# iEnFlow: Endogenous Control-Flow Attacks via Conditional Branch Prediction on Apple Silicon

Kaiyuan Rong*
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China
rky22@mails.tsinghua.edu.cn

Jiajie Chen*
Tsinghua University
Beijing, China
cjj21@mails.tsinghua.edu.cn

Junqi Fang
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China
fangjq24@mails.tsinghua.edu.cn

Peng Qu
Tsinghua University
Beijing, China
qp2018@tsinghua.edu.cn

Hanyin Liu
Harbin Institute of Technology
Harbin, China
xiulau@stu.hit.edu.cn

Youhui Zhang†
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China
zyh02@tsinghua.edu.cn

Dapeng Ju
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China
judapeng@tsinghua.edu.cn

Dongsheng Wang
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China
wds@tsinghua.edu.cn

## Abstract

Control-flow attacks have drawn increasing attention in microarchitectural security research due to their ability to expose sensitive data by revealing or manipulating program control-flow. Prior work has primarily focused on x86 architectures, with relatively few studies exploring such attacks on ARM-based Apple silicon processors. Meanwhile, existing microarchitectural side-channels that leak control-flow information on Apple silicon either rely on microarchitectural components beyond the branch predictor or lack a detailed understanding of branch predictor designs, which limits their generality and scalability.

In this paper, we present iEnFlow, the first endogenous and fine-grained control-flow attacks on Apple silicon that directly exploit control-flow information originating from the branch predictor itself. We target the conditional branch predictor (CBP) and reverse-engineer its internal design, including branch history length, hashing function, and predictor-table indexing scheme. Based on these reverse-engineering results, we develop primitives to read and write branch history and predictor-table entries, enabling unprivileged leakage and manipulation of the CBP on macOS. Using these primitives, we implement two attacks to demonstrate the effectiveness of iEnFlow. First, we perform a KASLR break attack, successfully leaking kernel addresses and bypassing mitigations deployed on macOS. Second, we demonstrate an out-of-place Spectre-PHT attack on Apple silicon. Our results show that control-flow leakage and speculative exploitation can originate purely from the branch prediction unit itself, rather than from side-effects in other microarchitectural components.

## CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**.

## Keywords

Apple silicon, Conditional Branch Predictor, KASLR, Spectre-PHT

**ACM Reference Format:**
Kaiyuan Rong, Jiajie Chen, Junqi Fang, Peng Qu, Hanyin Liu, Youhui Zhang, Dapeng Ju, and Dongsheng Wang. 2026. iEnFlow: Endogenous Control-Flow Attacks via Conditional Branch Prediction on Apple Silicon. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830454.3832569

## 1 Introduction

Over the past decade, a growing number of microarchitectural attacks have been reported. These attacks exploit various processor components, including caches [33, 56, 58, 62], branch prediction units (BPU) [11, 25, 59, 63], translation lookaside buffers (TLB) [13, 19, 44], and other microarchitectural buffers used for performance optimizations [6, 22, 23, 29, 48], to construct side-channels that leak sensitive information across privilege boundaries. As a result,

*Both authors contributed equally to this research.
†Corresponding author.


*CCS '26, The Hague, Netherlands*

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832569

attackers can extract cryptographic secret keys [6, 13, 33, 58, 63], bypass address space layout randomization (ASLR) [11, 19, 23], launch browser-based attacks [22–24], and so forth.

In recent years, microarchitectural attacks on Apple silicon have received increasing attention [3, 5, 6, 15–17, 19, 22–24, 28, 34, 42, 43, 47, 48, 56, 62]. Prior work has investigated a wide range of microarchitectural components in Apple silicon, including the data memory-dependent prefetcher (DMP) [6, 48], cache hierarchies [15, 42, 56, 62], the memory disambiguation unit (MDU) [28], and load address and value prediction mechanisms (LAP and LVP) [22, 23]. Based on reverse-engineering these components, various microarchitectural attacks have been conducted to successfully recover cryptographic keys [6, 15, 62], leak sensitive data from victim webpages in web browsers [22–24], and perform website fingerprinting [28, 56].

However, many existing attacks on Apple silicon focus on dataflow leakage [6, 15, 28, 48, 56], largely because most components that have been reverse engineered are located in the memory subsystem. In contrast, control-flow attacks remain underexplored. Existing control-flow attacks either (1) exploit microarchitectural components other than branch prediction units to indirectly leak control-flow information [28] or to perform control-flow hijacking via alternative speculative execution mechanisms [22, 23], or else (2) lack a detailed understanding of branch predictor designs [16, 17, 47], which prevents more fine-grained attacks. As a result, the scalability and generality of these attacks remain limited.

In this paper, we present iEnFlow, the first endogenous and fine-grained control-flow attacks on Apple silicon that exploit the conditional branch predictor (CBP). Here, *endogenous* means that the control-flow information exploited by our attacks originates from the branch predictor itself, rather than being indirectly inferred from other microarchitectural components (e.g., MDU [28], LAP [23], LVP [22]) relied upon by prior Apple silicon control-flow attacks. The fine-grained capability of iEnFlow stems from detailed knowledge of the CBP design. While attacks that exploit branch history information in predictors such as the CBP have been extensively studied on other architectures [2, 27, 52, 59, 60], they remain largely unexplored on Apple silicon. To enable iEnFlow, we reverse-engineer the CBP designs of both performance (P) cores and efficiency (E) cores on Apple M1–M4 processors. Building on prior work [60], we design and implement experimental methods specifically tailored for Apple silicon, and construct a diverse set of test cases. In particular, we observe that the branch history buffer (BHB) design of the E-cores differs from that of the P-cores and Intel architectures [60], posing a challenge for further investigation of the CBP. To address this challenge, we develop a novel testing technique that generalizes to reverse-engineering CBP designs on Apple silicon. Using our proposed methods, we recover key microarchitectural details, including the branch history length, the hash function, the pattern history table (PHT) mapping, and its associativity. These reverse-engineered results form the foundation of iEnFlow.

Based on our reverse-engineering results, we construct four unprivileged attack primitives that enable fine-grained observation and manipulation of the CBP's internal state. Since Apple silicon and macOS do not provide a high-precision unprivileged timing primitive by default [17], we adopt a counting thread [17, 34] as our timing measurement mechanism. Through experiments, we further demonstrate that the counting-thread timer can reliably distinguish the prediction outcomes of conditional branches, which serves as a key building block for the two read primitives.

Leveraging these attack primitives, iEnFlow enables two attacks that leak sensitive information across privilege boundaries. First, we demonstrate a practical Kernel Address Space Layout Randomization (KASLR) break on macOS. We propose a two-step approach that progressively eliminates invalid KASLR candidates. In the first step, we perform differential analysis on branch history by identifying and comparing two similar control-flow paths in the kernel, which allows us to rule out incorrect kernel base address candidates. In the second step, we construct precise PHT mappings to induce collisions with a target kernel branch, leveraging the recovered mapping function to infer the kernel branch address. By combining the results of these two steps, we successfully break KASLR on the latest version of macOS running on Apple M1–M4 processors, achieving over 95% accuracy while bypassing the existing macOS defense [18] against SysBumps [19], the only previously demonstrated KASLR-breaking attack on macOS.

Second, we demonstrate an out-of-place Spectre-PHT attack, in which PHT entries trained from user space influence branch prediction in the kernel. We construct a user-space conditional branch whose program counter (PC) and branch history collide with those of a target kernel branch in the PHT, causing both branches to map to the same PHT entry. We further employ Evict+Reload [14, 17] as a covert channel to leak sensitive data from kernel space. Our experimental results confirm the feasibility of cross-address-space PHT mistraining on both core types across all tested Apple M1–M4 processors. This provides further evidence that the PHT state is shared between user and kernel execution on the same core. In contrast to prior Spectre-PHT attacks on Apple silicon [17, 19, 24, 34], which rely on in-place training, our out-of-place Spectre-PHT attack relaxes the in-place training constraint, substantially expanding the set of potentially exploitable gadgets.

In summary, we make the following contributions in this paper:

- We design and implement experimental methodologies tailored for Apple silicon and reverse-engineer the CBP designs of both the P-cores and E-cores on Apple M1–M4 processors (cf. Section 3).
- We construct four unprivileged attack primitives that enable fine-grained observation and manipulation of the CBP's internal state entirely from user mode (cf. Section 4).
- We propose a two-step methodology, breaking KASLR on the latest version of macOS running on Apple M1–M4 processors with over 95% accuracy (cf. Section 5).
- We demonstrate the feasibility of out-of-place Spectre-PHT on Apple silicon, showing that user-space PHT entries can influence kernel branch prediction on both P-cores and E-cores across M1–M4. This confirms that the PHT is shared between user and kernel on each core (cf. Section 6).

## 2 Background

### 2.1 Branch Prediction

Branch prediction (BP) is a widely used mechanism in modern processors to improve instruction execution efficiency and overall

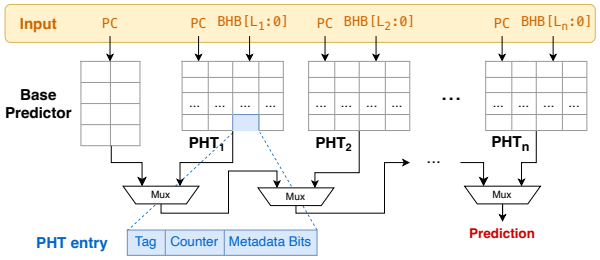


**Figure 1: Structure of TAGE conditional branch predictor. Each PHT is associated with a distinct branch history length, satisfying the ordering $L_1 < L_2 < \cdots < L_n$.**

throughput. The branch prediction unit (BPU) is typically located in the frontend of the processor pipeline and is responsible for predicting branch-related information for the fetched instruction stream. By anticipating the control-flow direction after the execution of the current instructions, branch prediction effectively alleviates control hazards in the pipeline, reducing stalls and delays caused by unresolved branches.

Modern BPUs are often highly sophisticated, as they must handle different types of branch instructions. On Intel processors, the BPU includes a branch target buffer (BTB) [11, 35, 63], which predicts the presence of branch instructions in the instruction stream, their types, and their target addresses. For conditional branches, the conditional branch predictor (CBP) [59, 60] predicts whether a branch will be taken. For indirect branches, the indirect branch predictor (IBP) [27] further predicts target addresses. Return instructions are handled separately using a return stack buffer (RSB) [26, 52].

## 2.2 TAGE

Within the BPU, the conditional branch predictor (CBP) is responsible for predicting whether conditional branches will be taken. One of the most influential CBP designs is TAGE (TAgged GEometric history length predictor) [41], originally proposed by André Seznec *et al.* in 2006 and subsequently extended with multiple variants [38–40]. The core idea of TAGE is to combine program counter (PC) information with branch histories of varying lengths, organized in a geometric progression, to dynamically predict the outcome of conditional branches.

A TAGE-based CBP consists of two main components. The first is the branch history buffer (BHB), which records branch history information derived from previously executed branches, typically through a hash function. The BHB is typically implemented as a shift register, where each update shifts the existing history and incorporates a footprint derived from the newly executed branch.[1] Depending on the type of branch history being recorded, this footprint can represent either the branch outcome, as in a Global History Register (GHR), or address-derived information from the branch instruction and target address, as in a Path History Register (PHR) [60]. The second component comprises multiple prediction tables (see Figure 1), including a base predictor and several pattern history tables (PHTs). The base predictor is indexed solely by the PC, while

[1] $\text{BHB}_{\text{new}} = (\text{BHB} \ll k) \oplus \text{footprint}$.

**Table 1: Evaluated Apple M-series processors.**

| Model | M1 | M2 | M3 | M4 |
|---|---|---|---|---|
| **Device** | MacBook Pro | Mac Mini | MacBook Air | Mac Mini |
| **P/E #Cores** | 4 / 4 | 4 / 4 | 4 / 4 | 4 / 6 |

**Table 2: CBP parameters on all tested Apple processors. $L_{\text{PHRT/B}}$ denotes the length of the PHRT/B, and $f_{\text{PHRT/B}}$ refers to the footprint in the PHRT/B update function.**

| Model | P-core | | E-core | |
|---|---|---|---|---|
| | M1 & M2 | M3 & M4 | M1 | M2 & M3 & M4 |
| $L_{\text{PHRT}}$ | 100 | 120 | 60 | 60 |
| $f_{\text{PHRT}}$ | T[31:2] | T[31:2] | T[48:2] | T[48:2] |
| $L_{\text{PHRB}}$ | 28 | 28 | 16 | 16 |
| $f_{\text{PHRB}}$ | B[5:2] | B[5:2] | B[5:2] | B[5:2] |
| **PC Input** | PC[18:2] | PC[14:2] | PC[17:2] | PC[18:2] |
| **PHT Assoc.** | 4 | 6 | 4 | 4 |

each PHT is organized as a set-associative structure indexed using a combination of the PC and a specific length of branch history stored in the BHB. Among these tables, $\text{PHT}_1$ corresponds to the shortest history length, whereas the last PHT ($\text{PHT}_n$) uses the longest history.

Each entry in the prediction tables contains a saturating counter that represents the predicted outcome of a conditional branch. During prediction, all tables are accessed in parallel to identify matching entries. If multiple PHTs contain matching entries, the prediction is determined by the counter associated with the entry indexed using the longest branch history. If no PHT entry matches, the prediction falls back to the base predictor. The multiplexers (mux) in Figure 1 select the prediction from the matching PHT with the longest branch history.

# 3 Reverse Engineering of Apple CBPs

In this section, we describe our investigation of the Conditional Branch Predictor (CBP) on Apple silicon. We assume that the CBP on Apple silicon is a variant of the TAGE structure, as mentioned in Apple's patent [1]. The branch history is stored in a buffer similar to the GHR or PHR. The CBP comprises multiple predictor-tables, each indexed using the conditional branch address (PC) together with branch history of varying lengths. All hash functions used in the CBP are based on XOR operations over multiple PC and branch-history bits.

All experiments are performed on multiple Apple M-series processors (see Table 1), spanning Apple M1 through M4. Both performance (P) cores and efficiency (E) cores are tested. In our experiments, we use *kperf* [57] to measure the execution results, specifically the overall misprediction rate of conditional branches for each test case.

Our reverse engineering methodology is derived from the approach used in Half&Half [60], which investigates Intel CBPs. We extend and refine this methodology in several aspects. In particular, we introduce an essential enhancement to address a challenge that does not arise on Intel processors, as detailed in Section 3.2. For brevity, the results presented in the following two sections are

based on experiments conducted on the M1 processor. Complete results for all tested processors and cores are provided in Table 2.

Section 3.1 identifies the hash function of the Branch History Buffer (BHB), which records the branch history and is used, in conjunction with the program counter (PC), to index the Pattern History Tables (PHTs). Then in Section 3.2, we characterize the PHT input function, i.e., the index and tag computation. Among all PHTs, we exclusively focus on the last table, which uses the longest branch history as input, since it has the highest priority in prediction. As we demonstrate, exploiting only the last table is sufficient to successfully mount the attacks we consider. We have also designed experiments to reverse engineer other tables, but noise is always inevitable. Therefore, for rigor and clarity, we present only the results for the last table. In the following sections, *PHT* denotes the last PHT unless stated otherwise.

## 3.1 Branch History in the CBP

First, we investigate the design of the Branch History Buffer (BHB) within the CBP. Half&Half [60] analyzes the BHB in three steps: determining its capacity, identifying the branch and target address bits used as hash inputs, and recovering the mapping from input bits to footprint positions. We propose an integrated experiment that combines these three steps to directly measure the contribution of each address bit to the BHB hash function. This integration avoids inaccuracies in estimating the BHB capacity, which can be unintentionally influenced by the choice of the training branch and target address, and enables a more direct and reliable reconstruction of the hash mapping.

**Branch Address in the BHB.** First, we measure the contribution of the branch address to the BHB. Figure 2(a) illustrates the experimental design. We generate two training branches, `tr0` and `tr1`, which differ by only one bit in their branch addresses but share the same target. The execution of these two branches depends on a random variable `k`. Specifically, when `k=0`, `tr0` is taken and `tr1` is not taken; when `k=1`, the outcomes are reversed. As a result, the execution of `tr0` and `tr1` may produce different BHB states, enabling us to determine whether and how branch addresses contribute to BHB updates. Before executing the training branches, we execute a branch chain to establish a consistent branch history (Line 2). After either training branch is taken, a certain number of dummy branches are executed to drain the training branch's history from the BHB (Line 9). Finally, a test branch is executed to verify whether the history has been fully drained, based on the resulting misprediction rate.

Figure 3(a) presents the results of the experiment described above on the P-cores. When the number of dummy branches is insufficient to fully drain the training branch from the BHB, the test branch can still observe the correlation between the random variable $k$ and the BHB state, resulting in almost no mispredictions. As the misprediction rate of the conditional branch `tr0` is 50%, the overall misprediction rate of conditional branches is 25%. In contrast, when the training branch is completely drained, the BHB value remains constant during the execution of the test branch, leading to a misprediction rate of approximately 50%.

The experimental results show that bits [5:2] of the branch address (denoted as B[5:2] in the following sections) form the input

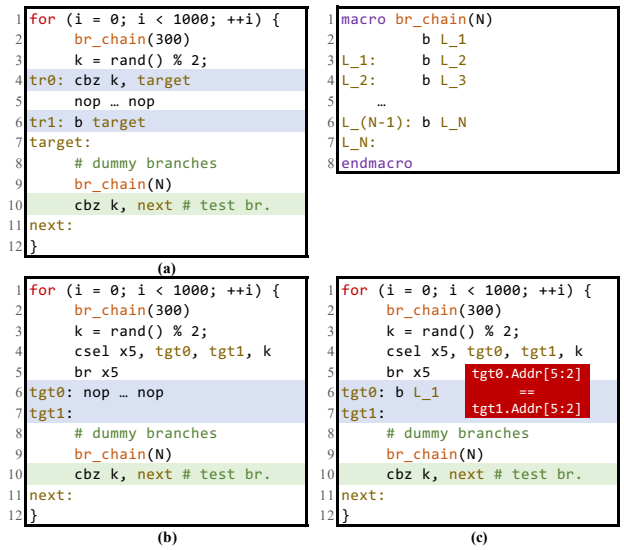


**Figure 2: The pseudo code that is used to disclose the branch history buffer structure within the CBP.**

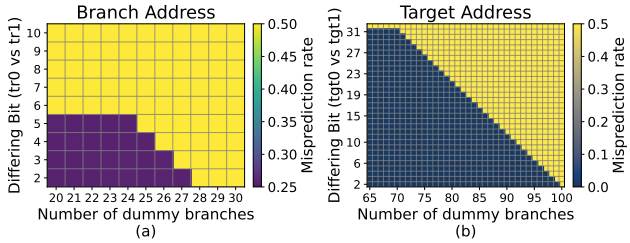


**Figure 3: Misprediction rate of conditional branches with varying numbers of dummy branches and differing branch and target address bits.**

footprint of the BHB update. By analyzing the transition points caused by different numbers of dummy branches, we further determine the relative position of each bit within the footprint and find that the BHB is shifted by one bit upon each update. The maximum number of dummy branches required to eliminate the training branch's influence indicates that the BHB can record up to 28 branch addresses.

This result also indicates that the BHB adopts a PHR-like structure, since the training branch can be drained by unconditional dummy branches (i.e., `br_chain(N)`). In the following sections, we refer to the branch history buffer as the PHR.

**Target Address in the BHB.** Next, we investigate the influence of the target address on the PHR. The experimental setup is shown in Figure 2(b), which is similar to that used for the branch address. The main difference is that we replace the two training branches with a single indirect branch that jumps to two different targets (`tgt0` and `tgt1`), whose addresses differ in one bit.

When the two targets differ in a higher-order address bit, the resulting address gap introduces a large number of nop instructions between them [60], causing enormous memory allocation and execution overhead. Meanwhile, executing these nops introduces significant noise into the measurement. We observe that a large number of nops can perturb the internal state of the CBP, preventing the predictor from maintaining the state established during training. To mitigate this, we redesign the experiment by inserting a branch instruction to skip over the gap between the two targets (Figure 2(c)). Concretely, we insert a branch at `tgt0` to jump to the second branch (`L_1`) among the dummy branches (Line 9). Since we have established that the PHR uses only bits [5:2] of the branch address in its hash function, this inserted branch does not produce an inconsistent branch history compared with executing nops, provided that the inserted branch has the same B[5:2] as the first dummy branch. This design eliminates the need to allocate and execute large regions of nop instructions, significantly reducing both memory overhead and measurement noise.

Experimental results (see Figure 3(b)) show that the PHR incorporates bits [31:2] of the target address (denoted as T[31:2] in the following sections) into its hash function and can record up to 100 target addresses on the P-cores of M1 processors.[2] Compared with the results for the branch address, we infer that the PHR employs two separate structures to record branch address history and target address history, rather than a single structure with a larger footprint. This hypothesis can be verified by the index and tag function. We refer to these two structures as PHRB and PHRT, respectively. The corresponding update functions are as follows:

$$\texttt{PHRB} = (\texttt{PHRB} << 1) \oplus \texttt{B[5:2]}$$
$$\texttt{PHRT} = (\texttt{PHRT} << 1) \oplus \texttt{T[31:2]}$$

**Other Branch Types.** We also examine other branch types, including direct and indirect jumps and calls, taken and non-taken conditional branches, as well as returns, to determine whether they are recorded in the PHR. All tested branch types are listed in Table 6 in Appendix D and are implemented as dummy branches in Figure 2. Our results show that all types of taken branches are recorded on M1–M3 processors and on the E-cores of the M4 processor. In contrast, on the P-cores of the M4, direct jumps (i.e., the b instruction) are not captured. Furthermore, taken conditional branches on the M4 P-cores sometimes fail to evict training branches in the PHR, suggesting that they are not consistently recorded in the PHR. We suspect that this behavior is caused by specific branch prediction mechanisms on the M4 P-cores [65]. We leave a detailed investigation of this behavior for future work.

## 3.2 Pattern History Table

In this section, we aim to reverse engineer the Pattern History Table (PHT) in the CBP, based on our prior knowledge of the PHRT and PHRB hash functions. Our reverse engineering process consists of two main stages. First, we develop a methodology based on a novel technique that allows us to construct conditional branches such that their PHRT, PHRB, and PC values can be precisely controlled, enabling manipulation of PHT inputs. Second, we apply this methodology to characterize the PHT organization, including the PC inputs, associativity, index hash function, and tag hash function. Together, these results provide a detailed understanding of the PHT structure on Apple silicon.

**Methodology.** All experiments presented in this section require constructing multiple conditional branches such that, when executed, each branch has a desired branch history state (PHRT and PHRB) and a predetermined program counter (PC). This controlled setup allows us to vary individual PHT inputs independently and observe how they affect the prediction behavior.

However, achieving such control introduces a unique challenge on the E-cores compared with the P-cores and Intel processors, because the PHRT on the E-cores incorporates almost all target-address bits into its hash function (see Table 2). This makes it difficult to establish a consistent history state across all tested branches when each tested branch uses its own branch-history construction code [27, 60], because branches with different targets inevitably produce different PHRT states. Moreover, using a shared code sequence to configure branch histories limits the flexibility of assigning different histories to different branches, and the final jump to the tested branch may further perturb the history state.

To address this challenge, we design a novel technique that allows setting arbitrary PHRT, PHRB, and program counter (PC) for a conditional branch, as illustrated in Figure 4. We allocate two memory regions, denoted as $R_0$ and $r_0$, to construct the branch history leading to the tested branch. The final two branches in this history are placed in separate memory regions, as they require independent control of their target addresses and the PC of the tested branch. Region $R_0$ is divided into multiple 256-byte blocks, while $r_0$ is divided into 128-byte blocks. The starting addresses of $R_0$ and $r_0$ are aligned to 4 GiB, and satisfy the relationship $Addr(R_0) = Addr(r_0) << 1$.

The core idea is to use two adjacent branches to cancel the contributions of target-address bits that are not intended to be controlled, while preserving the desired PHRT bits. On the E-cores, the first branch in the branch history jumps from $R_0$ to $r_0$, targeting either $r_0$+128 or $r_0$+132, depending on the desired PHRT[59]. The next branch jumps from $r_0$ back to $R_0$, targeting either $R_0$+256 or $R_0$+260, depending on PHRT[58]. Since $Addr(R_0) = Addr(r_0) << 1$, after these two branches are encoded into the PHRT, the contributions of ($r_0$+128) and ($R_0$+256) are eliminated, leaving only PHRT[59] and PHRT[58] in the PHRT state. The following blocks can be used to configure the remaining PHRT bits in the same pattern, eventually allowing all PHRT bits to be set.

Additionally, in our experiments, a specific PHRT bit may need to be set to the condition value $k$ of the tested conditional branch. This can be achieved easily by conditionally selecting the target address depending on the condition value, similar to the code snippet in Line 3-5 of Figure 2(c).

For the last two branches in the branch history, we must not only configure the desired PHRT[1:0] and PHRB[1:0] values, but also ensure that the execution reaches the tested branch located at the desired PC (i.e., `0xabcc` in Figure 4). To achieve this, we allocate another two 4-GiB-aligned memory regions, $R_1$ and $r_1$, and

[2] On the E-cores, our measurements confirm that bits T[46:2] participate in the PHR hash function. Due to macOS user-space address allocation constraints on our tested machines, however, addresses with bit 47 or higher set to 1 are inaccessible, preventing direct verification. In Section 5.3, however, we successfully create a PHT collision with a kernel conditional branch when creating branch history with T[48:2] as footprint. So it is reasonable to infer that T[48:2] is regarded as the footprint in the PHR update function.

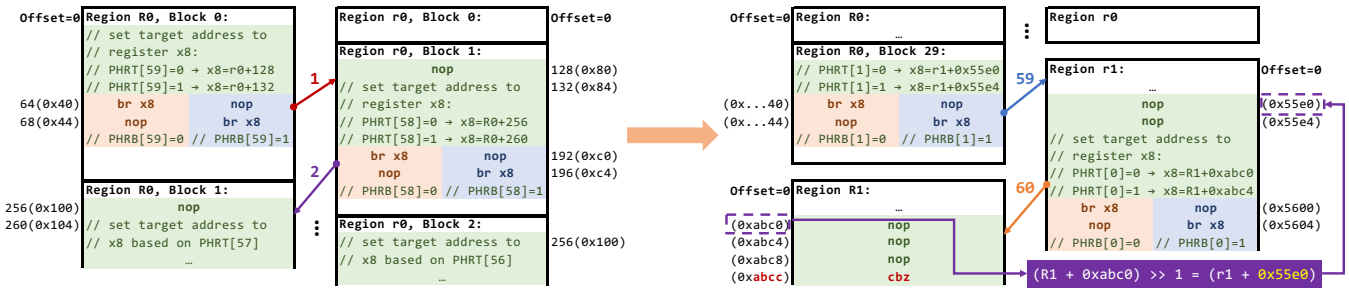


**Figure 4: The experimental design that is used to construct a conditional branch at a certain memory address (`0xabcc`) with arbitrary PHRT and PHRB values, using parameters of the CBP on the E-cores for illustration. The jumps labeled 1–60 are executed sequentially to configure the desired PHRT and PHRB values, where the final jump (No. 60) transfers control to a location near the desired PC of the tested branch. Please note that PHRB[≥ 16] are dummy values.**

place the tested branch in region $R_1$. The address of the tested conditional branch is computed as $Addr(R_1) + PC$. [3] Besides, we insert several `nop` instructions before the tested branch, and choose an 8-byte-aligned target address (`0xabc0`) used to set PHRT[0]. Then we calculate the target address used to set PHRT[1] by right shifting the target address used to set PHRT[0], and the resolved address is definitely within the region $r_1$ with a lower offset (`0x55e0`). Depending on PHRT[1] and PHRT[0], the two target addresses may be XORed with `0x4`. The above manipulation of addresses ensures following the same principle used to cancel the contributions of undesired bits in the PHRT.

Through this process, we can precisely configure arbitrary PHRT, PHRB, and PC values for the tested conditional branch. Moreover, this technique provides an additional benefit: it avoids executing a large number of `nop` instructions after setting the branch history and address [27, 60], thereby reducing noise and improving experimental efficiency.

**PC Inputs in the PHT.** First, we apply our proposed methodology to identify the PC bits used to index the PHT. We follow the same experimental design as Half&Half [60] and implement the experiments using our methodology, as shown in Figure 5(a). Specifically, we construct two tested conditional branches with identical PHRT and PHRB values, while their PCs differ in exactly one bit and their branch conditions are opposite. In addition, their PHRT[99] values (PHRT[59] on the E-cores) are set to the same condition value. We then measure the overall misprediction rate of the two conditional branches. The results show that when the differing PC bit lies within PC[18:2], the misprediction rate is close to zero; in contrast, when other PC bits differ, the misprediction rate increases significantly to approximately 50%. This indicates that the PHT uses PC[18:2] as part of its indexing function.

**Associativity.** Then we measure the associativity of the PHT. Instead of measuring the number of mispredicted branches with the same PHR but different PCs [60], we measure how many branches can be correctly predicted simultaneously (Figure 5(b)). If multiple branches are mapped to the same PHT set, they compete for the limited entries and cause prediction interference; therefore, the

[3] As described later, the PHT uses only a subset of the conditional branch address bits for indexing. Therefore, it is sufficient to ensure that the lower-order bits of the branch address satisfy the required alignment.

```
for (i = 0; i < 1000; ++i) {
    k = rand() % 2;
    /* jump to Region R0 */
    set_PHR_PC(k0..0, 0..0, 0x00)
    cbz k, next0
next0:
    /* jump to Region R0' */
    set_PHR_PC(k0..0, 0..0, 0x10)
    cbnz k, next1
next1:
}
// set_PHR_PC(PHRT, PHRB, PC)
```

(a)

```
for (i = 0; i < 1000; ++i) {
    k = rand() % 2;
    set_PHR_PC(k0..0, 0..0, 0x00)
    cbz k, next0
next0:
    set_PHR_PC(k0..0, 0..0, 0x20)
    cbz k, next1
next1:
    set_PHR_PC(k0..0, 0..0, 0x40)
    cbz k, next2
next2:
}
```

(b)

**Figure 5: The pseudo code that is used to discover the PC inputs and associativity in the PHT. The three parameters of `set_PHR_PC` represent the desired PHRT, PHRB, and PC values, respectively.**

maximum number of simultaneously predictable branches reveals the associativity.

To obtain this maximum number, we construct multiple conditional branches with identical PHRT and PHRB values but different PCs, and gradually increase the number of branches until misprediction occurs. We enumerate each PC bit, starting from this bit to generate different PC values for tested branches. This construction changes the selected PC bit and higher-order bits while keeping the lower-order bits identical. For example, from bit 5, we can generate `0x00`, `0x20`, `0x40`, `0x60`, and so forth. All tested branches are configured with consistent PHRT and PHRB values, and PHRT[99] is set to the condition value. Then, we measure the misprediction rate with different numbers of tested conditional branches.

The results are illustrated in Figure 6, which shows the maximum number of simultaneously predictable branches for each enumerated PC bit. We observe that when generating PCs from bit 6, almost sixteen branches can be correctly predicted. When generating from bit 4–5 and 7–9, almost eight branches can be correctly predicted. And for other bits, almost four branches can be correctly predicted. The larger numbers observed for specific PC bits indicate that varying these bits can cause branches to be distributed across multiple PHT sets. This suggests that these bits either directly affect the index hash function or allow the generated PC values to cover bits that contribute to the index. In contrast, when varying other bits, all branches are mapped to the same PHT set, and the maximum

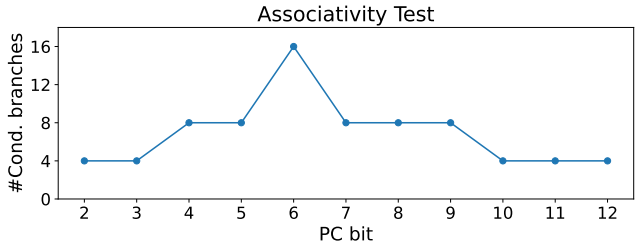


**Figure 6: The maximum number of simultaneously predictable conditional branches when generating PCs from each PC bit.**

number of simultaneously predictable branches therefore reveals the associativity of the PHT. From this observation, we can conclude that the associativity of the PHT is four, and PC[6] and PC[9] are involved in the index hash function, while other bits are only used in the tag function.

**Index Hash Function.** Next, we investigate the index hash function of the PHT. We divide this investigation into two steps, using a method similar to that in Half&Half [60]. First, we investigate which bits in PHRT and PHRB are used in the index hash function. We construct two groups of four conditional tested branches, each of which has the same PC[18:2], but the two groups differ in one distinct bit in the PHRT or PHRB. Also, PHRT[99] is set to k to ensure the prediction comes from the last PHT. Then we measure the misprediction rate of these eight branches. A significantly lower rate indicates the distinct bit is used as index bit.

Then we investigate which bits of the PHRT, PHRB, and PC contribute to the computation of a single index bit. We enumerate two bits involved in the index hash function, and also construct two groups of tested branches. In one group, the two bits are set to zero, while in the other group, the two bits are set to one. If the two bits are XORed in one index bit, the two groups point to the same PHT set, causing higher misprediction rate. In this way, we can obtain the concrete index hash function of the PHT. On the P-cores of M1, the PHT has a 10-bit index, and on the E-cores, the PHT has a 9-bit index. The detailed hash function is shown in Appendix E.

**Tag Hash Function.** Finally, we investigate the tag hash function of the PHT, which is not covered in Half&Half [60]. For PHRT, PHRB, and PC bits that are not used in the index hash function, we enumerate pairs of bits and test whether they are XORed into the tag hash function. Specifically, we use the experiment shown in Figure 5(a), configuring the two bits to zero for one tested branch and to one for the other. A significantly increased misprediction rate indicates that the two bits are XORed into the same tag bit, causing the two branches to collide in a single PHT entry.

However, this approach cannot be directly applied to bits that contribute to the index hash function. If one tested bit affects the index while the other does not, or if they map to different index bits, the two branches will be mapped to different PHT sets and therefore do not collide, regardless of their contribution to the tag hash. To overcome this limitation, we propose a new testing method to determine whether index bits and non-index bits are jointly XORed into the same tag bit. We select two index bits, denoted as $i_1$ and $i_2$, which correspond to the same index bit position, and two non-index bits, denoted as $t_1$ and $t_2$, and test whether the pairs $(i_1, t_1)$ and $(i_2, t_2)$ are XORed into any tag bit. For the first tested branch, all four bits are set to zero; for the second tested branch, all four bits are set to one. If such an XOR relationship exists, the two branches map to the same PHT entry and exhibit a high misprediction rate.

Combining the above results, we conclude that on both the P-cores and E-cores of the M1 processor, the PHT employs a 16-bit tag. The detailed tag hash functions are presented in Appendix F.

## 4 Attack Primitives

Based on our reverse-engineering results of Apple CBPs, this section presents four attack primitives that control and observe the state of the PHR (i.e., PHRT and PHRB) and the PHT. We first introduce the threat model in Section 4.1, including the attack scenario and the attacker's capabilities. We then describe the primitives for writing to and reading from the PHR and PHT under this threat model in Section 4.2, and evaluate two different measurements in Section 4.3.

### 4.1 Threat Model

We assume that the attack is performed on Apple silicon processors (M1–M4) running the stock macOS operating system. The attacker can execute arbitrary unprivileged code, but does not possess kernel or root privileges. The victim resides in a separate security domain and executes on the same core as the attacker. We further assume that the attacker can trigger the victim's execution, for example via system calls or function invocations, and that the victim's control-flow depends on secret data. The attacker aims to leak or speculatively influence this control-flow by exploiting the CBP.

### 4.2 Design of Attack Primitives

Before launching practical attacks, we first construct the necessary attack primitives as building blocks, including primitives for writing and reading the PHR and the PHT. Prior work in Pathfinder [59] designs and implements these primitives on Intel processors. In our work, we adopt a similar methodology to construct these primitives, with necessary adaptations to accommodate the design of Apple CBPs. Meanwhile, hardware performance counters (e.g., *kperf* [57]) and high-precision hardware timers, which are commonly used to measure branch mispredictions, are unavailable to unprivileged code on Apple silicon and macOS [17]. To tackle this challenge, we employ the counting-thread technique [17, 34] in both reading primitives to detect mispredictions of conditional branches. We further evaluate different measurements in Section 4.3.

**Write PHR.** The primitive for writing the PHR (i.e., PHRT and PHRB) aims to set the PHR to a desired value. The corresponding method is straightforward, since the PHR hash function has been reverse engineered in Section 3.1. We reuse the technique developed in our reverse-engineering experiments (cf. Figure 4) to construct the desired PHR state, as it is compatible with the PHR design on both the P-cores and E-cores. After executing 100 branches (P-cores) or 60 branches (E-cores), the PHR is set to the target value.

**Write PHT.** The primitive for writing the PHT aims to insert or manipulate a PHT entry such that it predicts a conditional branch as taken or not taken. Although the index and tag functions of the PHT have been exposed, directly executing a conditional branch whose

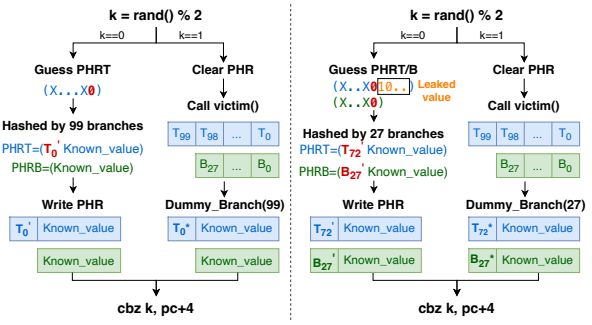


**Figure 7: Read both the PHRT and the PHRB. The left figure is used to read PHRT[0], while the right one is used to read PHRT[72] and PHRB[0] simultaneously on the P-cores of M1.**

PHR and PC match a given entry is insufficient to reliably insert that entry into the PHT, since we target the last table in the CBP. Our experiments show that repeatedly executing a conditional branch may still fail to allocate an entry in the last table, as entry allocation depends on the internal state of the CBP [41]. Fortunately, building on the same technique used in our reverse-engineering experiments (cf. Figure 4), we can reliably create a pair of entries in the last table, as the most significant bit of the PHRT representing the branch condition forces the last table to be used [47]. Because the last table uses the longest PHR and has the highest prediction priority, this primitive can influence a victim branch and speculatively hijack its control-flow even with only two allocated entries.

**Read PHR.** The primitive for reading the PHR aims to leak the values of both the PHRT and the PHRB. The methodology is similar to that of Pathfinder [59] and is illustrated in Figure 7. Figure 7(a) depicts the procedure for leaking PHRT[0] using a loop-based approach. In each iteration, we construct two different PHRT values for a tested conditional branch, depending on the condition value $k$. One PHRT value is generated based on a guess of PHRT[0], while the other is produced by the victim and then shifted by executing 99 dummy branches. This shifting moves PHRT[0] to the most significant bit, while all remaining PHRT and PHRB bits become known, as their contributions can be resolved from the addresses of the dummy branches. If the guess is correct, the two PHRT values are identical, resulting in a misprediction rate of approximately 50% for the tested branch; otherwise, the branch is predicted almost perfectly. This procedure can be repeated to leak PHRT[1], PHRT[2], and subsequent bits.

When leaking PHRT[72], PHRB[0] must be leaked simultaneously, since it remains in the PHRB when PHRT[72] is shifted to the most significant bit. Consequently, PHRB[0] must also be guessed in this case (see Figure 7(b)), requiring four executions to test all possible combinations of PHRT[72] and PHRB[0]. Using this approach, all bits in both the PHRT and the PHRB can be successfully leaked.

Compared with the method in Pathfinder [59], our approach applies two key adjustments. First, since the PHR is split into two separate buffers on Apple silicon, the PHRT and PHRB must be leaked in two steps (i.e., Figure 7(a) and (b)). Second, Pathfinder zero-shifts the PHR after victim execution by using dummy branches resolved to a zero footprint, resulting in a PHR of the following form ($T_0$ `0..0`). This is difficult to achieve on the E-cores, as the PHRT uses nearly all bits in the hash function. Instead, our design appends a fixed number of dummy branches and explicitly resolves their contributions to the PHRT, which are then written into the PHR accordingly.

**Read PHT.** The primitive for reading the PHT is designed to infer the state of a specific PHT entry, i.e., whether it currently predicts a branch as taken or non-taken. The approach is straightforward: we generate a conditional branch that aliases with the target PHT entry and observe whether this branch is predicted correctly. Such a branch can be constructed using the technique developed in our reverse-engineering experiments (cf. Figure 4), which allows us to precisely control both the PHR and the PC of the aliased branch. In addition, based on our experiments, we observe that PHT entries on all tested cores contain a 3-bit saturating counter. This observation is crucial for the effectiveness of our practical attacks.

## 4.3 Measurement of Branch Prediction

This section describes how we measure branch prediction outcomes under different scenarios. Section 4.2 proposes four attack primitives required in our practical attacks, two of which involve reading the PHR or PHT state and therefore rely on observing branch prediction behavior. In our reverse-engineering experiments, we obtain such information using *kperf* [57], which provides aggregated misprediction statistics for conditional branches. However, this approach is unsuitable for the attack setting, as it requires root privileges and violates our threat model. Consequently, alternative measurement techniques are necessary to infer branch prediction results without privileged access.

We attempt to employ timing primitives to infer branch prediction outcomes, as mispredicted branches incur higher execution latency [11]. We adopt a counting thread [17, 34] as a timing primitive, as illustrated in Figure 8, since the high-precision hardware cycle counter on Apple silicon is disabled by XNU in unprivileged contexts [17]. This mechanism introduces an additional thread that continuously increments a shared variable, thereby simulating a hardware cycle counter. The main thread reads this variable from shared memory to obtain timestamps and computes the relative execution time by subtracting two timestamps.

To evaluate the effectiveness of this timing primitive, we rerun the experiment used to determine the PHRT length (cf. Section 3.1), configuring the number of dummy branches to match the PHRT length. Under this configuration, the tested branch exhibits a misprediction rate of approximately 50%, meaning that it is correctly predicted and mispredicted with equal probability. Figure 9 presents the measured execution cycles of the tested branch over 1,000 iterations for each processor and core type. The results demonstrate a clear distinction between correctly predicted and mispredicted branches across all evaluated processors and cores. This demonstrates that the counting thread is effective for inferring branch prediction outcomes and is compatible with our unprivileged attacker model. Accordingly, we employ the counting thread as the timing primitive in our subsequent attacks.

```asm
mov x0, %[cnt_addr] # x0 = &counter
eor x1, x1, x1      # clear x1
loop:
    add x1, x1, 1   # x1 += 1
    str x1, [x0]    # counter = x1
    b loop
```

(a) counting thread

```asm
mov x0, %[cnt_addr]
ldr x1, [x0]   # pre-sampling
memory_barrier
cbz x8, pc+4   # tested branch
memory_barrier
ldr x2, [x0]   # post-sampling
sub x2, x2, x1 # x2 = exec. time
```

(b) main thread

**Figure 8: Counting thread.**

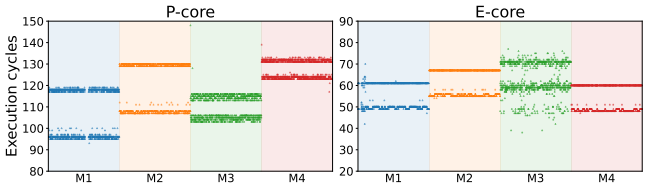


**Figure 9: Scatter plots showing execution cycles of the tested conditional branch using a counting thread on Apple M1–M4 P/E-cores.**

## 5 Breaking KASLR on macOS

Based on our proposed attack primitives, we present iEnFlow, endogenous control-flow attacks on Apple silicon. We demonstrate the capabilities of iEnFlow through two exploitation scenarios. The first breaks KASLR on macOS, illustrating the ability to leak kernel control-flow information from user space; this attack is described in this section. The second is an out-of-place Spectre-PHT attack, which shows that iEnFlow can manipulate the CBP to speculatively hijack kernel control-flow via out-of-place, cross-address-space training [4]. This attack is described in Section 6.

### 5.1 KASLR on macOS

Kernel Address Space Layout Randomization (KASLR) [51], derived from ASLR, is designed to protect the kernel against memory corruption attacks [8, 21, 37, 55]. The core idea of KASLR is to randomize kernel addresses at boot time, such that the locations of kernel code and data differ from those in the static kernel image. For the Linux kernel, KASLR was introduced in version 3.14 and is implemented by randomizing bits [29:21] of kernel virtual addresses, yielding 512 possible kernel base addresses [11, 51, 63].

Compared to Linux, the implementation of KASLR on macOS differs significantly. Starting from macOS 11, the XNU kernel and other essential drivers are packaged into a single Mach-O file that is loaded by the bootloader [10]. This Mach-O file is referred to as the *kernel collection*. KASLR is implemented using a random offset, known as the *slide*, which iBoot (Apple's bootloader) generates prior to loading the kernel collection. Once the kernel collection is loaded, its runtime base address is determined by adding the slide to the static base address specified in the kernel collection (i.e., `0xfffffe0007004000 + slide`).

The objective of iEnFlow is to break KASLR on macOS running on Apple silicon by leaking the kernel slide value from an unprivileged attacker operating in user mode. However, the implementation of iBoot is not publicly documented, and therefore the generation mechanism and range of the slide remain unknown. To address this limitation, we replicate the methodology in SysBumps [19]

**Table 3: Observed ranges of the normalized kernel slide index $s$ ($s = slide/\texttt{0x4000}$) and the corresponding guess ranges across Apple silicon.**

| Model | Index Range | | | Guess Range |
|---|---|---|---|---|
| | $s_{\min}$ | $s_{\max}$ | $\lvert\{s\}\rvert$ | $([s_{\max} - 32{,}768,\ s_{\min} + 32{,}768])$ |
| M1 | 8,249 | 40,954 | 32,706 | [8,186 , 41,017] |
| M2 | 45,204 | 77,903 | 32,700 | [45,135 , 77,972] |
| M3 | 45,340 | 77,846 | 32,507 | [45,078 , 78,108] |
| M4 | 45,238 | 77,941 | 32,704 | [45,173 , 78,806] |

to empirically characterize the slide. Specifically, we reboot each tested machine 1,500 times and record the slide value observed at each boot. The results are summarized in Table 3. Our findings closely match those reported in SysBumps. From the 1,500 collected samples, we observe an approximately uniform distribution of slide values, all of which are multiples of `0x4000`. We therefore normalize the slide by defining a discrete index $s = slide/\texttt{0x4000}$. Across all samples, the observed values of $s$ span approximately 32,768 distinct integers between the minimum and maximum. Based on this observation, we define a guess range, which represents the set of candidate kernel slide indices considered during the attack, over the normalized index $s$. Specifically, let $s_{\min}$ and $s_{\max}$ denote the minimum and maximum values of $s$ observed in our measurements; we set the guess range to $[s_{\max} - 32{,}768,\ s_{\min} + 32{,}768]$. Each guess of the kernel slide is then obtained by multiplying a candidate index in this range by `0x4000`.

### 5.2 Methodology

In this section, we present our methodology for breaking KASLR on macOS. Building on our attack primitives in Section 4.2, a seemingly direct approach would be to observe the PHR state corresponding to a kernel control-flow path (e.g., during a system call), and to resolve the KASLR slide by enumerating candidate slides and matching the resulting PHR hashes against the observed state. However, this approach is impractical, as it relies on extracting dynamic kernel control-flow at instruction-level granularity. To the best of our knowledge, there exists no technique that enables such precise extraction on Apple silicon. The hardware and software stack of Apple silicon and macOS is largely closed [7, 61, 64], and lacks publicly accessible mechanisms for tracing executed instructions or taken control-flow edges within the kernel. Consequently, reconstructing the complete and precise set of dynamically executed branch instructions required by this approach is currently infeasible in practice.

Instead of directly extracting the entire control-flow, we adopt a two-step approach to progressively narrow down the space of candidate KASLR slide values, as illustrated in Figure 10. The first step, called *PHRT-based differential analysis*, leverages the difference between two PHRT values obtained from distinct kernel control-flow executions to eliminate a large fraction of incorrect slide candidates through differential analysis. While this step alone cannot uniquely determine the correct slide value, it is highly effective at ruling out incorrect candidates. The second step, called *PHT collision*, exploits collisions in a PHT entry between a user-space conditional branch and a kernel branch to directly probe kernel addresses. Based on

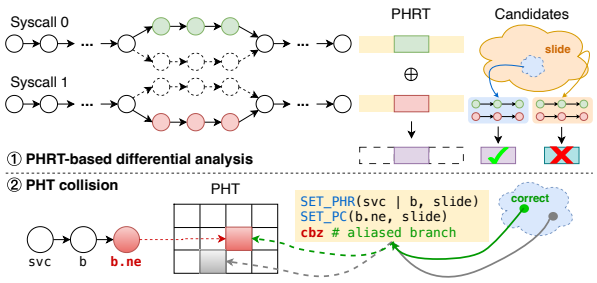


**Figure 10: A two-step approach to break KASLR on macOS.**

the significantly reduced candidate set produced by the first step, this second step is able to uniquely identify the correct slide value.

**PHRT-based Differential Analysis.** In the first step, we derive a set of potential KASLR slide candidates through PHRT-based differential analysis. We observe that the execution paths of several system calls (syscalls) in the kernel are highly similar, with differences occurring only in the portion that executes the syscall handler, while the rest of the control-flow is shared. Therefore, if two syscalls execute the same number of branch instructions within their handlers, their respective PHRT values become *aligned*. In this case, performing a bitwise XOR of the two PHRT values yields the hash values of the branch target addresses in the handler at their corresponding positions. Since extracting the branch instructions of syscall handlers from the kernel image is straightforward, we can enumerate all possible KASLR slide candidates and retain those whose computed hash values match the PHRT differential.

**PHT Collision.** The second step uniquely determines the slide value through a PHT collision. The PHT mapping function encodes substantial address information, such that when a user-space conditional branch and a kernel branch map to the same PHT entry, their corresponding PHR and PC values must satisfy specific constraints, enabling inference of the kernel branch address. Leveraging this observation, we can further rule out incorrect slide candidates. The procedure is as follows. First, we select a kernel branch and enumerate all possible slide values. For each candidate, we construct a user-space branch whose PHR and PC values are derived from the kernel branch address computed under the candidate slide, such that the two branches collide in the PHT. After executing the kernel branch, if a change in the state of the corresponding PHT entry is observed, the slide candidate is considered potentially correct. As proved in Section 5.3, by combining this step with the PHRT-based differential analysis, the slide value can be uniquely determined on each tested machine.

## 5.3 Evaluation

We now evaluate the performance of iEnFlow on Apple silicon using our proposed methodology. All tested machines, ranging from Apple M1 to M4, run macOS 26.2 (build version 25C56), the latest version at the time of writing.

**PHRT-based Differential Analysis.** In the first step, we reduce the space of KASLR slide candidates by performing differential analysis on the PHRT values obtained after executing two system calls. We select `getegid` and `sys_getdtablesize` as the target syscalls,

**Table 4: Time cost and overall accuracy of breaking KASLR on macOS. 'Time/syscall' denotes the time required to read the PHR for a single system call, '#Slides' denotes the average number of slide candidates remaining after Step 1, and 'Time/slide' denotes the time required to test one slide candidate by inducing a PHT collision. 'Total Time' denotes the expected total time estimated from the time cost of each step and the average number of remaining slide candidates.**

| Core | Step 1 | | Step 2 | Overall | Total |
|---|---|---|---|---|---|
| | Time/syscall | #Slides | Time/slide | Accuracy | Time |
| M1 (P) | 0.321s | 259.53 | 0.0128s | 98.1% | 3.96s |
| M1 (E) | 1.077s | 259.53 | 0.0161s | 97.2% | 6.33s |
| M2 (P) | 0.317s | 205.91 | 0.0054s | 95.5% | 1.75s |
| M2 (E) | 0.656s | 205.91 | 0.0164s | 96.6% | 4.69s |
| M3 (E) | 0.691s | 206.48 | 0.0203s | 99.7% | 5.57s |
| M4 (E) | 0.707s | 150.21 | 0.0163s | 99.8% | 3.86s |

as their kernel execution paths differ only at three branch instructions, while the remaining control-flow is shared. We extract the target addresses of these six branch instructions from the kernel image. Using our primitive for reading the PHR, we obtain the corresponding PHRT values for the two syscalls and compute their bitwise XOR to derive the PHRT differential. We then enumerate all possible slide candidates. For each candidate, we resolve the six branch target addresses under the enumerated slide, compute their hash values, and compare the resulting hash against the observed PHRT differential to determine whether the candidate is a potentially correct slide.

In our experiments, PHR values can be obtained from either the P-cores or the E-cores, and we observe that the resulting PHR differentials are consistent across the two core types, since the [48:32] bits of the target addresses of the six branches are the same and unaffected by KASLR. However, on the P-cores of the M4, PHR values are unstable: repeated executions produce different PHR values. We hypothesize that the observed PHR instability on the P-cores may be explained by our earlier observation in Section 3.1 that not all taken branch instructions are consistently recorded in the PHR. Fortunately, this limitation does not affect our attack, as reliable results can still be obtained on the E-cores. Table 4 reports the time required to read the PHR for each syscall. According to the table, reading the PHR takes approximately one second on each model and core type. We later use software simulation to estimate the average number of slide candidates remaining after this step, which determines the expected total time required in the next step.

**PHT Collision.** In the second step, we determine the final slide value by constructing a PHT collision. We select a conditional branch in the kernel and enumerate a slide candidate to derive its PC and PHR, which are then used to construct a user-space branch that aliases with the kernel branch in the PHT. Crucially, the PHR of the selected kernel branch must be readily obtainable, as constructing a PHT collision requires precise knowledge of the PHR value of the kernel branch. To this end, we choose the second branch instruction executed after the `svc` instruction[4], specifically

[4]On Apple silicon, system calls are invoked via the `svc` instruction, which transfers execution from user space to the kernel.

a `b.ne` instruction. The partial control-flow immediately following the `svc` instruction can be directly obtained from the kernel image, which allows the PHR of this branch to be computed from the target addresses of the preceding two branches.

For each slide candidate, we construct a user-space conditional branch with the same PHR value and the same lower 20 bits of the PC as the selected kernel branch. We then use our primitive for writing the PHT to train the corresponding PHT entry such that the condition of the user-space branch is opposite to that of the kernel branch. Since the PHT entry uses a 3-bit saturating counter, repeatedly executing the kernel branch eight times is sufficient to flip the prediction state of the PHT entry. Finally, we probe the state of this PHT entry using the user-space branch to determine whether a collision has occurred.

In our experiments, we observe that on the P-cores of the Apple M3 and M4, PHT collisions cannot be reliably triggered even when the enumerated slide value is correct. We hypothesize that this behavior arises from additional branch prediction mechanisms introduced on these cores, which affect the prediction behavior of the CBP for conditional branches [65]. Consequently, we conduct this step exclusively on the E-cores, where reliable PHT collisions can be consistently observed. For M1 and M2 processors, we perform this step on both the P-cores and E-cores, independently inferring the slide values on each core type. Table 4 reports the time required to test a PHT collision for a single slide candidate.

**Results.** We evaluate the performance of our KASLR break on four Apple processors. For each machine, we reboot the system 10 times. After each reboot, we execute the attack 100 times, yielding 100 inferred slide values per reboot. We then measure the overall success rate across all 1,000 attack attempts, as shown in Table 4, demonstrating the effectiveness and efficiency of our method. Incorrect slides arise either when incorrect candidates satisfy the inferred constraints, or when noise and microarchitectural instability prevent a valid candidate from being identified. Moreover, the success of our attack demonstrates that the PHR and PHT are shared between user space and the kernel.

**Uniqueness of Slide Solution.** Our method iteratively eliminates invalid slide candidates, but it is important to ensure that this process ultimately yields a unique solution rather than multiple candidates. To verify this, we implement a software simulator that, for each possible slide value, reproduces the attack procedure by computing the resulting PHRT differential and the corresponding PHT entry index and tag for the kernel branch, based on our reverse-engineering results. We then check whether two or more slide values produce the same PHRT differential and PHT entry index and tag. The simulation results show that, for all six cores where stable PHT collisions can be observed, our method yields a unique solution in every case.

Furthermore, for each slide candidate, we calculate the number of slides remaining after applying the PHRT differential (i.e., the number of candidates that would need to be tested via a PHT collision) and take the average across all slides. This provides an estimate of the average number of slide candidates that must be tested during the PHT collision procedure. The results are also summarized in Table 4. Therefore, the expected overall execution time can be estimated as the time for two PHR reading operations plus the time for testing the remaining slide candidates through PHT collisions after Step 1. The worst-case execution time is approximately 6.33 seconds on the M1 E-core, calculated as $2 \times 1.077\text{s} + 259.53 \times 0.0161\text{s}$, and all tested processors complete the attack within this time.

```
// x is attacker-controllable
if (x < array1_size) {
    y = array2[array1[x] * 4096]
}
```
(a)

```
// almost_false is not controllable
if (almost_false) {
    y = array2[array1[x] * 4096]
}
```
(b)

**Figure 11: Two Spectre-PHT speculative gadgets: (a) the branch condition is attacker-controllable, (b) the branch condition cannot be influenced by the attacker.**

Notably, without the first step, relying solely on PHT collisions to identify the final slide can still reduce the candidate space to two even in the worst case, as indicated by our software simulation. However, this approach requires testing more than 30,000 candidates (see Table 3), resulting in prohibitively long execution time. In contrast, the first step filters out a large number of incorrect candidates, significantly shrinking the search space, while requiring substantially less execution time. Therefore, we adopt a two-step approach to efficiently break KASLR.

# 6 Out-of-place Spectre-PHT on Apple Silicon

## 6.1 Spectre Attacks

Spectre attacks represent a prominent class of microarchitectural attacks that exploit transient execution to leak sensitive data across privilege boundaries. Following the disclosure of the Spectre-v1 and Spectre-v2 variants [25], the research community has proposed a wide range of Spectre variants targeting various microarchitectural components and processors of different vendors [2, 12, 36, 46, 49, 50, 52–54]. According to the taxonomy introduced by Claudio Canella *et al.* [4], Spectre attacks can be categorized based on their mistraining strategies, which characterize how the branch predictor is manipulated:

(1) **In-place vs. out-of-place**: whether the attacker trains the branch predictor using the same branch address as the victim branch (in-place), or an aliased branch at a different address (out-of-place).
(2) **Same-address-space vs. cross-address-space**: whether the training branch and the victim branch that triggers transient execution reside in the same address space or across different address spaces.

Compared to out-of-place training, Spectre-PHT (also known as Spectre-v1), which trains in-place, suffers from several inherent limitations. First, in the attack scenario where user-space code attempts to mistrain kernel-space branches, in-place training is fundamentally infeasible, since the user and kernel address spaces do not overlap and therefore cannot share identical branch addresses. Second, some in-place Spectre-PHT attacks relied on "self-training" using kernel branches directly, which imposes strict requirements on speculative gadgets: the branch condition must be controllable by user input (see Figure 11(a)). This constraint excludes many otherwise exploitable gadgets whose branch conditions are not directly influenced by user-controlled data (Figure 11(b)), resulting in missed attack surfaces.

Leveraging our reverse-engineering analysis of Apple CBPs, we present the second attack of iEnFlow: an out-of-place Spectre-PHT attack on Apple silicon. By exploiting the primitive of writing the PHT described in Section 4.2, the attacker can inject entries into the PHT, thereby directly influencing kernel branch predictions. This manipulation induces transient execution in the kernel and ultimately enables information leakage.

## 6.2 Implementation and Evaluation

In this section, we implement and evaluate the Spectre-PHT attack, demonstrating the feasibility of a cross-domain, out-of-place Spectre-PHT attack on Apple silicon. To this end, we manually inject a Spectre-PHT gadget into the XNU kernel using a kernel extension (kext), and expose this gadget to user-space programs via a `sysctl` interface. In our attack model, the attacker is an unprivileged user-space program that triggers the transient execution of this gadget through `sysctl` calls, thereby leaking kernel data. The branch condition of the gadget is not controllable by the attacker. Moreover, since the attacker cannot construct a branch at the same address as the kernel branch, we adopt an out-of-place training strategy. In addition, because macOS does not permit user-space programs to execute data cache flush instructions by default, we employ an Evict+Reload [14, 17] covert channel to recover the leaked information. Specifically, we allocate a sufficiently large memory region and evict cache lines by repeatedly loading from this region to implement the eviction primitive. Through empirical evaluation, we determine the optimal parameter configuration. For our kext implementation, we disable the Privileged Access Never (PAN) mechanism, allowing kernel code to directly access the user-mode reload buffer.

We evaluate the performance of our out-of-place Spectre-PHT attack on Apple M1–M4 processors, across both the P-cores and the E-cores. The results, summarized in Table 5, report the attack success rate and the leakage bandwidth. From the experimental results, we observe that our out-of-place Spectre-PHT mistraining succeeds on both the P-cores and the E-cores across all tested Apple M1–M4 processors. This demonstrates that PHT entries trained in user space can be used by kernel branches for branch prediction. However, the attack exhibits relatively low accuracy on some cores and non-negligible variability across repeated executions. Our evaluation primarily aims to establish the feasibility of out-of-place Spectre-PHT attacks on Apple silicon, rather than demonstrating a fully practical exploitation. We leave improving the robustness of this proof-of-concept, for example through more effective cache covert channels [20, 31], and extending the attack toward practical scenarios to future work.

It is worth noting that the knowledge of the underlying mapping mechanism of the CBP is essential for constructing out-of-place Spectre-PHT attacks, particularly on the E-cores. On these cores, the PHRT hashes nearly the full instruction address bits, rendering simple alignment of low-order branch address bits or randomized collision attempts largely ineffective [47]. While prior work conducted by Lorenz Hetterich *et al.* [17] also demonstrates Spectre attacks on Apple silicon, those attacks rely on in-place training using victim branches. In contrast, our out-of-place Spectre-PHT approach offers improved extensibility and generality.

**Table 5: Accuracy and leakage rates of out-of-place Spectre-PHT across different Apple silicon processors.**

| Model | Accuracy | | Leakage rate (Bytes/s) | |
|---|---|---|---|---|
| | P-core | E-core | P-core | E-core |
| M1 | 90.64% | 76.52% | 1029.86 | 436.82 |
| M2 | 81.63% | 87.77% | 598.39 | 450.74 |
| M3 | 97.64% | 68.17% | 75.26 | 42.69 |
| M4 | 99.92% | 95.14% | 47.85 | 904.34 |

# 7 Discussion

## 7.1 Possible Mitigations

**PHT Isolation.** A straightforward mitigation strategy is to enforce execution-domain isolation in the PHT. Specifically, each PHT entry can be augmented with an additional field indicating the current execution domain (e.g., user mode vs. kernel mode). A PHT entry would then only be used to predict conditional branches originating from the corresponding execution domain. Such hardware-based isolation mechanisms have already been adopted in the branch prediction units of other processors (e.g., Intel and AMD) [12, 27], and are therefore theoretically sound and effective.

However, even in the presence of execution-domain isolation, the PHT remains physically shared. As a result, it is still possible to infer kernel branch behavior via PHT-based Prime+Probe [33] attacks. Based on our software simulation results, replacing the PHT collision used for breaking KASLR with a PHT-based Prime+Probe attack remains effective in significantly reducing the candidate space, with the worst case leaving only 56 candidates on the M3 E-cores.

**Sanitization of the PHR.** In addition to isolating the PHT, the two PHR buffers encode a substantial amount of sensitive information. Consequently, flushing the PHR upon transitions between different privilege domains represents another viable mitigation strategy. Such a defense can effectively mitigate cross-privilege-domain leakage and injection enabled by our PHR read and write attack primitives. However, branch history information collected across privilege domains also contributes to branch prediction accuracy. As a result, flushing the PHR on every privilege transition may incur non-negligible performance overhead [2].

**Memory Barrier.** For Spectre-PHT attacks, inserting speculation barriers around sensitive conditional branches can prevent unintended transient execution [45]. Similarly, isolating the PHT across execution domains can also effectively mitigate out-of-place, cross-address-space Spectre-PHT attacks.

## 7.2 Comparison with Related Work

In this work, iEnFlow successfully demonstrates both breaking KASLR and mounting out-of-place Spectre-PHT attacks on macOS. Compared to similar previous work, SysBumps [19] leverages TLB-based side-channels and speculative execution to break KASLR on macOS running on Apple M1 and M2 processors; however, this attack has been mitigated by Apple [18]. In contrast, our KASLR-breaking attack remains effective on the latest macOS releases at the time of writing, successfully bypassing Apple's defenses deployed against SysBumps, and continues to work on newer platforms,

including M3 and M4. MistleTunnel [30] exploits syscall-induced TLB traces on Apple M-series processors, but limited TLB index coverage restricts its leakage to the lower eight bits of the KASLR offset. In contrast, iEnFlow completely recovers all KASLR random bits, enabling a full KASLR bypass.

Other work [47] conducted by Adam Tuby *et al.* attempts to realize out-of-place Spectre attacks on Apple M1 by reverse-engineering the PHR length and combining a proposed LPC primitive with brute-force techniques, achieving user-to-kernel Spectre attacks on the P-cores (Firestorm) but failing on the E-cores (Icestorm). We conjecture that this limitation arises because, on the E-cores, nearly the full target address participates in the PHRT hashing function, making it practically infeasible to construct aliased branches via brute force. In contrast, our work performs an in-depth reverse engineering of Apple CBPs and introduces techniques for arbitrary PHR writes on the E-cores. As a result, we successfully achieve out-of-place Spectre attacks on the E-cores of the M1 as well as on all core types across Apple M2–M4 processors.

Several prior works have explored control-flow microarchitectural attacks on Apple silicon. Lorenz Hetterich *et al.* [17] port Spectre-v1 and Spectre-v2 to Apple silicon, overcoming practical challenges such as the absence of unprivileged high-precision timers and cache maintenance instructions on Apple silicon. However, their evaluation of Spectre-PHT is limited to same-address-space, in-place training scenarios. Due to the lack of reverse engineering of branch prediction components, it is unable to realize out-of-place training. Conjuring [16] exploits the PHT to construct speculative fetch attacks that leak control-flow information, but similarly does not reverse engineer the CBP, preventing fine-grained attacks. In addition, iLeakage [24] and PACMAN [34] also rely on in-place Spectre-PHT to trigger speculative execution. As their primary focus lies elsewhere, these works do not investigate control-flow microarchitectural components in depth, and consequently do not enable out-of-place Spectre-style attacks.

Beyond control-flow attacks, several works exploit memory-related microarchitectural components on Apple silicon. Augury [48] and GoFetch [6] leverage the data memory-dependent prefetcher (DMP) to leak data at rest and cryptographic keys. ARMeD channels [28] exploit the memory disambiguation unit (MDU) to construct a covert channel and perform website fingerprinting attacks. SLAP [23] and FLOP [22] utilize speculative execution induced by load address predictor (LAP) and load value predictor (LVP), respectively, to leak sensitive data from the victim's webpage. Cache-based side-channel attacks [15, 42, 56, 62] have also been demonstrated on Apple silicon. These memory-related attacks typically leak memory access traces, from which sensitive information can be inferred. In contrast, iEnFlow directly exploits the CBP to leak sensitive information through control-flow behavior.

Our work builds most directly on two prior studies of Intel's conditional branch predictor. Half&Half [60] presented the first comprehensive reverse engineering of Intel CBPs, recovering the PHR structure, PHT indexing functions, and associativity across multiple microarchitectures. Our reverse-engineering methodology (Section 3) follows their experimental paradigm. Pathfinder [59] subsequently developed attack primitives for reading and writing the PHR and PHT on Intel processors, which we adapt in Section 4.2. Relative to these prior works, iEnFlow makes four distinct contributions. First, we uncover Apple's split-PHR design (PHRB for branch addresses, PHRT for target addresses), which fundamentally differs from Intel's unified PHR. Second, we develop the technique (Figure 4) that enables arbitrary PHRT-PHRB-PC configuration on the E-cores, despite the challenge that nearly all target-address bits participate in hashing, which is absent on Intel processors. Third, we recover the PHT tag hash function, which Half&Half did not investigate. Fourth, we demonstrate a practical KASLR break and an out-of-place Spectre-PHT attack on Apple silicon.

### 7.3 Limitations and Future Work

Our work has several limitations. We focus exclusively on the user-to-kernel attack scenario. Other scenarios, such as user-to-user or cross-process attacks, may also present exploitable attack surfaces. In addition, our KASLR-breaking attack on macOS relies on control-flow analysis techniques to leak kernel address information. These techniques depend on specific kernel characteristics, such as similar system call control-flows, which limits their applicability. Exploring more attack scenarios and developing more robust analysis methods remain important directions for future work.

## 8 Conclusion

In this paper, we present iEnFlow, the first endogenous and fine-grained control-flow attacks on Apple silicon that directly exploit the conditional branch predictor (CBP). By reverse-engineering the CBP designs of both performance (P) and efficiency (E) cores across Apple M1–M4 processors, we uncover key microarchitectural details and develop unprivileged attack primitives that enable precise observation and manipulation of branch prediction behavior from user mode. Leveraging these primitives, we demonstrate two attacks across privilege boundaries: a KASLR break on the latest version of macOS and an out-of-place Spectre-PHT attack that generalizes beyond prior in-place training assumptions. Our findings reveal that branch prediction structures on Apple silicon expose a previously underexplored attack surface for control-flow leakage and speculative execution attacks.

## Acknowledgments

We thank the anonymous reviewers for their insightful comments and constructive feedback, which helped us improve the clarity and presentation of this work.

## References

[1] Muawya M Al-Otoom, Ian D Kountanis, and Conrado Blasco. 2020. Branch prediction system. US Patent 10,719,327.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 971–988. https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[3] Zechao Cai, Jiaxun Zhu, Wenbo Shen, Yutian Yang, Rui Chang, Yu Wang, Jinku Li, and Kui Ren. 2023. Demystifying Pointer Authentication on Apple M1. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2833–2848. https://www.usenix.org/conference/usenixsecurity23/presentation/cai-zechao

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association,

Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[5] Nikhil Chawla, Chen Liu, Abhishek Chakraborty, Igor Chervatyuk, Thais Moreira Hamasaki, Ke Sun, and Henrique Kawakami. 2024. Uncovering Software-Based Power Side-Channel Attacks on Apple M1/M2 Systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 305, 6 pages. doi:10.1145/3649329.3656531

[6] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1117–1134. https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru

[7] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 749–763. doi:10.1145/3460120.3484564

[8] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1093–1110. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng

[9] DeepSeek. 2026. DeepSeek. https://www.deepseek.com. Accessed: 2026-08-06.

[10] Dortania. 2020. What's new in macOS 11, Big Sur! — dortania.github.io. https://dortania.github.io/hackintosh/updates/2020/11/12/bigsur-new.html#new-kernel-cache-system-kernelcollections. Accessed: 2026-01-13.

[11] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783743

[12] Jean-Claude Graf, Sandro Ruegge, Ali Hajiabadi, and Kaveh Razavi. 2026. VMSCAPE: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments . In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1710–1727. doi:10.1109/SP63933.2026.00046

[13] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[15] Gregor Haas, Seetal Potluri, and Aydin Aysu. 2021. iTimed: Cache Attacks on the Apple A10 Fusion SoC. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 80–90. doi:10.1109/HOST49136.2021.9702290

[16] Ali Hajiabadi and Trevor E. Carlson. 2024. Conjuring: Leaking Control Flow via Speculative Fetch Attacks. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. doi:10.1145/3649329.3655895

[17] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different - Spectre Attacks on Apple Silicon. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lorenzo Cavallaro, Daniel Gruss, Giancarlo Pellegrino, and Giorgio Giacinto (Eds.). Springer International Publishing, Cham, 116–135.

[18] Apple Inc. 2025. About the security content of macOS Sequoia 15.2. https://support.apple.com/en-us/121839. Accessed: 2026-01-12.

[19] Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 64–78. doi:10.1145/3658644.3690189

[20] Tom Kessous and Niv Gilboa. 2024. Prune+PlumTree - Finding Eviction Sets at Scale. In *2024 IEEE Symposium on Security and Privacy (SP)*. 3754–3772. doi:10.1109/SP54263.2024.00173

[21] Mahmood Jasim Khalsan and Michael Opoku Agyeman. 2018. An Overview of Prevention/Mitigation against Memory Corruption Attack. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control* (Stockholm, Sweden) *(ISCSIC '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 6 pages. doi:10.1145/3284557.3284564

[22] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 2595–2614. https://www.usenix.org/conference/usenixsecurity25/presentation/kim-jason

[23] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3549–3566. doi:10.1109/SP61157.2025.00098

[24] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2038–2052. doi:10.1145/3576915.3616611

[25] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. doi:10.1109/SP.2019.00002

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

[27] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. 2024. Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2137–2154. https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi

[28] Chang Liu, Zhouyang Li, Haixia Wang, Pengfei Qiu, Gang Qu, and Dongsheng Wang. 2025. Exploiting ARMeD Channels by Reverse Engineering ARM Memory Disambiguation Unit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/TCAD.2025.3585078

[29] Chang Liu, Dongsheng Wang, Yongqiang Lyu, Pengfei Qiu, Yu Jin, Zhuoyuan Lu, Yinqian Zhang, and Gang Qu. 2024. Uncovering and Exploiting AMD Speculative Memory Access Predictors for Fun and Profit. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 31–45. doi:10.1109/HPCA57654.2024.00014

[30] Hanyin Liu, Jin Wu, Rihui Sun, Kaiyuan Rong, Hongpei Zheng, Yun Chen, Jian Dong, and Dongsheng Wang. 2026. MistleTunnel: Attacking KASLR on Apple M-Series Silicon with a Novel TLB Side Channel. In *Proceedings of the 63rd ACM/IEEE Design Automation Conference (DAC)*. Association for Computing Machinery, Long Beach, CA, USA. doi:10.1145/3770743.3804016

[31] Bradley Morgan, Gal Horowitz, Sioli O'Connell, Stephan van Schaik, Chitchanok Chuengsatiansup, Daniel Genkin, Olaf Maennel, Paul Montague, Eyal Ronen, and Yuval Yarom. 2025. Slice+Slice Baby: Generating Last-Level Cache Eviction Sets in the Blink of an Eye. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3479–3496. doi:10.1109/SP61157.2025.00264

[32] OpenAI. 2026. ChatGPT. https://chatgpt.com. Accessed: 2026-08-06.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, David Pointcheval (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[34] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 685–698. doi:10.1145/3470496.3527429

[35] Kaiyuan Rong, Junqi Fang, Haixia Wang, Dapeng Ju, and Dongsheng Wang. 2026. OCCUPY+PROBE: Cross-Privilege Branch Target Buffer Side-Channel Attacks at Instruction Granularity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2026.230925

[36] Sandro Rüegge, Johannes Wikner, and Kaveh Razavi. 2025. Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 2615–2631. https://www.usenix.org/conference/usenixsecurity25/presentation/ruegge

[37] Majid Salehi, Luca Degani, Marco Roveri, Danny Hughes, and Bruno Crispo. 2023. Discovery and Identification of Memory Corruption Vulnerabilities on Bare-Metal Embedded Devices. *IEEE Transactions on Dependable and Secure Computing* 20, 2 (2023), 1124–1138. doi:10.1109/TDSC.2022.3149371

[38] André Seznec. 2007. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007), 1–6.

[39] André Seznec. 2011. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 117–127. doi:10.1145/2155620.2155635

[40] André Seznec. 2014. TAGE-SC-L Branch Predictors. In *Proceedings of the 4th Championship Branch Prediction*. Minneapolis, United States. https://inria.hal.science/hal-01086920

[41] André Seznec and Pierre Michaud. 2006. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism* 8 (Feb. 2006), 23. https://inria.hal.science/hal-03408381

[42] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2863–2880. https://www.usenix.org/conference/

usenixsecurity21/presentation/shusterman
[43] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and Arm SoCs. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6275–6292. https://www.usenix.org/conference/usenixsecurity23/presentation/taneja
[44] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 989–1007. https://www.usenix.org/conference/usenixsecurity22/presentation/tatar
[45] The Linux Kernel Documentation Team. 2026. Spectre Side Channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html#kernel-mitigation. Accessed: 2026-01-12.
[46] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7303–7320. https://www.usenix.org/conference/usenixsecurity23/presentation/trujillo
[47] Adam Tuby and Adam Morrison. 2025. Reverse Engineering the Apple M1 Conditional Branch Predictor for Out-of-Place Spectre Mistraining. arXiv:2502.10719 [cs.CR] https://arxiv.org/abs/2502.10719
[48] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1491–1505. doi:10.1109/SP46214.2022.9833570
[49] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. 2024. InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 577–594. https://www.usenix.org/conference/usenixsecurity24/presentation/wiebing
[50] Sander Wiebing and Cristiano Giuffrida. 2025. Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3599–3616. doi:10.1109/SP61157.2025.00253
[51] Wikipedia contributors. 2025. Address space layout randomization — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Address_space_layout_randomization&oldid=1318513566. [Online; accessed 5-January-2026].
[52] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3825–3842. https://www.usenix.org/conference/usenixsecurity22/presentation/wikner
[53] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3516–3533. doi:10.1109/SP61157.2025.00089
[54] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/3613424.3614275
[55] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1187–1204. https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei
[56] Tianhong Xu, Aidong Adam Ding, and Yunsi Fei. 2025. EXAM: Exploiting Exclusive System-Level Cache in Apple M-Series SoCs for Enhanced Cache Occupancy Attacks. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1294–1308. doi:10.1145/3708821.3710844
[57] YaoYuan. 2021. kpc_demo.c: A demo shows how to read Intel or Apple M1 CPU performance counter in macOS. GitHub Gist. Available at: https://gist.github.com/ibireme/173517c208c7dc333ba962c1f0d67d12.
[58] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom
[59] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. 2024. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 770–784. doi:10.1145/3620666.3651382
[60] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. 2023. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1220–1237. doi:10.1109/SP46215.2023.10179415
[61] Tingting Yin, Zicong Gao, Zhenghang Xiao, Zheyu Ma, Min Zheng, and Chao Zhang. 2023. KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5039–5054. https://www.usenix.org/conference/usenixsecurity23/presentation/yin
[62] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. 2023. 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)*. USENIX Association, Anaheim, CA, 1973–1990. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong
[63] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. BunnyHop: Exploiting the Instruction Prefetcher. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7321–7337. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-bunnyhop
[64] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. 2024. CrossFire: Fuzzing macOS Cross-XPU Memory on Apple Silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3749–3762. doi:10.1145/3658644.3690376
[65] Yuhui Zhu and Alessandro Biondi. 2025. Exploiting Inaccurate Branch History in Side-Channel Attacks. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 2519–2538. https://www.usenix.org/conference/usenixsecurity25/presentation/zhu-yuhui

# A Ethical Considerations

All experiments presented in this paper, including the CBP reverse engineering and attack evaluations, were conducted exclusively on machines owned and fully controlled by the authors. The study does not involve experimentation on third-party systems, production environments, or devices without explicit authorization. No experiments were performed against real users, and no private, sensitive, or personal data were accessed, collected, or exfiltrated at any stage. The purpose of the experimental evaluation is solely to understand the security properties and potential risks of the studied mechanisms, not to exploit them for malicious or unethical purposes.

**Responsible Disclosure.** We have already reported our findings to Apple's Security Team. Apple acknowledged our findings and confirmed that the reported behavior is expected.

# B Open Science

The artifact associated with this work contains the code, scripts, and supporting files required to reproduce and evaluate the key contributions of this paper, including the reverse engineering code presented in Section 3, the implementation of the KASLR-breaking attack described in Section 5, and the Spectre-PHT proof-of-concept (PoC) presented in Section 6. This artifact enables researchers to reproduce our analysis and experimental results. We have released our artifact at https://github.com/CPU-Security/iEnFlow and https://zenodo.org/records/20769810.

# C Generative AI Usage

We used ChatGPT [32] to assist with minor grammar checking and language polishing for improving the readability of the manuscript. During the artifact evaluation phase, we used an agent based on the DeepSeek [9] large language model to assist in refactoring manually written code to improve readability and maintainability. All AI-assisted modifications were manually reviewed and verified by the authors.

# D Types of Branches Recorded in the PHR

In Section 3.1, we examine various types of branches, checking whether they can be recorded in the PHR. The detailed branch types and their corresponding instructions are shown in Table 6.

| Branch Types | Instructions |
|---|---|
| Direct jump | `b` |
| Indirect jump | `br` |
| Conditional branch (taken) | `cbnz`, `cbz`, `b.cond`, `tbnz`, `tbz` |
| Conditional branch (non-taken) | `cbnz`, `cbz`, `b.cond`, `tbnz`, `tbz` |
| Direct call | `bl` |
| Indirect call | `blr` |
| Return | `ret` |

**Table 6: Tested types of branches.**

# E The Index Function of the Pattern History Table

## E.1 P-cores

Based on our reverse-engineering results in Section 3.2, the PHT index hash function on the P-cores of the Apple M1 and M2 is given below:

(1) PHRT[99] ⊕ PHRT[48] ⊕ PHRT[7]
(2) PHRT[93] ⊕ PHRT[43] ⊕ PHRT[2]
(3) PHRT[88] ⊕ PHRT[38] ⊕ PC[9]
(4) PHRT[83] ⊕ PHRT[33] ⊕ PHRB[25]
(5) PHRT[78] ⊕ PHRT[27] ⊕ PHRB[20]
(6) PHRT[73] ⊕ PHRT[22] ⊕ PHRB[15]
(7) PHRT[68] ⊕ PHRT[17] ⊕ PHRB[10]
(8) PHRT[63] ⊕ PHRT[12] ⊕ PHRB[5]
(9) PHRT[58] ⊕ PHRT[53] ⊕ PHRB[0]
(10) PC[6]

The PHT index hash function on the P-cores of the Apple M3 and M4 is given below:

(1) PHRT[117] ⊕ PHRT[111] ⊕ PHRT[54]
(2) PHRT[116] ⊕ PHRT[59] ⊕ PHRB[1]
(3) PHRT[106] ⊕ PHRT[49] ⊕ PHRT[11]
(4) PHRT[101] ⊕ PHRT[44] ⊕ PHRT[6]
(5) PHRT[97] ⊕ PHRT[40] ⊕ PHRT[2]
(6) PHRT[92] ⊕ PHRT[35]
(7) PHRT[87] ⊕ PHRT[30] ⊕ PHRB[25]
(8) PHRT[82] ⊕ PHRT[25] ⊕ PHRB[20]
(9) PHRT[78] ⊕ PHRT[21] ⊕ PHRB[16]
(10) PHRT[73] ⊕ PHRT[16] ⊕ PHRB[11]
(11) PHRT[68] ⊕ PHRT[63] ⊕ PHRB[6]
(12) PC[6]

## E.2 E-cores

The PHT index hash function on the E-cores of the Apple M1 is given below:

(1) PHRT[59] ⊕ PHRT[28] ⊕ PHRB[3]
(2) PHRT[55] ⊕ PHRT[25] ⊕ PHRB[0]
(3) PHRT[52] ⊕ PHRT[22] ⊕ PHRT[3]
(4) PHRT[49] ⊕ PHRT[19] ⊕ PHRT[0]
(5) PHRT[46] ⊕ PHRT[15] ⊕ PC[8]
(6) PHRT[43] ⊕ PHRT[12] ⊕ PHRB[15]
(7) PHRT[40] ⊕ PHRT[9] ⊕ PHRB[12]
(8) PHRT[37] ⊕ PHRT[6] ⊕ PHRB[9]
(9) PHRT[34] ⊕ PHRT[31] ⊕ PHRB[6]

The PHT index hash function on the E-cores of the Apple M2 is given below:

(1) PHRT[59] ⊕ PHRT[27] ⊕ PHRB[0]
(2) PHRT[55] ⊕ PHRT[24] ⊕ PHRT[3]
(3) PHRT[52] ⊕ PHRT[20] ⊕ PC[11]
(4) PHRT[48] ⊕ PHRT[17] ⊕ PC[8]
(5) PHRT[45] ⊕ PHRT[13] ⊕ PHRB[13]
(6) PHRT[41] ⊕ PHRT[10] ⊕ PHRB[10]
(7) PHRT[38] ⊕ PHRT[6] ⊕ PHRB[6]
(8) PHRT[34] ⊕ PHRT[31] ⊕ PHRB[3]
(9) PC[6]

The PHT index hash function on the E-cores of the Apple M3 is given below:

(1) PHRT[57] ⊕ PHRT[52] ⊕ PHRT[22]
(2) PHRT[56] ⊕ PHRT[25] ⊕ PHRB[0]
(3) PHRT[49] ⊕ PHRT[19] ⊕ PHRT[0]
(4) PHRT[46] ⊕ PHRT[16]
(5) PHRT[43] ⊕ PHRT[12] ⊕ PHRB[15]
(6) PHRT[40] ⊕ PHRT[9] ⊕ PHRB[12]
(7) PHRT[37] ⊕ PHRT[6] ⊕ PHRB[9]
(8) PHRT[34] ⊕ PHRT[3] ⊕ PHRB[6]
(9) PHRT[31] ⊕ PHRT[28] ⊕ PHRB[3]
(10) PC[6]

The PHT index hash function on the E-cores of the Apple M4 is given below:

(1) PHRT[59] ⊕ PHRT[53] ⊕ PHRT[22]
(2) PHRT[57] ⊕ PHRT[50] ⊕ PHRT[20]
(3) PHRT[56] ⊕ PHRT[25] ⊕ PHRB[2]
(4) PHRT[47] ⊕ PHRT[17] ⊕ PHRT[0]
(5) PHRT[44] ⊕ PHRT[14]
(6) PHRT[42] ⊕ PHRT[11]
(7) PHRT[39] ⊕ PHRT[9] ⊕ PHRB[13]
(8) PHRT[36] ⊕ PHRT[6] ⊕ PHRB[10]
(9) PHRT[33] ⊕ PHRT[3] ⊕ PHRB[8]
(10) PHRT[31] ⊕ PHRT[28] ⊕ PHRB[5]
(11) PC[6]

# F The Tag Function of the Pattern History Table

## F.1 P-cores

Based on our reverse-engineering results in Section 3.2, the PHT tag hash function on the P-cores of the Apple M1 and M2 is given below:

(1) PC[2]
(2) PC[3]
(3) PC[4]
(4) PC[5]
(5) PC[7] ⊕ PHRT[0:96:12] ⊕ PHRB[8,21]
(6) PC[8] ⊕ PHRT[1:97:12] ⊕ PHRB[9,22]
(7) PC[9] ⊕ PHRT[2:98:12] ⊕ PHRB[10,23,24]

(8) PC[10] ⊕ PHRT[3:99:12] ⊕ PHRB[11,12,25]
(9) PC[11] ⊕ PHRT[4:88:12] ⊕ PHRB[0,13,26]
(10) PC[12] ⊕ PHRT[5:89:12] ⊕ PHRB[1,14,27]
(11) PC[13] ⊕ PHRT[6:90:12] ⊕ PHRB[2,15]
(12) PC[14] ⊕ PHRT[7:91:12] ⊕ PHRB[3,16]
(13) PC[15] ⊕ PHRT[8:92:12] ⊕ PHRB[4,17]
(14) PC[16] ⊕ PHRT[9:93:12] ⊕ PHRB[5,18]
(15) PC[17] ⊕ PHRT[10:94:12] ⊕ PHRB[6,19]
(16) PC[18] ⊕ PHRT[11:95:12] ⊕ PHRB[7,20]

In these tag functions, the expression PHRT[start:end:step] denotes PHRT[start] ⊕ PHRT[start+step] ⊕ ··· ⊕ PHRT[end]. PHRB[x,y,z] denotes PHRB[x] ⊕ PHRB[y] ⊕ PHRB[z].

The PHT tag hash function on the P-cores of the Apple M3 and M4 is given below:

(1) PC[2]
(2) PC[3]
(3) PC[4]
(4) PC[5]
(5) PC[7] ⊕ PHRT[0:112:8] ⊕ PHRB[0,9,18,27]
(6) PC[8] ⊕ PHRT[1:113:8] ⊕ PHRB[1,10,19]
(7) PC[9] ⊕ PHRT[2:114:8] ⊕ PHRB[2,11,20]
(8) PC[10] ⊕ PHRT[3:115:8] ⊕ PHRB[3,12,21]
(9) PC[11] ⊕ PHRT[4:116:8] ⊕ PHRB[4,13,22]
(10) PC[12] ⊕ PHRT[5:117:8] ⊕ PHRB[5,14,23,24]
(11) PC[13] ⊕ PHRT[6:118:8] ⊕ PHRB[6,15,16,25]
(12) PC[14] ⊕ PHRT[7:119:8] ⊕ PHRB[7,8,17,26]

## F.2 E-cores

The PHT tag hash function on the E-cores of the Apple M1 is given below:

(1) PC[2]
(2) PC[3]
(3) PC[4]
(4) PC[5]
(5) PC[6] ⊕ PHRT[0:48:12] ⊕ PHRB[6]
(6) PC[7] ⊕ PHRT[1:49:12] ⊕ PHRB[7]
(7) PC[8] ⊕ PHRT[2:50:12] ⊕ PHRB[8]
(8) PC[9] ⊕ PHRT[3:51:12] ⊕ PHRB[9]
(9) PC[10] ⊕ PHRT[4:52:12] ⊕ PHRB[10]
(10) PC[11] ⊕ PHRT[5:53:12] ⊕ PHRB[11,12]
(11) PC[12] ⊕ PHRT[6:54:12] ⊕ PHRB[0,13]
(12) PC[13] ⊕ PHRT[7:55:12] ⊕ PHRB[1,14]
(13) PC[14] ⊕ PHRT[8:56:12] ⊕ PHRB[2,15]
(14) PC[15] ⊕ PHRT[9:57:12] ⊕ PHRB[3]
(15) PC[16] ⊕ PHRT[10:58:12] ⊕ PHRB[4]
(16) PC[17] ⊕ PHRT[11:59:12] ⊕ PHRB[5]

The PHT tag hash function on the E-cores of the Apple M2, M3, and M4 is given below:

(1) PC[2]
(2) PC[3]
(3) PC[4]
(4) PC[5]
(5) PC[7] ⊕ PHRT[0:48:12] ⊕ PHRB[6]
(6) PC[8] ⊕ PHRT[1:49:12] ⊕ PHRB[7]
(7) PC[9] ⊕ PHRT[2:50:12] ⊕ PHRB[8]
(8) PC[10] ⊕ PHRT[3:51:12] ⊕ PHRB[9]
(9) PC[11] ⊕ PHRT[4:52:12] ⊕ PHRB[10]
(10) PC[12] ⊕ PHRT[5:53:12] ⊕ PHRB[11,12]
(11) PC[13] ⊕ PHRT[6:54:12] ⊕ PHRB[0,13]
(12) PC[14] ⊕ PHRT[7:55:12] ⊕ PHRB[1,14]
(13) PC[15] ⊕ PHRT[8:56:12] ⊕ PHRB[2,15]
(14) PC[16] ⊕ PHRT[9:57:12] ⊕ PHRB[3]
(15) PC[17] ⊕ PHRT[10:58:12] ⊕ PHRB[4]
(16) PC[18] ⊕ PHRT[11:59:12] ⊕ PHRB[5]