode completion. The **Pattern History Table (PHT)** [27, 68, 69] aids in predicting conditional branches by tracking their past outcomes using *saturation counters*.

History-Based Branch Prediction. While early branch predictors relied on simple Program Counter-based indexing to correlate branch addresses with their targets, research has shown this approach to be insufficient since branch outcomes often depend on the control flow context established by preceding branches. Contemporary BPUs leverage this insight by implementing history-based prediction policies that capture correlations among branches.

The majority of modern BPUs introduce **Branch History Buffer (BHB)** [68, 69] to maintain a record of recent branch

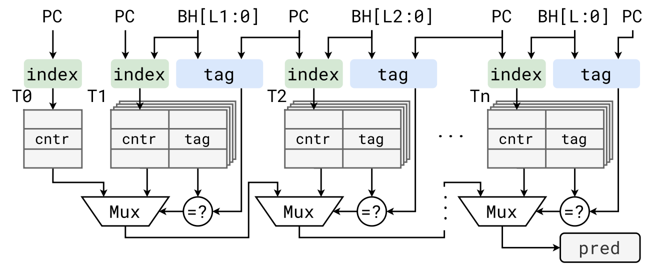


Figure 1: TAGE branch predictor.

outcomes on the execution path, typically as a shift register of taken/not-taken bits, which length is pre-defined. Some implementations employ an enhanced variant of BHB known as the **Path History Register (PHR)** [36, 47]. Unlike the canonical BHB, which only records taken/not-taken outcomes, the PHR maintains a complete jumping path by storing multi-bit footprints that encode both the source and target addresses of each *taken* branch, thus providing more distinctive signatures for different control flow paths. In PHR, the historical information is combined with the branch address to create sophisticated indexing functions for the BTB and PHT [10]. This approach maintains separate prediction entries for each unique control flow context, significantly improving prediction accuracy by capturing path-specific branch behaviors. Recent studies [6, 8, 31, 66, 67] have provided detailed insights into the BHB updating mechanisms in modern processors.

Tags in BPU implementations. *Tags* are unique identifiers in cache memory that verify the presence of requested information. Differently from index functions, which in set-associative caches may map multiple aliasing addresses to the same cache set, tags *enable unique identification* of cached elements within each set. Upon a cache query, tags help filter out aliased entries that share the same index and allows the cache to signal a *query miss* when no matching tag is found.

Since branch prediction only guides speculative execution without affecting the architectural state, tag fields in BTB and PHT can be optional. While many early works [10, 13, 27, 30, 68, 69] omitted tags from their designs based on this rationale, some researchers [24, 54] introduced tags in their full-associative BTB/PHT implementations to prevent aliasing-induced mispredictions.

Tags play a central role in *TAgged GEometric history length* (TAGE), the state-of-the-art branch predictor design [48–50]. As illustrated in Fig. 1, TAGE consists of multiple prediction tables: several tagged tables that use different combinations of the Program Counter (PC) and branch histories with geometrically increasing length (`BH[L1:0]`, `BH[L2:0]`,...) for indexing, and an untagged *base predictor* (`T0`) that relies solely on PC-based indexing. During prediction, all tables are queried in parallel for candidate results. The selection process prioritizes predictions from tables with *longer history lengths*, as these capture more detailed branch correlation patterns.

TAGE leverages tags not just for entry identification but as a fundamental mechanism for prediction selection. When a tag mismatch occurs in a table, TAGE falls back to checking results from tables with shorter history lengths, ultimately defaulting to `T0` if no matches are found in other tables.

2.2 Spectre Attacks

By exploiting speculative execution driven by the BPU, researchers have discovered multiple variants of Spectre attacks [8,9,19,28,29,32,57,62,72] that can transform harmless memory load operations into data disclosure gadgets. When a branch's resolution is delayed due to unresolved data dependencies, the BPU makes predictions based on patterns stored in its prediction caches. Although mispredictions are eventually reversed through pipeline flushes, the speculative execution may perform architecturally unauthorized operations before the flush occurs. This constitutes the basis for Spectre attacks.

Beyond Spectre-v1/v2 attacks, researchers have discovered additional vulnerabilities [57, 64]. *Straight-Line Speculation* (SLS) [5, 61] concerns the speculative execution of instructions immediately following another instruction that should change the control flow (e.g., a branch, a return, etc.). Branch History Injection [8] further exploited the interaction between PC values and BHB content in BTB indexing. By manipulating these components to generate colliding indices, attackers can force speculative execution of specific gadgets.

Canella et al. [9] systematically categorized attack vectors based on privilege levels and execution contexts. Their experiments revealed multiple mis-training vectors: the branch can be mis-trained either ***in-place*** (using the vulnerable branch itself) or ***out-of-place*** (using a branch at a conflicting virtual address), and the mis-training can occur from either the same address space (victim process) or across different address spaces (attacker-controlled process).

Their work revealed that the effectiveness of these attacks varies significantly across platforms due to microarchitectural differences. For instance, while out-of-place Spectre-v2 attacks were demonstrated on Intel processors, experiments on ARM's Cortex-A57 core (tested on Nvidia Jetson TX1) showed resistance to this attack vector.

3 Microarchitectural Details of the BPU

In this paper, we evaluate multiple processors summarized in Table 1.

3.1 Reference Snippet

To present our following experiments, it is convenient to introduce a reference vulnerable code snippet in Listing 1, which utilizes and manipulates history-based prediction.

SoC	μarch	Linux
NXP i.MX8QM	Cortex-A72	5.15.71
BCM2712 (RaspberryPi 5)	Cortex-A76	6.6.63
Nvidia Jetson AGX Orin	Cortex-A78AE	5.10.104
AMD Ryzen 7 7840U	Zen4	6.12.20
Intel N100	Gracemont	6.1.0
Intel Core Ultra 7 155H	Redwood Cove Crestmont	6.8.0

Table 1: SoCs and Linux kernel versions tested in our paper.

```
BH_n:            // BH[n], populate branch history
  Bcond/BLR/BR
  // ......
  Bcond/BLR/BR
Bx_prime:                  // optional
  Bcond/BLR/BR             // replace with padding NOPs
  B Bi_pred
```

```
Bi_pred:
  LDR X1, target           // target=[t_safe or t_leak]
  BR X1

t_leak:                    // alias t_primary:
  LDR X2, [X3]
  LDR X5, [X4, X2]         // refill gadget
  RET

t_safe:                    // alias t_alt:
  ADD X2, X3, X4           // example benign addition
  RET
```

Listing 1: Pseudocode of the reference vulnerable snippet.

`BH[n]` is a series of branches (of any type) that *always completely* populate the BHB with a specific value `BHB(BH[n])` [1]. **`Bx_prime`** is an optional indirect or conditional branch that, when executed after `BH[n]`, populates the BHB with an additional footprint.

`Bi_pred` is an indirect branch with two possible targets, designated as `t_safe` and `t_leak`. Both `BH[n]` and `Bx_prime` contribute to its speculation through the combined BHB value `BHB(BH[n]Bx_prime)`. Upon executing `Bi_pred`, speculative selection between these targets can be monitored through micro-architectural probes (e.g., cache hit), enabling inspection of prediction results.

The snippet reports a leakage gadget for target `t_leak` and a benign operation for target `t_safe`. Sometimes, in the following, we just need to distinguish between these two targets, independently of the corresponding code: in these cases, the reader can ignore the leakage and benign instructions and the two targets are referred to as `t_primary` and `t_alt`, respectively.

Note that this is just an example of vulnerable code: in practice, many other snippet structures can be vulnerable to our attacks.

[1]This notation may also be replaced with the actual BHB values, which are denoted with the notation `[<footprint 1>, <footprint 2>, ...]`, e.g., `[A, B, C, D]`.

3.2 Branch History or Path History

We follow the method proposed by Yavarzadeh et al. [67], which focused on Intel processors, to investigate BHB implementations in tested ARM and AMD processors. Our experiments reveal that the BHBs in A76 and A78AE do not record not-taken conditional branches but can distinguish between footprints of taken branches at different addresses. Moreover, all branch types—direct, indirect, and conditional—update a unified BHB. These observations suggest that both models implement a PHR, rather than a canonical BHB. ARM's official documentation [6] recommends BHB population loops of 24 and 32 iterations for A76 and A78AE respectively, with each iteration executing 2 branches, our experiments confirm that their PHRs can store footprints of twice these values: 48 and 64 branches, respectively.

Conversely, our experiments on A72 and Zen4 revealed a completely different implementation. Based on our observations, we conjecture that A72 is designed with 2 separate BHBs, one canonical BHB and one PHR. Both buffers are 8 bits in size. The BHB holds eight 1-bit outcomes for conditional branches, while the PHR holds four 2-bit footprints obtained from the [5:4] bits of the indirect branch target. These two buffers are updated and stored separately, then XOR'd together when read by the BPU. AMD Zen4 also follows this design, but direct and conditional branches also update the PHR. In the following, whenever we do not need to distinguish between these detailed implementations, we will simply use the term BHB.

3.3 BTB/PHT Mis-training & Eviction

While untagged BTB/PHT implementations allow one branch to directly *pollute (or mis-train)* the prediction of another branch due to set-index conflict, however, when a mismatching tag is detected for a committed branch, the tagging mechanism will consider it as an unrecorded one, thus will replace an existing record and resulting in *eviction* of the existing record.

The effects of tagging are also evidenced in recent research. Canella et al.'s evaluation of Spectre-v2 [9] found that only Intel processors are vulnerable to the out-of-place variants, while AMD processors remain unaffected. This observation suggests that AMD CPUs employ more comprehensive tagging strategies compared to Intel's implementations. This is further supported by Wieczorkiewicz's work on SLS [61]. ARM processors were found to be immune to out-of-place Spectre-v2 but vulnerable to SLS for indirect branches [5], suggesting that BTB eviction should also appear on these processors. According to some papers [10, 13, 27], since predictions for conditional branches may also involve BTB records, i.e., depending on branch targets, we further suspect that BTB eviction may also influence the prediction of conditional branches.

```
void test_eviction(bool *flag, char *dc_signal) {
    populate_bh(); // or replace with padding NOPs
    if ( *flag == 0 )
        char junk = *dc_signal;
}
```

Listing 2: Snippet to test BTB/PHT mis-training.

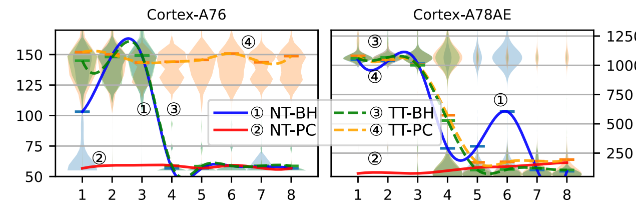


Figure 2: Cache access latency (ns) as a function of the number of mis-train snippets (x-axis) after out-of-place training of victim if-load snippet. Lower latency indicates not-taken speculative execution of the victim branch. Victim branch initially trained to not-taken (NT) or taken (TT), with out-of-place training using matching (BH) or different branch histories (PC). Results averaged over 12,800 tests per configuration.

Mis-training. To verify this conjecture, we design an experiment using a Spectre-v1 snippet shown in Listing 2. We maintain one copy of the snippet as the victim while creating multiple congruent copies at 20-bit-aligned addresses (i.e., keeping lower 20 bits identical to ones in the address of the victim) to serve as mis-train snippets.

The experiment begins with establishing a branch highly biased to *not-taken* by repeatedly executing (e.g., 32 times) the victim snippet with `flag=0` (NT-* in Fig. 2). Following this initialization, out-of-place training attempts to mis-train the branch to *taken* by executing multiple mis-train snippets with `flag=1`. To observe the prediction outcome through data cache signals, the victim snippet runs again with `flag=1` set. We also further evaluate the opposite case in which the victim branch is initially trained as taken (`flag=1`) (TT-* in Fig. 2).

Varying the number of mis-train snippets. We first conducted the test on A76 and A78AE. Results are reported in Fig. 2. Our tests start with no mis-train snippets as a baseline, then increase the number from one to eight. The baseline test without mis-train snippets successfully establishes the biased prediction. When initially training the victim branch as not-taken, if using 1 to 3 mis-train snippets, the cache signal disappears, indicating the branch is successfully mis-trained to taken through out-of-place Spectre-v1 attack (NT-BH in Fig. 2). Interestingly, starting from 4 branches, the cache signal reappears, suggesting an unexpected change in BTB and PHT behavior.Even when initially training the victim branch as taken (`flag=1`), exceeding this threshold of congruent branches causes the prediction to revert to not-taken (TT-BH in the figure). Similar patterns emerge on A78AE, where out-of-place Spectre-v1 attacks only succeed with one to four mis-train snippets. Furthermore, on A72, just two mis-train snippets were sufficient to trigger this inversion phenomenon.

We also performed the same experiment on x86 processors. While we were able to perform regular out-of-place mis-training for conditional branches on all tested processors, similar reverting behavior appeared on AMD Zen4 when executing 16 mis-train snippets containing conditional branches jumping across a 4K-aligned boundary. However, this reverting behavior was not observed on Intel processors, suggesting a different implementation of tagging mechanisms.

This experiment demonstrates that, in tested ARM and AMD processors, while saturation counters can be shared among multiple branches and lead to out-of-place mis-training, exceeding a threshold number of congruent branches accessing the same branch prediction entry triggers an eviction-like behavior for conditional branches. This may not match the statement of ARM claiming only unconditional branches vulnerable to SLS [5]. We tentatively attribute this phenomenon to *BTB/PHR eviction* mechanisms.

Mis-training and eviction with different branch histories were also tested: details are available in Appendix B.

4 Threat Model

We consider a data disclosure threat model where the attacker possesses knowledge of targeted hardware and can identify or inject vulnerable execution patterns in the victim system. The unprivileged or privileged attacker can execute the corresponding vulnerable code snippets. The target system has no software vulnerabilities. All recommended Spectre mitigations are enabled with recommended configurations unless explicitly noted as being disabled for specific experiments.

5 Exploitation 1: Bias-Free Branch Prediction

During our tests on the BHB of A72, we observe that an indirect branch consistently jumping to the same target from program initialization (e.g., a fixed function call in C code compiled as an indirect branch) never updates the path history. This unusual behavior of A72 suggests an undocumented BHB and PHR update mechanism that selectively records branch footprints based on certain conditions. We attribute this behavior to **Bias-Free Branch Prediction** [2, 17], and present primitives exploiting this undocumented behavior.

5.1 Filtering Biased Branches from BHB

In history-based branch prediction, the fixed size of the BHB imposes a limitation on its capacity to store control flow information. To maximize prediction accuracy, the BPU must effectively eliminate irrelevant data from the control flow, ensuring the BHB retains older yet meaningful footprints within the limited storage budget.

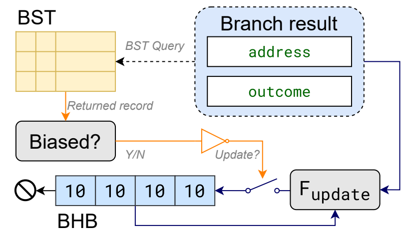


Figure 3: BHB update process in bias-free branch prediction. When a branch resolves, the BPU determines the branch's bias status using the process described in Algorithm 2, excluding biased branches from the updating process.

One significant source of irrelevant information is *biased branches*. These are branches that consistently produce the same outcome throughout a program's execution. A biased *conditional* branch always maintains the same taken or not-taken status, while a biased *indirect* branch consistently jumps to the same target address. Since biased branches neither affect future control flow nor depend on earlier branches, their footprints do not contribute to meaningful history for branch prediction. Instead, they would occupy space in the buffer, reducing its capacity to capture correlations from earlier, more informative branches, thereby degrading prediction accuracy.

The **Bias-Free Branch Predictor** [2, 17] was introduced building on this intuition. Although the proposed implementations differ in their details, their key innovation lies in optimizing the BHB update mechanism by assessing the bias status of a branch before adding its footprint to the BHB. This ensures that only footprints from non-biased branches are recorded, effectively preserving space for older branch histories.

This method incorporates a logical **Branch Status Table (BST)** alongside the conventional buffers and registers used in history-based branch predictors. As shown in Figure 3, the BST plays a critical role in the branch history update process by tracking the bias status of executed branches. Each *BST entry* contains two fields: *last branch outcome* and *bias status*, which together represent the status of branches.

The interested reader can refer to Appendix A for the update algorithm of BST entries. Most importantly, note that a branch that has never been seen before is always considered biased, as there is no evidence to suggest otherwise. This status only changes when the branch produces a secondary outcome that differs from the previously recorded one, at which point the relevant BST record is updated.

Experimental validation. To investigate this feature's behavior, we employed the reference snippet in Listing 1. In our experimental setup, `BH[n]` functions as a chain of indirect branches and conditional branches designed to fully populate the BHB and flush information from the previous context. The jump targets of indirect branches are carefully selected to chain them together while ensuring each branch consistently jumps to the same target address.

Based on this controlled BHB value, the prediction of `Bi_pred` should theoretically be manipulable through the outcome of `Bx_prime`. On most processors we evaluated, we consistently observed that the prediction of `Bi_pred` is indeed influenced by the outcome of `Bx_prime`, confirming the expected behavior of standard history-based prediction.

However, during our tests on the A72's BHB, we observed that the processor consistently yielding predictions matching the architectural target. This observation suggests that such a chain of branches fails to fully update the path history, leaving residual hints from earlier contexts within the BHB. By methodically chaining the indirect branches and directing them to different targets prior to initiating our tests, we were able to achieve the aforementioned control until process termination using the same branch chain. Interestingly, interference from other processes sharing the same processor core could subsequently disrupt this control mechanism.

These distinctive behaviors strongly indicate that the A72 may have implemented a bias-free branch predictor that utilizes a globally shared Branch Status Table.

5.2 BST Eviction

As Gope and Lipasti [17] suggested, the BST should be implemented as a fully-associative table indexed by the lower bits of branch addresses. To enhance isolation among different contexts, the BST may include an additional *tag* field for each entry, which is generated using a different hash function from the one used for indices. By verifying the *tag* value upon a query, it prevents the retrieval of records assigned to a different branch, thus mitigating potential value injection.

However, similar to other caches that use tags for isolation, in the context of the BST, *eviction* occurs when a branch attempts to acquire the slot that is already occupied by a victim. While the new branch will replace the existing entry with its own data, the record associated with the victim branch is removed from the BST. When the victim branch is executed again after eviction, *the victim branch will be classified as biased*, regardless of its previous behavior before eviction.

Observing BST eviction. Based on the interaction between the BHB and BST discussed above, we note that BST eviction can be monitored by observing its side effects on history-based branch prediction. Building on this premise, we demonstrate how BST eviction can be observed by *monitoring mis-speculations triggered by inaccurate branch histories*.

Consider the code snippet of Sec. 3.1. We train the BPU to predict `Bi_pred` under two distinct execution flows in which `Bx_prime` is configured as an indirect branch:

- $\mathbb{F}_A$: it invokes `BH[n]`, skips the optional `Bx_prime`, and then causes `Bi_pred` to jump to `t_primary`; and
- $\mathbb{F}_B$: it invokes `BH[n]` and `Bx_prime` in sequence, then causes `Bi_pred` to jump to `t_alt`.

Due to the presence of `Bx_prime`, $\mathbb{F}_B$ generates a BHB value for `Bi_pred` that differs from the one generated by $\mathbb{F}_A$. When alternatively executing $\mathbb{F}_A$ and $\mathbb{F}_B$ under normal conditions, the BPU should be able to differentiate these two flows and make accurate predictions based on the following BTB entries[2]:

$$\begin{aligned} &\mathbb{F}_A : (\texttt{Bi\_pred, BHB(BH[n])}) \rightarrow \texttt{t\_primary}, \\ &\mathbb{F}_B : (\texttt{Bi\_pred, BHB(BH[n]+Bx\_prime)}) \rightarrow \texttt{t\_alt}. \end{aligned} \tag{1}$$

We now introduce **`Bx_evict`**, a branch that contends for the same BST entry as `Bx_prime` (see Listing 1) through index aliasing, though it exists outside our reference snippet. `Bx_evict` causes the eviction of the BST entry upon its execution, altering the branch prediction status of `Bx_prime`. To demonstrate the effects of BST eviction, we can hence invoke `Bx_evict` in the middle of the two flows $\mathbb{F}_A$ and $\mathbb{F}_B$. On the execution of $\mathbb{F}_B$, this eviction will force `Bx_prime` to be classified as biased. The footprint of `Bx_prime` will hence be omitted, and the BHB will be populated solely based on the footprints of `BH[n]` as for $\mathbb{F}_A$.

Consequently, despite executing $\mathbb{F}_B$, the BTB entry for $\mathbb{F}_A$ will be used for predicting `Bi_pred`, leading to `t_primary` being mis-speculated in the mismatching context of $\mathbb{F}_B$.

BST on Cortex-A72. We implemented the reference snippet from Listing 1 as a userspace program to test this behavior on A72 using the NXP i.MX8QM chip. In this program, we define `Bx_evict` as an always-taken conditional branch and copy it to an address that shares at least 16 lower bits with with `Bx_prime` at program initialization. `t_primary` triggers a cache fetch to a designated probe address, while `t_alt` performs no observable operations.

The mis-speculation was observed with 100% success rate, demonstrating the presence of a Bias-Free Branch Predictor in the ARM A72 CPU. We also found that the logical BST is implemented as a 4096-entry table indexed by a 12-bit value derived from bits [15:4] of an instruction's virtual address. The successful eviction with a single branch indicates that the table is full-associative and tagged. Since `BH[n]` is not called before `Bx_evict`, the high success rate of mis-speculation indicates that branch history is not involved in the indexing process. Our experiments further revealed other possible sources of BST eviction, as detailed in Table 2, which we confirmed by testing different branch types for `Bx_evict`.

Additionally, we find that this behavior differs for conditional branches: details are available in Appendix C.

[2] Notation `(x, BHB(y)) → z` represents a BTB entry, indicating that branch `x` is predicted to target `z` when the branch history is `BHB(y)`.

Type	Mnemonics	Evict?
Indirect	BR, BLR	Yes
Conditional	B.cond, TB(N)Z, CB(N)Z	*When taken*
Unconditional	B, BL	No
Return (indirect)	RET	No
Other	SVC	No

Table 2: Operations triggering BST evictions on Cortex-A72.

Cross-context eviction. We further test if this primitive can bypass process isolation, privilege levels, and Spectre mitigations. We implement the reference snippet in a custom system call handler in Linux to check whether a userspace `Bx_evict` can affect branch prediction in kernel space. While we can train the BPU to distinguish $\mathbb{F}_A$ and $\mathbb{F}_B$ through the `syscall()` interface, `Bx_evict` can still induce mis-speculation in $\mathbb{F}_B$ and leave an observable data cache signal.

This result reveals gaps in the isolation and sanitization of BST by current Spectre software-based mitigations. The ARM-proposed Spectre-BHB mitigation [6] can effectively prevent explicit BHB value manipulation through population, but our kernel-mode proof of concept demonstrates a lack of sanitization of update policy after this barrier, creating a residual attack surface for crafting BHB values.

In the second experiment, we implement the reference snippet and `Bx_evict` as separate userspace programs. The victim program contains the reference snippet, while `Bx_evict` is placed at the BST-aliasing address and runs in an infinite loop. Both programs run concurrently on the same core, with the victim program actively yielding CPU time to allow `Bx_evict` execution. However, when the victim program regains the processor, we can neither observe the data cache signal nor detect the previously established entries for $\mathbb{F}_A$ and $\mathbb{F}_B$.

The Spectre-v2 mitigation on A72 employs a BPU flush implemented in the ARM Trusted Firmware to invalidate all BPU information [4]. While this reset can effectively sanitize all prediction state, we find that Linux applies it only to userspace context switches, similar to the restricted scope of IBPB on x86 processors. [1,21] With this mitigation disabled, we successfully observed mis-speculation caused by eviction from another userspace process.

Recent ARM processors implement a hardware-based isolation feature `FEAT_CSV2` [7] that adds context-dependent values to BTB tags. However, although our tested processor revision (r0p2) does not support this feature, ARM's feedback confirms that such eviction remains unrestricted by this feature.

5.3 Attack Flow #1: BiasScope

Building on BST eviction, this section presents **BiasScope**, a side-channel attack that leverages BST features to leak the outcome of branches, even in other exception levels.

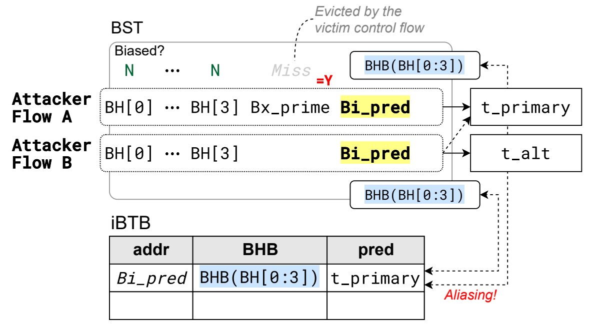


Figure 4: Monitoring a victim branch using BiasScope.

In spirit, BiasScope offers capabilities similar to those of *BranchScope* [14], but leveraging a different attack vector. BranchScope is a side-channel attack that extracts coarse-grained control-flow information by analyzing saturation counter values in the *Pattern History Table (PHT)* of modern branch predictors. Conversely, our BiasScope exploits BST eviction to monitor the execution flow of a victim program, resulting from a taken branch that evicts an existing BST entry with a shared aliasing index.

The core concept of BiasScope builds upon the primitive described in Section 5.2. BiasScope does not perform the eviction between $\mathbb{F}_A$ and $\mathbb{F}_B$. Instead, it initializes and maintains the *non-biased* record for `Bx_prime`, and yields the processor to the victim process. By alternating the execution of $\mathbb{F}_A$, $\mathbb{F}_B$, and the victim process, the attacker can detect whether a secret-dependent branch in the victim context triggered a BST eviction, effectively repurposing the BST itself as a side channel to extract information from other processes.

The attack flow is depicted in Figure 4. This BST side-channel involves a secret-dependent **sender** branch in the victim context and a **receiver** snippet controlled by the attacker. The sender branch corresponds to `Bx_evict` and `Bx_prime` must be selected by the attacker so that the two branches share the same BST entry (e.g., same address bits [15:4] on Cortex-A72). According to the BST eviction behavior we found, it can be a *bare* conditional branch that can be monitored directly, or an indirect branch *nested* within a conditional block, thereby exposing the execution status of the preceding conditional branch. The receiver leverages flows $\mathbb{F}_A$ and $\mathbb{F}_B$ introduced in the previous section, utilizing the history-related components `BH[n]`, `Bi_pred`, and `Bx_prime` to determine whether the sender branch was taken.

The BiasScope attack proceeds as follows:

1. **Preparation**: The attacker first forces `Bx_prime` to alternate between two legit targets to establish its *non-biased* record in the BST. Additionally, the attacker ensures that the branches in `BH[n]` are recorded as *non-biased* to fully populate the BHB.

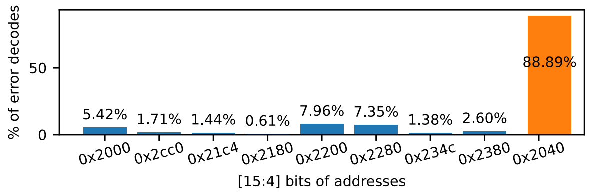


Figure 5: Error rate of BST side channels under intra-process BiasScope with branch addresses with different [15:4] bits.

2. **Victim Execution**: The attacker yields control of the CPU, allowing the secret-dependent sender branch to execute in the victim context.

3. **Observation**: After regaining CPU control, the attacker alternates between executing $\mathbb{F}_A$ and $\mathbb{F}_B$ to verify the presence of the BST entry of `Bx_prime`.

If the sender branch was taken during this period, the *non-biased* status of `Bx_prime` will be lost. Consequently, the BPU will classify `Bx_prime` as a biased branch, omitting its footprint in $\mathbb{F}_B$ and causing the BHB value to match that of $\mathbb{F}_A$. In this scenario, `t_primary` is mis-speculated in $\mathbb{F}_B$, which can be detected using a micro-architectural probe (e.g., cache hit). This allows the attacker to leak the state of the victim branch, possibly revealing secrets in the victim context.

Evaluation. We first evaluated whether we could leak the taken status of an injected kernel branch with *all default mitigations enabled*. While controlling its conditional direction from userspace, our results demonstrate that this vector can accurately detect taken events of conditional branches executing in kernel space. We further evaluated the performance of this BST side-channel using both sender and receiver running in userspace, disabling the Spectre-v2 mitigation (discussed in Section 5.1) for the sole purpose of this experiment. The sender encodes an 8-bit secret using eight independent conditional branches, with each branch controlled by one bit of the secret byte. In each iteration, the receiver yields the core to the sender by sleeping briefly, allowing the sender to encode a secret byte into the BST side channel. When the receiver resumes execution, it attempts to decode all eight bits. The error rates in decoding each bit are illustrated in Fig. 5. Our experiments demonstrate a high signal-to-noise ratio side channel. However, we also observe that some branch addresses (e.g., with bits [15:4] = `0x2080`) become completely jammed for certain periods, suggesting interference from other code snippets sharing the same processor core.

BiasScope converts the presence of a BST entry into observable branch latency. To effectively perform the attack, the attacker must have a detailed understanding of the target CPU's branch latency characteristics to decode the measured branch latency. Furthermore, since the BST can track multiple branches simultaneously, BiasScope can monitor several non-aliasing branches concurrently, improving the granularity of observations on the victim's execution flow.

5.4 Attack Flow #2: Spectre-BSE

While BiasScope demonstrated how data leakage can be facilitated by observing BST eviction caused by the victim, we now exploit this behavior in the opposite direction, triggering malicious mis-speculations in the victim context. We introduce **Spectre-BSE (Branch Status Eviction)**, a novel *target reuse attack* primitive that exploits BST evictions to facilitate BHB aliasing and trigger malformed speculative executions.

As we demonstrated before, since the bias-free mechanism has an obvious influence on the BHB updating policy, an attacker may manipulate the generation of BHB value by controlling the presence of relevant BST records. This may cause unexpected BHB values, which can be further exploited to induce BTB index aliasing, reaching a similar result to *Branch History Injection* (Spectre-BHB) [8]. In this section, we demonstrate how BST eviction can manipulate branch prediction and trigger secret data disclosure.

Spectre-BHB alters the BTB query and selection behavior by manipulating the BHB values with footprints from attacker-controlled branches, unlike Spectre-v2, which directly injects a BTB entry into the target entry. In history-based BPUs, the BTB index function typically incorporates both the branch address and the BHB value. This dependency can be exploited by crafting malicious BHB values, leading to confusion between two different branches. This behavior enables a "Target Reuse Attack", facilitating implicit *out-of-place* branch target injection and bypassing existing Spectre-v2 mitigations.

However, Barberis et al. [8] stated that they were unable to reproduce out-of-place Spectre-BHB attacks on ARM devices. Considering that our attack Spectre-BSE ultimately relies on the same BTB indexing mechanism as Spectre-BHB, we currently limit our demonstration to in-place branch target training, while demonstrating an inherent out-of-place vector for manipulating BHB updates on the Cortex-A72 processor. It is important to note that since neither Barberis et al. [8] nor ARM [6] has completely ruled out the possibility of out-of-place BHI attacks on ARM processors, we conjecture that it remains feasible to conduct a *fully* out-of-place target reuse attack using our Spectre-BSE.

Consider the reference snippet of Sec. 3.1 without `Bx_prime` and the following execution flows:

- $\mathbb{F}_A$: it invokes `BH[n]` then `Bi_pred` jumps to a disclosure gadget **`t_leak`**; and

- $\mathbb{F}_B$: it invokes `BH[n]`, sets a secret-related context (e.g., in registers), and eventually jumps to **`t_safe`**, a benign target that poses no security risks.

While branches in `BH[n]` may, e.g., validate the passed parameters to prevent micro-architectural illegal memory loads, they should also create distinct BHB values, resulting in two BTB entries corresponding to $\mathbb{F}_A$ and $\mathbb{F}_B$, respectively. Similar to typical Spectre attacks, triggering the mis-speculation of `t_leak` in a mismatching context, i.e., $\mathbb{F}_B$, allows the attacker to induce data disclosure from a secret context.

BST eviction plays a critical role in this attack by *forcing a typically non-biased branch to be classified as biased*, thereby generating an unexpected BHB value in the BTB query or update process. When a branch is executed after the corresponding BST entry is evicted, the BHB is not updated, causing one oldest footprint to remain inside the BHB, leading to an unexpected BHB value in subsequent control flows. In some occasions, this may make the BHB value used to predict `Bi_pred` alias with a mismatching execution flow, causing the wrong record to be used in the prediction.

The Spectre-BSE attack hence proceeds as follows:

1. **Preparation:** Exploitability hinges on the BHB footprint of `BH[n]` in both flows $\mathbb{F}_A$ and $\mathbb{F}_B$. The attacker identifies flows $\mathbb{F}_A$ and $\mathbb{F}_B$ such that `BH[n]` generates two footprint sequences: $\texttt{BHB(BH[n]}|\mathbb{F}_A\texttt{)}$ and $\texttt{BHB(BH[n]}|\mathbb{F}_B\texttt{)}$. Exploitation is possible if, by excluding a subset $\mathcal{B}_{ev} \subseteq$ `BH[n]` from $\mathbb{F}_B$, BHB aliasing occurs, i.e.,
$$\texttt{BHB(BH[n]}|\mathbb{F}_A\texttt{)} = \texttt{BHB(BH[n]-}\mathcal{B}_{ev}|\mathbb{F}_B\texttt{)}. \tag{2}$$
The attacker then invokes $\mathbb{F}_A$ to initialize a BTB entry.

2. **Eviction:** The attacker performs a targeted BST eviction on $\mathcal{B}_{ev}$ using another branch that contend for the same BST entry due to aliasing.

3. **Leakage:** The attacker invokes $\mathbb{F}_B$, inducing mispeculation towards `t_leak` while retaining a secret-related context.

Compared to Spectre-BHB [8], Spectre-BSE manipulates BHB values through BST eviction rather than directly injecting attacker-controlled branches. Furthermore, it does not require a short execution path between the snippet entry and the victim branch, allowing for more flexibility in attack scenarios and significantly broadening the potential attack surfaces.

Suppose `BH[n]` comprises five indirect branches, labeled as `BH[0]` through `BH[4]`. All these branches are initially non-biased, and their bias statuses are trained and stored in the BST before the attack. Under unconstrained BHB budget capacity, the execution of this branch sequence would yield the following BHB values for the two flows: **(i)** $\texttt{BHB(BH[n]}|\mathbb{F}_A\texttt{)=[A,B,C,D,E]}$; **(ii)** $\texttt{BHB(BH[n]}|\mathbb{F}_B\texttt{)=[B,C,D,E,F]}$, where capital letters denote some example BHB values. As the budget capacity of BHB is limited in practice, let us assume the BHB retains only the four most recent footprints (as, for instance, we found happening in Cortex-A72).

To setup the attack, we first invoke $\mathbb{F}_A$ to initialize a BTB entry. This results in the following BTB entry:

$$\mathbb{F}_A : (\texttt{Bi\_pred, [B,C,D,E]}) \rightarrow \texttt{t\_leak}. \tag{3}$$

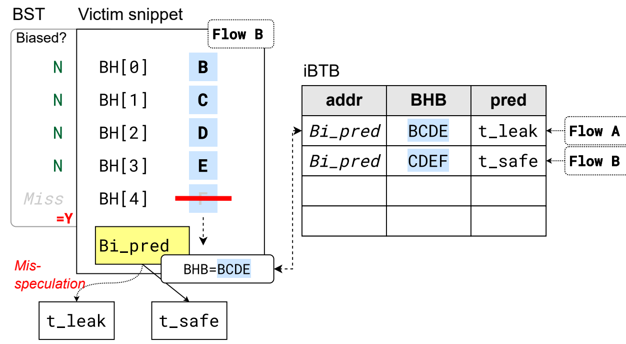


Figure 6: BHB aliasing in Spectre-BSE.

At this stage, the environment is almost prepared, and the attacker is ready to proceed with the malicious actions. The attacker performs a targeted BST eviction on $\mathcal{B}_{ev}$ = `BH[4]`, then invokes the vulnerable code snippet with $\mathbb{F}_B$.

As depicted in Fig. 6, due to eviction, the BST query for `BH[4]` results in a miss, causing its footprint to be omitted during BHB updates. Hence, differently from the the nominal case in which $\texttt{BHB(BH[n]}|\mathbb{F}_B\texttt{)} = \texttt{[C,D,E,F]}$, upon $\mathbb{F}_B$'s execution the BHB value will be updated to $\texttt{BHB(BH[n]-}\mathcal{B}_{ev}|\mathbb{F}_B\texttt{)} = \texttt{[B,C,D,E]}$ instead, with the footprints of `BH[0:3]` remaining inside the buffer and aliasing with the value associated with $\mathbb{F}_A$ (see Eq. (3)). This maliciously-constructed BHB value forces the BPU to speculate `t_leak` instead of `t_safe` for `Bi_pred`, potentially exposing sensitive data during transient execution.

Evaluation. We evaluate our exploit using a userspace branch to evict victim branches in both the same process and a `syscall()` handler with *all default mitigations enabled*. When the eviction branch is placed at a 32-byte aligned address sharing bits [15:4] with the victim, we observe cache hits from the victim flow with 99.9% success rate in both intra-process and cross-privilege attacks, consistently with our previous findings.

6 Exploitation 2: Branch History Speculation

This section presents Spectre-BHS. Before proceeding, it is necessary to introduce Branch History Speculation.

6.1 Early BHB Updates

Due to the significant speed disparity between memory access and pipeline execution in modern processors, branch resolution can be delayed by up to hundreds of cycles. Speculative execution enables the CPU frontend continue filling the pipeline by speculatively executing instructions during this period, potentially issuing hundreds of uncommitted instructions within the speculation window. While this technique is essential for pipeline efficiency, it introduces new challenges when *branches appear within the speculation window*.

To maintain backend utilization and avoid pipeline stalls, the frontend must predict and execute additional branches encountered in the speculative execution path until backend resources are exhausted, rather than halting speculation upon encountering new branches.

Although history-based branch prediction is widely adopted for this purpose, predicting a branch within the speculation window poses a unique challenge: since preceding branches may remain uncommitted and the execution path is still speculative, constructing an accurate branch history for the current prediction becomes difficult. However, if the BHB updates only upon branch resolution, these predictions, made inside speculation windows, will rely on an outdated branch history, if any exists.

To address this limitation, it becomes essential to *update the BHB speculatively based on predictions*, even before branch outcomes are confirmed, rather than waiting for the commit stage which would significantly delay predictions. Modern BPUs introduce **Branch History Speculation (BHS)** [18] through various rollback mechanisms [16, 47, 53]. This approach allows speculatively-predicted branch outcomes to immediately update the global branch history, either in the main BHB or a dedicated speculative history buffer.

Since speculative predictions become immediately visible to subsequent branches in the speculation window, the BHB remains up-to-date for further predictions. This allows the pipeline to follow previously-learned execution paths, improving efficiency even when data dependencies remain unresolved.

6.2 Attack Flow #3a: Spectre-BHS

This mechanism further implies that a branch predicted in the speculation window may be influenced by the outcomes of earlier branches that are also speculated but not yet resolved. To systematically investigate how unretired instructions impact early BHB/PHR updates, we further extend the experiment in Section 3.3 in combine with the reference snippet in Section 3.1. Through this analysis, we introduce Spectre-BHS, a novel variant of Spectre attack that manipulating the BHB updating mechanism through BTB/PHT mis-training.

Cascaded mis-speculation. In our experimental setup, we configure `Bx_prime` as a *conditional* branch dependent on a variable `flag`. While `BH[n]` populates the BHB with a predetermined path, the prediction of `Bi_pred` is principally determined by the outcome of `Bx_prime`. To ensure the speculation window opens at `Bx_prime`, we flush both `flag` and the jump pointer for `Bi_pred` from the cache.

We define two flows, $\mathbb{F}_A$ and $\mathbb{F}_B$, similar to those in Section 5.2, which create distinguishable BTB/PHT entries for `Bi_pred` based on `Bx_prime`'s footprint:

- $\mathbb{F}_A$: `Bx_prime` is *not taken* then `Bi_pred` jumps to `t_leak`; and
- $\mathbb{F}_B$: `Bx_prime` is *taken* then `Bi_pred` jumps to `t_safe`.

Following the methodology detailed in Section 3.3, we further evaluated whether we could manipulate the prediction of `Bi_pred` by controlling the outcome of `Bx_prime`. We started from single mis-train branch. The results demonstrate that all processor cores we tested consistently yielded the expected prediction of `Bi_pred` based on the prediction of `Bx_prime`, providing strong evidence that the BHB is indeed updated speculatively across all evaluated microarchitectures. This confirms that history-based branch predictions are consistently made using the most recent branch histories, even when those histories include speculative branches.

To validate this hypothesis, we constructed multiple address-congruent mis-train snippets, each containing the identical `BH[n]` sequence and a mis-train conditional branch `Bc_mt` strategically placed at addresses that conflict with `Bx_prime` (i.e., sharing the same lower address bits). This experimental configuration enables us to systematically mis-train `Bx_prime`'s prediction by executing these conflicting branches while simultaneously establishing a conflicting history pattern in the PHR.

Out test works as follows:

1. **Preparation:** We repeatedly invoke $\mathbb{F}_A$ and $\mathbb{F}_B$ to train the BPU to recognize both flows, letting `Bc_fp` to be recorded as taken in $\mathbb{F}_B$.
2. **Mis-training:** Then, we invoke all mis-train snippets, with the mis-train branch to influence `Bx_prime`'s prediction record.
3. **Mis-speculation:** To ensure a large speculation window that involves both `Bx_prime` and `Bi_pred`, we flush from cache variable `flag` and the pointer variable used by indirect branch `Bi_pred`. Finally, we execute $\mathbb{F}_B$ and monitor for the presence of the data cache signal left by `t_leak`.

Similar to the experiment in Section 3.3, we initiated our testing with a single mis-train snippet. Through manipulation of the direction of `Bc_mt` with just one mis-train snippet, we could control the speculated target of `Bi_pred` with approximately 100% success rate on all evaluated platforms. This result demonstrates that speculated branch outcomes can indeed update branch history and influence subsequent speculations within the same speculation window.

BTB/PHT eviction in PHR. Our observations in Sec. 3.2 show that Cortex-A76 and A78AE employ path history (see also Sec. 2.1), where `Bx_prime` updates the history in PHR only when taken. Based on this architectural insight, we hypothesize that when BTB/PHT eviction forces the BPU to

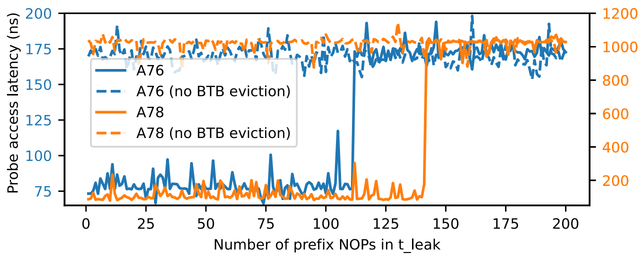


Figure 7: Access latency of data cache probe with varying numbers of prefix `NOP` instructions in `t_leak`. Lower latency indicates successful speculative execution of data load. Results averaged over 1,280 tests per configuration, shown with and without BTB evictions.

"forget" a branch, such a not-yet-recorded branch will not update the BHB before it is detected, causing it to be *implicitly misinterpreted as not-taken* in the path history. This creates confusion between $\mathbb{F}_A$ and $\mathbb{F}_B$ during speculation so that $\texttt{PHR(BH[n]}|\mathbb{F}_A\texttt{)} = \texttt{PHR(BH[n]-}\mathcal{B}_{ev}|\mathbb{F}_B\texttt{)}$. This could induce mis-speculation to `t_leak` in the context of $\mathbb{F}_B$, which can be exploited to achieve data disclosure like existing Spectre attack vectors.

On A76 and A78AE, as the number of *taken* mis-train branches increased. we observed the processor began selecting the path with `Bx_prime` as not taken, leading to speculating `t_leak`. This behavior closely parallels the phenomenon documented in Section 3.3. In our userspace testing of 12,800 trials, we successfully achieved mis-speculation through BTB eviction with success rates of 99.84% and 99.29% on A76 and A78AE respectively. We extended this experiment with kernel-space victims on A76, achieving a comparable success rate of 99.33%. This result confirms that PHR is prone to transient confusion about the branch outcomes based on incomplete and unconfirmed branch history. Additionally, as we previously excluded the possibility of inducing BTB/PHT eviction in Intel processors, we were consequently unable to replicate this specific eviction-based mis-speculation vector across the Intel architecture family.

BTB/PHT eviction in BHB+PHR. Having established different BHB implementations in Cortex-A72 and AMD Zen4, we could not reproduce the eviction-induced behavior on these processors as we discussed above. This indicates fundamental architectural differences in branch history management, which we attribute to their hybrid BHB+PHR implementations. For a comprehensive analysis of these architectural differences and potential attack surface, see Appendix C.

Speculation window. While the BPU transiently overlooks the presence of $\mathcal{B}_{ev}$ branches, once these evicted branches retire, the BPU corrects the BHB value and flushes the pipeline to recover the correct state. Therefore, unlike Spectre-BSE which has a persistent effect on branch history, the speculative execution of `Bi_pred` and any leakage operations must occur *before* all data and control dependencies of $\mathcal{B}_{ev}$ are satisfied and before the branches are resolved. We observe that barrier instructions (`dsb isb` or `mfence`) placed before `Bi_pred` effectively prevent speculating `t_leak` under vulnerable configurations, further confirming the speculation window requirements.

This timing constraint is crucial to exploit the speculative path effectively. As illustrated in Fig. 7, our experiments demonstrate that when induce the mis-speculation through BTB/PHT eviction, Coretx-A76 and A78AE can execute more than 100 instructions within the speculative window, successfully performing the data leakage operation at its end. Furthermore, if the BPU can detect a branch before its resolution (e.g., during the decoding stage), the speculation window may terminate prematurely. Notably, we also observed that mis-training techniques yield speculation windows of comparable size, suggesting that the evaluated BPU implementations only detect the presence of a branch upon its architectural resolution rather than during earlier pipeline stages.

7 Exploitation 3: BHS & Fallback Predictions

While the BPU updates the BHB with speculated outcomes, other architecturally resolving branches within the speculation window also influence predictions, potentially deviating from learned execution paths. This section explores how this effect in BHS schemes can be exploited to truncate history-based predictions, demonstrated using *extended Berkeley Packet Filter* (eBPF) [46].

7.1 A Special Case of Legit eBPF Programs

Let us examine another code snippet consisting of two main blocks: the latter performs a data-dependent load, while the former initializes the register context that can repurpose the latter block as a Spectre gadget. To prevent data disclosure through architectural execution, the snippet employs two `if` statements (conditional branches), denoted as **`Bc_init`** and **`Bc_load`**, that are mutually exclusive through complementary conditions (i.e., `Bc_init="if(flag)"` and `Bc_load="if(!flag)"`). This complementary structure ensures these blocks never execute together architecturally in the same instance. Additional branches may exist in implementations of such snippets to handle supplementary logic.

In eBPF, the *verifier* statically examines each potential execution path in submitted programs to identify any violations of safety constraints. Among its various strict safety checks, the verifier enforces memory safety through two key requirements: first, all memory accesses must refer to a base pointer of a pre-allocated buffer, with the actual pointer value fixed at JIT compilation time; second, any added offset must be a scalar value within the buffer's size limits in any branch path. Since the verifier evaluates these mutually exclusive blocks as separate execution paths and confirms their individual safety properties, such programs are deemed memory-safe and approved for loading.

For programs with the structure introduced in Sec. 7.1, exploitation would be straightforward in an environment vulnerable to out-of-place Spectre-v1. Since the attacker has full control over the execution contexts of mis-trai branches, they can separately mis-train multiple branches while preparing appropriate BHB values, even when BTB/PHT is indexed using both Program Counter and branch history.

However, our experiments on ARM processors in Section 3.3 demonstrate that both out-of-place Spectre-v1 training and BHB eviction require a congruent BHB value to succeed. This requirement poses significant challenges for setting up the attack, as determining the necessary BHB value through static analysis can be difficult in real world. Moreover, as shown in Section 6.2, Straight-Line Speculation (SLS) can interfere with branch history generation. Specific attack configurations—such as triggering SLS on `Bc_init`—may prevent the processor from speculatively executing `Bc_load` to reach the data disclosure gadget due to altered branch history. Given these complexities, we limited our analysis to the general case without considering out-of-place Spectre-v1 and SLS effects.

7.2 Breaking The Speculative Path

The mechanism of BHS suggests that a typical Spectre-v1 attack can affect the speculation of all subsequent branches under the BHS scheme. However, while this behavior creates a limited attack surface for controlling subsequent speculation, it also impedes the creation of speculative flows that combine code blocks from different execution contexts. Hence, a critical question emerges: *is it possible to construct a code snippet that induces branch misprediction by exploiting history-based path speculation itself?*

Let us re-examine the behavior of branch history updating. When both branches introduced in Sec. 7.1 appear within the speculation window, the speculated outcome of `Bc_init` immediately updates the BHB, thus influencing the prediction of `Bc_load`. However, if an attacker can force `Bc_load` to be *predicted without using global branch history*, then `Bc_load`'s prediction may become independent of `Bc_init`'s outcome. This could enable combining elements from different legitimate flows into a single, BHS-induced speculative execution. While history-based prediction can improve accuracy in most situations, BPUs must maintain the ability to predict based solely on Program Counter value to achieve better coverage in complex environments, particularly for branches that correlate poorly with history. Many state-of-the-art BPU designs, such as TAGE [48–51, 67], have implemented both PC-based and history-based sub-predictors (see Sec. 2.1).

Updating TAGE upon mis-prediction. TAGE predictors always update based on the encountered outcome of branches. First, they update the provider component, which is the table that supplied the final prediction. Then, upon a misprediction, if the provider component is not the table with the longest branch history, the BPU may allocate new entries in tables with longer histories, recognizing that the branch might correlate better with a longer history pattern.

For previously *non-executed* branches, we thus hypothesize that the fallback mechanism selects the base predictor `T0` as the provider component, since all other tables report query misses. The branch's outcome is then recorded in `T0`, establishing a new prediction entry with zero-length history.

7.3 Attack Flow #3b: Summon the Chimera from Fallback Predictions

While BHS limits the construction of in-place Spectre attacks on the programs of Sec. 7.1, fallback behaviors in BPUs create an opportunity: branch predictions may not always depend on the speculated path, enabling BHS for execution paths that never existed architecturally from fragments of legitimate ones. We demonstrate this variant of Spectre-BHS attack with an eBPF program that complies with the layout of Sec. 7.1.

Branch history shuffling. Similarly to the observations made by Wikner and Razavi in [63] for x86_64 architectures, while creating a nested speculative execution environment, we note that it is possible to "shuffle" the branch history related to `Bc_load`, forcing the BPU to *make predictions independent of branch history*. A simple way to do so is to provide a dedicated conditional branch, say *BHB-shuffle*, that is never architecturally taken before the attack so that it never updated branch history. Conversely, BHB-shuffle is intentionally taken at the stage of attack, hence injecting a history that was never encountered before. This causes prediction to fall back to the `T0` predictor, which is also required to be trained.

Algorithm 1: A vulnerable program passing the eBPF verifier.

```
params ← LEGIT_PARAMS;
if take_sc is FALSE then
    if set_ptr is TRUE then                    // Bc_init
        params ← &SECRET;
    if set_ptr is TRUE & esc is TRUE then
        exit;
    if shuffle_BH is FALSE then NOP;
if esc is FALSE then
    exit;
if set_ptr is FALSE then                       // Bc_load
    memload (params);
```

Exploitable snippet. We demonstrate a vulnerable snippet in Alg. 1 that satisfies the conditions for a successful attack. Besides `Bc_init` and `Bc_load`, given that the discussed layout and eBPF verifier allow additional branches while preserving our requirements, we introduce some additional components to make the snippet practically exploitable.

1. A conditional *BHB-shuffle* branch (line 7) between `Bc_init` and `Bc_load` meant to force fall-back predictions with `T0` when taken. This splits speculative execution into two parts: **(i)** the *history-based part*, where branches are predicted using BHB, and **(ii)** the *PC-based part*, where we will induce the BPU to predict using `T0` only (i.e., using PC values).

2. Conditional escape blocks redirecting control flow outside the snippet both before and after the BHB-shuffle branch. The first escape (line 5) provides an execution path where the BHB-shuffle branch never executes architecturally, while the second one (line 8) allows branches in the PC-based part to avoid training the BPU.

3. A shortcut path to the PC-based part. This branch (line 2) bypasses the history-based part, enabling isolated training of `T0` for branches in the PC-based part.

Note that due to diverse microarchitectural implementatons and behaviors in real-world processors, other snippet structures may also be vulnerable to similar in-place mis-training.

Preparation. We initialize exploited BTB/PHT records using the following two flows. During these training flows, we ensure the BHB-shuffling branch remains not-taken by setting shuffle_BH=FALSE:

(A) take_sc=FALSE, esc=FALSE, set_ptr=TRUE.

(B) take_sc=TRUE, esc=TRUE, set_ptr=FALSE.

Their branch traces are available in Appendix D. Our attack aims to mis-speculate a *crafted* execution flow combining the *history-based part* of Flow **(A)** with the *PC-based part* of Flow **(B)**. The PC-based part of Flow **(B)** skips the escape command (esc=TRUE) and transmits data using a side channel (line 10). Since this part is never architecturally observed in the same execution flow together with taken branches from the preceding lines, it will leverage the base predictor `T0` for branch speculation. To prevent TAGE from escalating to longer history-based predictions, we invoke this flow before any other training and avoid executing these branches in different contexts, ensuring these prediction records are created and remains in `T0`.

The history-based part of Flow **(A)** initializes registers to make pointers dereference a secret address. Since the snippet always executes through a common entry point (where we assume consistent branch history), we must train the BPU to speculate Flow **(A)** under the default history-based prediction scheme, ensuring execution of register initializations (line 4).

Triggering data disclosure. We construct a vulnerable context by setting shuffle_BH=TRUE to start the attack. The attacker flushes set_ptr from cache, then invokes the snippet with a **dedicated attack flow**, with take_sc=FALSE, set_ptr=TRUE, and esc=TRUE to trigger the data leakage.

Due to the cache miss on line 3, the processor opens a speculation window and executes subsequent branches based on learned predictions. Since the BPU is trained to speculate Flow **(A)** at the common entry, line 3 will be predicted as *not taken* and line 5 as *taken*. When the execution flow reaches line 7, since shuffle_BH remains in the cache and the branch resolves as *taken*, it leaves a footprint in the BHB. From this point, subsequent branches encounter a previously unseen "shuffled" history. Based on the fallback prediction mechanism discussed earlier, all subsequent branches on lines 8 and 10 will be predicted using the PC-based `T0` base predictor.

Line 8 resolves quickly as taken since esc remains in the cache, thus skipping the escape opcode. Finally, line 10 will use the PC-based prediction left by Flow **(B)**, which is *not taken*. This triggers a memory load with the illegal, secret-dependent params set in line 4.

In the end, the processor discovers its mis-speculation and reverts all speculative changes. All speculative results, including the BHB-shuffling branch on line 7 and correlations among speculated branches, are not recorded by the BPU. The branch on line 7 remains never-taken architecturally, and predictions for lines 8 and 10 stay unchanged. To maintain an exploitable environment, the attacker must preserve and refresh the BTB/PHT entries for Flow **(A)** and **(B)** in their respective sub-predictors.

We first implement Algorithm 1 as a C program. This program demonstrate successful speculation of both pointer setting and load gadget operations, achieving 100% and 99.85% success rates on Cortex-A76 and A78AE, respectively.

Evaluation in eBPF. Kirzner and Morrison [26] demonstrated that although eBPF verification ensures memory safety in architectural execution paths, it cannot protect eBPF against speculative execution. Their work showed how *cross-address-space, out-of-place* Spectre-v1 can compromise this safety assumption. Their work proposed enhanced verification for unprivileged user programs that rigorously examines memory access constraints across all potential speculative execution paths, even those that are unreachable. We believe this patch, implemented in v5.13rc7 and subsequently backported by various distributions, prevented the execution of Chimera from unprivileged contexts. Consequently, our testing was conducted in privileged mode to bypass these restrictions.

In the eBPF implementation, `LEGIT_PARAMS` satisfies the verifier by initializing two registers with a legitimate buffer pointer and offset for the data load block, while `Bc_init` sets these registers to zero and `&SECRET`, respectively.

We tested this program in privileged mode. To confirm whether the BHS is enabled in kernel space, we intention-

µArch	*Mitigation*			*Primitive*			
	BHB Clear	CSV2	BPU Flush	BHB	BSE/BiasScp.	BHS	Chimera
Cortex-A72	◐	●	◒	✗	✓	✓	—
Cortex-A76/A78AE	◐	●	◒	✗	—	✓	✓
	BHI_DIS_S	e/aIBRS	IBPB	BHB	BSE/BiasScp.	BHS	Chimera
Zen4	—	◐	◒	✗	—	M+C & E+C	✓
Gracemont	◐	◐	◒	✗	—	M+C	✓
Redwood Cove / Crestmont	◐	◐	◒	✗	—	M+C	✓

Table 3: Spectre mitigation techniques and exploitability of proposed attack vectors. For *mitigations*, ◐ =Enabled for cross-privilege, ◒ =Enabled for cross-context, ● =Enabled by default, and "—" Not applicable. For *primitives*, ✓ =exploitable with recommended mitigations, ✗ =fully mitigated, and — =Not applicable or not exploitable on this architecture. For Spectre-BHS on x86 processors, "**M+C**" and "**E+C**" indicate using **M**is-training or **E**viction to hijack kernel **C**onditional branches, respectively.

ally mis-trained the bias of `Bc_init` and flushed all four flag variables. We observed that `LEGIT_PARAMS` is successfully encoded in the data cache when `Bc_init` is biased toward taken, confirming the presence of the BHS scheme for privileged conditional branches. Given that Spectre-v1 attacks are widely recognized as not fully mitigated and rely on ad-hoc software mitigations, we conjecture that conditional branches, including privileged ones, can be exploited in Spectre-BHS attacks as victims through the manipulation of other branches within their speculative execution paths.

Through the malicious configuration discussed above, it successfully leaks arbitrary kernel memory contents. While mis-speculations occasionally fail, we find that restarting it causes the kernel to assign a new address to the JITed snippet, avoiding interference and stabilizing the attack. Under optimal conditions, we achieve a leakage rate of 24,628 Bit/s on A76 using single-pass bit extraction with 100% accuracy.

Moreover, we successfully replicated these results across evaluated AMD and Intel processors, though failed to reproduce this attack vector on A72, suggesting the absence of the fallback mechanism on this microarchitecture.

However, it is important to note that, while this experiment was conducted with some mitigations disabled, similar exploitable patterns likely persist in production environments lacking comprehensive ad-hoc protections.

8 Mitigations

In Table 3, we provide a comprehensive overview of existing hardware and software mitigations against our branch prediction attacks on the tested processors.

ARM has introduced a software-based BHB populating sequence [6] to clear branch footprints generated in user mode. However, our experiments have demonstrated that this approach fails to isolate the BHB updating policy between user mode and privileged mode, allowing our attack primitives to bypass this countermeasure in all tested ARM processors.

AMD's AutoIBRS on Zen4 disables execution of predicted targets for kernel indirect branches, while Intel's `BHI_DIS_S` disables history-based prediction in kernel space entirely [60]. These mechanisms significantly constrain the exploitability of kernel-space indirect branch targets on x86 processors; however, our testing reveals that prediction of conditional branches remains unrestricted despite these mitigations, leaving potential conditional branch-based attack vectors exploitable. Thus far, our efforts have not yielded positive results in this area.

Recent microarchitectures implement implicit predictor mode separation by incorporating additional context information into branch prediction records. Notable implementations include ARM's CSV2 and Intel's eIBRS [20]. Our evaluation demonstrates that these features remain insufficient to prevent BTB and PHR manipulation through mis-training and eviction, which subsequently influence the BHB updating process.

Some processors adopt aggressive Spectre-v2 mitigations like IBPB (x86) [1, 21] and BPU flush (ARM) [4] that clear prediction records. ARM applies them to pre-CSV2 processors (e.g. A72), while they are widely deployed across x86. Although these mitigations could effectively neutralize our attacks by invalidating malicious BPU configurations, a full deployment may degrade system performance by more than 50% [12]. Hence, the deployment is typically restricted to user space context switches, leaving syscalls unprotected.

9 Conclusion

This paper investigated how resource sharing and contention in modern BPUs can originate security vulnerabilities in speculative execution when injecting inaccurate branch history. Our findings allowed to propose three novel attacks, Spectre-BSE, Spectre-BHS, and BiasScope, which were successfully tested on multiple processors, exhibiting a very high signal-to-noise ratio. A variant of Spectre-BHS was implemented by means of eBPF, demonstrating its capability of leaking kernel memory contents at 24,628 bit/s. In the light of recent work [23, 37, 39, 59] that revealed a wide availability of Spectre gadgets in the Linux kernel and the threats posed by speculative trojans [72], this research should set the stage for future investigations to prevent these new attacks in uncontrolled environments.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU, and by the Dottorato di Ricerca Nazionale in Cybersicurezza.

Ethics Considerations

We disclosed our findings to ARM in September and October 2024. Among all the disclosed issues, ARM issued CVE-2024-10929 for Spectre-BSE, released a security advisory to address this vulnerability, and confirmed that BiasScope depends on the same underlying behaviour with it. We also disclosed our findings to Intel and AMD in November 2024.

We followed best practices for responsible disclosure, notifying ARM of our findings as soon as possible, and keeping our findings confidential. No experiments were performed with live systems.

Open Science

The artifacts implementing proof-of-concept demonstrations of Spectre-BSE, Spectre-BHS, BiasScope, and the Chimera attack are publicly available at https://zenodo.org/records/15612187. These artifacts include comprehensive test modules, cross-context demonstrations, and the complete eBPF-based Chimera implementation. We are confident that these artifacts enable the security community to verify and reproduce our findings.

References

[1] The Linux Kernel documentation - Spectre Side Channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

[2] Muawya M. Al-Otoom, Paul Caprioli, and Jeffrey J. Cook. Detecting and Filtering Biased Branches in Global Branch History, June 2014. https://patents.google.com/patent/US20140156978A1/en.

[3] AMD. Software Optimization Guide for the AMD Zen4 Microarchitecture, January 2023.

[4] ARM. Trusted Firmware-A 2.12.0 documentation: 9.6. Advisory TFV-6 (CVE-2017-5753, CVE-2017-5715, CVE-2017-5754). https://trustedfirmware-a.readthedocs.io/en/latest/security_advisories/security-advisory-tfv-6.html, June 2018.

[5] ARM. Straight-line Speculation Whitepaper. https://developer.arm.com/documentation/102825/latest/, June 2020.

[6] ARM. Spectre-BHB: Speculative Target Reuse Attacks version 1.8-r0p2. https://developer.arm.com/documentation/102898/0107/?lang=en, December 2023.

[7] ARM. Arm Architecture Reference Manual for A-profile architecture version L.a. https://developer.arm.com/documentation/ddi0487/la/?lang=en, December 2024.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[10] Po-Yung Chang, Marius Evers, and Yale N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. *International Journal of Parallel Programming*, 25(5):339–362, October 1997.

[11] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. *ACM SIGARCH Computer Architecture News*, 25(2):274–283, May 1997.

[12] Bjoern Doebel. Re: [PATCH 0/3] arm64: Proton-pack: Add Spectre-BSE mitigation for Cortex-A7{2,3,5} - Doebel, Bjoern. https://lore.kernel.org/linux-arm-kernel/965352b6-c21f-4161-9f23-2e96cc1267f6@amazon.de/, January 2025.

[13] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 249–258, December 1998.

[14] Dmitry Evtyushkin, Ryan Riley, Nael CSE and ECE Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM SIGPLAN Notices*, 53(2):693–707, March 2018.

[15] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2256–2272, May 2023.

[16] Amit Golander and Shlomo Weiss. Checkpoint allocation and release. *ACM Trans. Archit. Code Optim.*, 6(3):10:1–10:27, October 2009.

[17] Dibakar Gope and Mikko H. Lipasti. Bias-Free Branch Predictor. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 521–532, December 2014.

[18] Eric Hao, Po-Yung Chang, and Yale N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 228–232, New York, NY, USA, November 1994. Association for Computing Machinery.

[19] Jana Hofmann, Emanuele Vannacci, Cedric Fournet, Boris Kopf, and Oleksii Oleksenko. Speculation at fault: Modeling and testing microarchitectural leakage of CPU exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7143–7160, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/hofmann.

[20] Intel. Speculative Execution Side Channel Mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[21] Intel. Indirect Branch Predictor Barrier. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html, March 2018.

[22] Yu Jin, Pengfei Qiu, Chunlu Wang, Yihao Yang, Dongsheng Wang, Xiaoyong Li, Qian Wang, and Gang Qu. Overtake: Achieving Meltdown-type Attacks with One Instruction. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6, December 2023.

[23] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In *NDSS Symposium*, volume 2022, 2022. https://www.ndss-symposium.org/ndss-paper/auto-draft-247/.

[24] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *1991The 18th Annual International Symposium on Computer Architecture*, pages 34–42. IEEE Computer Society, January 1991.

[25] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1955–1972, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/katzman.

[26] Ofek Kirzner and Adam Morrison. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2399–2416, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/kirzner.

[27] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. The bimode++ branch predictor. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05)*, pages 8 pp.–, January 2005.

[28] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019.

[29] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018. https://www.usenix.org/conference/woot18/presentation/koruyeh.

[30] D. J. Lalja. Reducing the branch penalty in pipelined processors. *Computer*, 21(7):47–55, July 1988.

[31] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2137–2154, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi.

[32] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18,

pages 2109–2122, New York, NY, USA, October 2018. Association for Computing Machinery.

[33] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 633–649, September 2021.

[34] Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede. SpectrEM: Exploiting electromagnetic emanations during transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6293–6310, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/de-meulemeester.

[35] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You Cannot Always Win the Race: Analyzing mitigations for branch target prediction attacks. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 671–686, July 2023.

[36] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, November 1995.

[37] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752, May 2023.

[38] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, pages 205–217, New York, NY, USA, July 2023. Association for Computing Machinery.

[39] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS Symposium*, 2021.

[40] Pengfei Qiu, Qiang Gao, Dongsheng Wang, Yongqiang Lyu, Chang Liu, Xiaoyong Li, Chunlu Wang, and Gang Qu. PMU-Spill: Performance Monitor Unit Counters Leak Secrets in Transient Executions. In *2022 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6, December 2022.

[41] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: Exploiting and mitigating speculative race conditions. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6185–6202, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/ragab.

[42] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking ARM pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 685–698, New York, NY, USA, June 2022. Association for Computing Machinery.

[43] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I See Dead *M*ops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 361–374, June 2021.

[44] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse Engineering the Stream Prefetcher for Profit. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 682–687, September 2020.

[45] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 975–989, New York, NY, USA, November 2023. Association for Computing Machinery.

[46] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt, 2024.

[47] A. Seznec. Analysis of the O-GEometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405, June 2005.

[48] André Seznec. A 64 kbytes ISL-TAGE branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

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

[50] André Seznec. A new case for the TAGE branch predictor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 117–127, December 2011. https://ieeexplore.ieee.org/abstract/document/7851464.

[51] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction-Level Parallelism*, (8):1–23, 2006.

[52] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1753–1770, May 2023.

[53] Kevin Skadron, Margaret Martonosi, and Douglas Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism*, 2, 2000. http://www.jilp.org/vol2/v2paper1.pdf.

[54] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. *SIGARCH Comput. Archit. News*, 25(2):284–291, May 1997.

[55] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible Probe: Timing Attacks with PCIe Congestion Side-channel. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 322–338, May 2021.

[56] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 989–1007, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/tatar.

[57] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/trujillo.

[58] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1491–1505, May 2022.

[59] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 577–594, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/wiebing.

[60] Sander Wiebing and Cristiano Giuffrida. Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3599–3616. IEEE Computer Society, May 2025.

[61] Pawel Wieczorkiewicz. The AMD Branch (Mis)predictor Part 2: Where No CPU has Gone Before (CVE-2021-26341). https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before, March 2022.

[62] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/wikner.

[63] Johannes Wikner and Kaveh Razavi. Breaking the Barrier: Post-Barrier Spectre Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 89–89. IEEE Computer Society, November 2024.

[64] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting Decoder-detectable Mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 49–61, New York, NY, USA, December 2023. Association for Computing Machinery.

[65] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. Rendering contention channel made practical in web browsers. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3183–3199, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/wu-shujiang.

[66] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages*

and Operating Systems, Volume 3, volume 3 of *ASPLOS '24*, pages 770–784, New York, NY, USA, April 2024. Association for Computing Machinery.

[67] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, May 2023.

[68] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, September 1991. Association for Computing Machinery.

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

[70] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple M1 via hardware synchronization instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1973–1990, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong.

[71] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7267–7284, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-ruiyi.

[72] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 667–682, New York, NY, USA, March 2020. Association for Computing Machinery.

[73] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the instruction prefetcher. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7321–7337, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-bunnyhop.

A Cortex-A72: Determining Bias Status and Updating BST Records

Using the recorded data and the committed branch result, the BPU in Cortex-A72 determines the bias status of a branch via Alg. 2 and excludes footprints from biased indirect branches during PHR updates.

Algorithm 2: Determining the bias status of a branch.

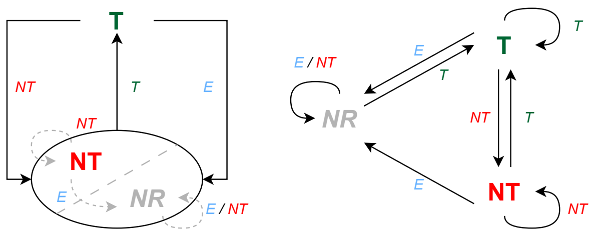
```
Data: outcome and addr of the comitted branch
Result: biased status of the commited branch
rec ← queryBST(addr);
if rec is NOT_FOUND then
    biased ← TRUE;
else
    if rec.biased is TRUE then
        biased ← (rec.outcome = outcome);
    else
        biased ← FALSE;
updateBST(addr, biased, outcome);
```

B Cortex-A76/A78AE: BTB/PHT Eviction and PC-Based Indexing

Since BTB and PHT may employ multiple indexing mechanisms, we investigate the extent of branch history's contribution to index generation on Cortex-A76 and A78AE. By modifying the branch history population parameters in mis-train snippets, we create scenarios where the BPU updates occur under branch history contexts that are different from the victim snippet. Interestingly, while these modified mis-train snippets no longer achieve out-of-place mis-training, they still trigger BTB eviction when reaching the previously identified threshold on Cortex-A78AE. This observation suggests that the BPUs in tested processors may employ both PC-based and history-based indexing schemes in different tables simultaneously.

C BTB/PHT eviction in canonical BHB

Never-taken branches in BHB. PHR implementations overlook undetected branches, effectively conflating "not recorded" and "not taken" states during speculative execution, while canonical BHB maintains a distinct "Not Recorded" default status for all branches and clearly differentiates between these states. Our analysis on A72 reveals that conditional branches in the canonical BHB remain unrecorded until their first taken execution. This observation aligns with AMD's explicit documentation that "global history is not updated for not-recorded branches" [3].



(a) PHR (b) Canonical BHB

Figure 8: Transitions in BPU record status for a branch, induced by relevant branch instances. From the BPU's perspective, the status of a branch can be classified into three types: (i) **"T"** for *taken*, (ii) **"NT"** for *not taken*, and (iii) **"NR"** for *not recorded*. Transitions occur based on the recorded outcome (NT or T) or due to eviction (E).

Based on these mechanisms, we can clearly identify three distinct states for conditional branches in the BHB updating process, as illustrated in Fig. 8b. In such a canonical BHB implementation, evicting a previously-taken branch's entry does not cause it to appear not-taken in subsequent speculations; rather, it resets the branch to its initial unrecorded state.

In the attack vectors previously discussed, when attempting to evict the record of the conditional branch `Bx_prime` after eviction, the BPU may base its prediction of `Bi_pred` on an entry associated with a third control flow path, distinctly different from both $\mathbb{F}_A$ and $\mathbb{F}_B$. This third path requires specific preparation strategies for successful exploitation. Moreover, this "not-recorded" state may persist until the branch is taken for the first time, creating a long-lasting effect on BHB updating mechanism that extends beyond a single speculation window.

Revealing *not-recorded* state. To demonstrate the impact of this explicit "not-recorded" state in BHB, we constructed an experiment that preserves the core setup from Section 6.2. In this experimental setup, we insert a speculation barrier between `Bx_prime` and `Bi_pred` to ensure all preceding branches are resolved and properly update the BHB before `Bi_pred` is speculated. `t_safe` is modified to emit a distinctive side-channel signalenabling clear differentiation from mis-speculation to `t_leak`. `Bx_prime` is executed as taken at least once prior to any training or testing sequences, ensuring its proper registration by the BPU. Following all training and eviction operations, we invoke a dedicated test flow in which `Bx_prime` is not taken and `Bi_pred` jumps to a third architectural target `t_arch`.

To assess potential mis-speculation of `Bi_pred`, we employ high-precision CPU timers (e.g., `rdtsc` on x86 processors and `mrs reg, pmccntr_el0` on ARM processors) to measure the branch latency of `Bi_pred`. When `Bi_pred` is correctly predicted to `t_arch`, we observe relatively low branch latency since no speculation rollback is required.

Evaluation. We evaluated this setup on Zen4. When eviction succeeded, `Bi_pred` consistently demonstrated correct prediction with minimal latency, with no detectable side-channel signals from either `t_leak` or `t_safe`. This indicates that following `Bx_prime` eviction from BTB/PHR, the test flow generates a unique BHB value and associates it with `t_arch`. Even when `Bx_prime` architecturally resolves as not taken and all preceding branch outcomes match $\mathbb{F}_A$, the BPU predicts `Bi_pred` using a third distinct state where `Bx_prime` is omitted due to its not-recorded classification. However, this implicit "bias" status handling for conditional branches differs significantly from the bias-free scheme observed for indirect branches on A72. Our analysis reveals that the canonical BHB implementation on A72 does not apply the bias-free scheme to conditional branches, which are always recorded in the 8-slot BHB even when consistently taken, following a distinctly different mechanism than that applied to indirect branches.

For conditional branch prediction, since only two possible predictions exist (taken or not taken), our experiments could not systematically demonstrate its perturbation effect. However, inducing an unexpected BHB value may force the BPU to make PC-based predictions using fallback prediction, resembling the phenomenon discussed in Section 7.

D Execution Traces of Chimera

The execution traces of training and attacks flows of the eBPF program used in Chimera (Section 7.3) are reported in Table 4.

Flow	**A**	**B**	**Attack**
Inputs	F F T	T T F	F T T
Line 2	NT	TT	NT
Line 3	TT	–	NT
Line 5	TT	–	NT
Split by BHB-shuffle branch on Line 7			
Line 8	NT	TT	–
Line 10	–	NT	–

Table 4: Architectural execution traces of training and attack flows. Three traces shown: Flow **(A)**, Flow **(B)**, and the attack flow. Input variables (take_sc, esc, set_ptr) are shown in the header, with "T" indicating TRUE and "F" indicating FALSE. For each branch, identified by line number, "TT" denotes taken, "NT" denotes not-taken, and "–" denotes not executed. Branch outcomes used to craft the malicious speculative execution path are highlighted in yellow.