2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

SSBleed: Non-speculative Side-channel Attacks via Speculative Store Bypass on Armv9 CPUs

Chang Liu[†], Hongpei Zheng[†], Xin Zhang[‡], Dapeng Ju[†], Dongsheng Wang[†(✉)], Yinqian Zhang[§], Trevor E. Carlson[¶]

[†]Tsinghua University, {cliu21, zhenghp23}@mails.tsinghua.edu.cn, {judapeng, wds}@tsinghua.edu.cn
[‡]Peking University, zhangxin00@stu.pku.edu.cn
[§]Southern University of Science and Technology, yinqianz@acm.org
[¶]National University of Singapore, tcarlson@comp.nus.edu.sg

***Abstract*—Modern CPUs employ Speculative Store Bypass (SSB) to reduce load latency and improve performance. In response to transient attacks such as Spectre, CPU vendors have also introduced mitigations to prevent incorrect speculation from leaking data. In this work, we show that the SSB on Armv9 CPUs introduces a previously unexplored form of non-speculative data leakage. Specifically, we find that the SSB on Armv9 performance cores is governed by an undocumented predictor. Through reverse engineering, we uncover the design of this predictor and show that it lacks isolation across security domains. Furthermore, existing mitigations such as SSBS are insufficient to prevent leaks. Based on this, we present SSBleed, the first non-speculative side-channel attack via SSB on Armv9 CPUs. We validate the practicality of SSBleed through five case studies, including cross-process RSA signature and key generation attacks on the latest version of MbedTLS and WolfSSL, interrupt detection, and improved data transmission in two transient attacks. Finally, we propose a flush-based mitigation through a kernel patch, which incurs an average performance overhead of 0.46%.**

I. INTRODUCTION

Memory accesses have become one of the primary bottlenecks in modern CPUs, especially in high-performance CPU architectures that support out-of-order and speculative execution. To mitigate performance loss due to load stalls, various techniques have been introduced, including address prediction [2], [18], [38], value prediction [17], [37], [43], and memory dependence prediction [19], [40], [52]. Among them, Speculative Store Bypassing (SSB) is a key optimization, which allows a load to be speculatively executed before a preceding store completes, and exposes additional parallelism in the CPU pipeline. SSB has been widely adopted in modern CPUs [21], [47], [48], [56], [59], including recent Armv9 CPUs used in servers [7] and mobile performance cores [6].

On the other hand, SSB has also raised security concerns. Under misprediction, a load may obtain stale data from an address before a preceding store overwrites it [3], [33], or receive the value of a store targeting a different address [4]. Although the CPU will resolve such mispredictions to preserve correct program behavior, these mispredictions can leave microarchitectural traces, enabling transient attacks [11], [48].

(✉) Corresponding author.

To mitigate such attacks enabled by SSB, mainstream CPU vendors have implemented several hardware mitigations. In contrast to x86 CPUs that disable SSB entirely, known as SSB Disable (SSBD), Arm CPUs mitigate SSB attacks by preventing speculative data forwarding of loads during SSB rather than blocking it statically, named SSB Safe (SSBS) [5]. However, it is unclear whether the latter approach may still leave security risks beyond speculative execution. Particularly, if the SSB behavior is controlled by a predictor, similar to a branch predictor [22], [42], [82], [83], the internal state of this predictor may leak non-speculative changes persistently across security boundaries, without relying on speculative execution that has been already mitigated. However, the implementation details of SSB are largely undocumented, and such non-speculative leaks have remained underexplored. This raises two fundamental and unresolved questions: *Does the SSB expose an attack surface at the non-speculative level? If so, can existing defenses prevent such kind of attacks?*

In this work, we address these questions by studying the details of SSB on Armv9 CPUs. We uncover a previously undocumented Memory Dependency Predictor (MDP) responsible for controlling SSB behavior on both the Neoverse-N2 CPUs for servers and Cortex-X3 CPUs for mobile devices. Through reverse engineering, we uncover the design of the MDP, including its state machine, organization, and replacement policy. We find that the MDP state can be persistently updated by any load, even in the absence of a preceding store. Moreover, the MDP is indexed by a part of the load's virtual address. We also observe that the MDP can be updated before the load commits, enabling it to persistently record leaked data across transient execution.

Attack Primitives. Building on these findings, we present SSBleed, the first non-speculative side-channel attack that exploits the SSB on Armv9 CPUs. SSBleed enables both cross-process and cross-privilege leakage of control-flow and data-flow information. To make the attack practical in user space, we design a primitive that probes MDP states without relying on cache side channels, thereby bypassing SSBS.

Case Studies. We evaluate the effectiveness of SSBleed through five case studies, and show that existing defenses such as SSBS are insufficient to prevent this non-speculative data leakage. First, we demonstrate two control-flow attacks on Armv9, that break secret-dependent branches in the RSA signature and RSA key generation functions in the latest version of WolfSSL [80] and MbedTLS [50], respectively. These attacks show SSBleed's practicality in real-world environments, as well as its effectiveness with a single execution. Second, we use SSBleed to monitor interrupts from other context switches with microsecond-level granularity. Finally, we use SSBleed to improve two transient attacks: SpectreRewind [23] and TikTag [39], making SpectreRewind practical in user space and significantly reducing the number of repetitions required by TikTag.

Defense. Given that SSBleed bypasses all existing defenses, we propose a software-based mitigation that can be deployed with a low performance overhead. Specifically, we implement a kernel patch that flushes the MDP state during context switches, effectively preventing cross-process and cross-privilege attacks. Evaluation on SPEC2017 shows that the defense has an average performance overhead of 0.46%.

Contribution. In summary, our contributions are as follows:

- We study the SSB on two Armv9 CPUs and discover an undocumented predictor, MDP. We reverse-engineer its state machine, organization, and replacement policy.
- We propose SSBleed, the first non-speculative side-channel attack that exploits security weaknesses in the MDP. SSBleed bypasses existing defenses and is practical when an unprivileged coarse-grained timer is used.
- We demonstrate the effectiveness of SSBleed through five case studies targeting both control-flow and data-flow leakage. To the best of our knowledge, SSBleed is the first byte-level control-flow attack on Armv9.
- We propose and evaluate a software-based mitigation that flushes MDP state during context switches.

Responsible Disclosure. We reported this vulnerability to Arm in March 2025. Arm confirmed that existing mitigations are insufficient to defend against SSBleed. At the time of publication, we observe that SSBS has been enhanced to totally disable the MDP on Neoverse-N2 CPUs, and SSBleed can be mitigated via SSBS. Moreover, CVE-2025-54764 has been assigned by the Mbed TLS security team to track the vulnerability that allows SSBleed to break the inverse modular function in the library.

II. BACKGROUND

A. Speculative Store Bypass and MDP

Speculative Store Bypass (SSB) primarily targets uncertain Read-After-Write (RAW) data dependencies [63], [85]. Specifically, when the address of a store instruction is not yet resolved, and a younger load instruction with a ready address is issued, the CPU cannot immediately determine whether the store and load are data-dependent, i.e., whether they access the same address. If so, the load must wait for the store to complete. If not, the load can be safely executed ahead of time. To improve performance in such cases, modern CPUs implement the SSB [3], [5], [34], which speculatively executes the load before the preceding store completes. Once a store address is resolved, the CPU checks for the dependence. If the load is incorrectly executed, it is squashed and re-issued.

To reduce the frequency of incorrect speculations, SSB is typically guided by a predictor known as the Memory Dependence Predictor (MDP) [19], [40], [52], [57], [69], [71]. The MDP follows general predictor design principles. It consists of several entries selected by the instruction address of the store or load. Based on the current state of the entry, the MDP predicts whether SSB should be allowed for the load, and updates its state based on the actual dependence.

B. Transient Attacks

Transient attacks primarily exploit transient execution (i.e., incorrect speculative execution) of loads to access arbitrary data from arbitrary addresses [41] or test the validity of addresses [39], [62]. Although CPUs eventually detect such incorrect speculation and discard their architectural effects (e.g., changes to registers or memory), the accessed data can still affect microarchitectural state during transient execution. These effects include both transient contention [9], [10], [23] and non-speculative persistent state updates on caches [30], [39], TLBs [62], prefetchers [31], micro-op caches [64], and the MDP of Armv9 CPUs.

Transient attacks can be classified based on their triggering mechanisms [12]. These include mispredictions on conditional branches [41], indirect branches [65], return instructions [78], decoder [79], as well as microarchitectural data sampling [11], [68], [73] and Speculative Store Bypassing (SSB) [12], [48], [59]. These attacks can breach isolation boundaries between sandboxes [37], [38], processes [41], [48] and the kernel [8], [65], [78]. To mitigate such attacks, modern CPUs introduce mechanisms to reduce or prevent incorrect speculation. These include serialization barriers and controls to disable specific types of speculation. For SSB, these controls include SSBD on x86 CPUs [3], [33] and SSBS on Arm CPUs [5].

C. Non-speculative Microarchitectural Side-channel Attacks

Non-speculative microarchitectural side-channel attacks exploit persistent state changes in microarchitectural components to leak information. For example, memory access instructions may load data into the cache. After a victim executes such an instruction, an attacker can probe cache states using methods such as Flush+Reload [49], [81], Prime+Probe [44], or Flush+Flush [29] to infer the victim's memory access patterns. Similar leakage sources include the TLB [28], [62], [72], data prefetchers [13], [14], [16], [31], branch predictors [22], [32], [82], and the MDP discovered in this work.

These non-speculative side channels can be used to encode and transmit data obtained during transient execution [41], [48], [64]. They can also leak architectural-level instruction information. For example, cache, TLB, and prefetcher side channels can enable data-flow leakage, revealing memory access patterns and compromising cryptographic keys in algorithms such as AES [29], [44], RSA [49], [81], ECDH [70], EdDSA [28], and post-quantum schemes [13].

Similarly, side channels in branch predictors [22], the decoder and micro-op caches [20], [58], and prefetchers [16] can be used for control-flow leakage, exposing which code paths are executed and revealing secret-dependent control flows or monitoring kernel and I/O events [14], [29].

III. REVERSE ENGINEERING OF MDP

In this section, we perform reverse engineering on the MDP design on Armv9 CPUs based on the SSB behavior. Our experimental platforms include Microsoft Azure D2ps v6 (2 Neoverse-N2 cores) and Google Pixel 8 Pro (1 Cortex-X3 core, 3 Cortex-A720 cores, and 4 Cortex-A520 cores). We confirm the existence of MDP on the performance cores (Section III-A) and analyze its implementation details (Section III-B to III-E).

A. Existence of MDP

SSB allows a load to execute speculatively before the preceding store computes its address, and the MDP predicts whether SSB is triggered. To verify the existence of the MDP, we design the microbenchmark shown in Fig. 1. In the `stld` function, we include a store targeting `array1[delayed_idx]` (line 4) and a load targeting `array1[0]` (line 5). Before executing `stld`, we use the `dc civac` instruction to evict `delayed_idx` from the cache, which delays the generation of the store address. To determine whether the load triggers SSB, we use the Flush+Reload cache side channel [49], [81]. Specifically, before executing `stld`, we flush all addresses in `array2` from the cache. Then, the data loaded to `temp` is used as an address to access `array2` (line 6). If `delayed_idx` equals 0 and SSB is triggered, the load accesses `old_value` before it is overwritten by `new_value`. By probing which address in `array2` is cached through cycle-level timer `PMCCNTR` in the Arm Performance Monitor Unit (PMU), we infer whether SSB is triggered.

To test the existence of MDP, we use the `experiment` function. By setting `delayed_idx` to 0 or 10, we control whether there is a data dependence between the store and load in `stld`. We denote data-dependent accesses as `SA` (Same Address) and independent accesses as `DA` (Different Address). We first train the MDP by executing a large number of `stld` functions with `old_value` equal to `new_value`. Then we update `old_value` to 240 and execute `stld` once to check for SSB. We design two experiments as follows:

- 100`SA`–k`DA`: 100 `SA` followed by k `DA` during training.
- k`SA`–100`DA`: k `SA` followed by 100 `DA` during training.

As shown in Fig. 2, we observe that on Neoverse-N2, Cortex-A715 and Cortex-X3 CPUs, the k`SA`–100`DA` always triggers SSB, while 100`SA`–k`DA` triggers SSB only when k exceeds a boundary. This indicates that these CPUs implement MDP: after training with 100 `DA`, the MDP predicts SSB; after training with 100 `SA`, the predictor learns that the load should not trigger SSB, and more than 10 `DA` are required to reverse that prediction. We do not observe the existence of MDP on efficiency cores, i.e., Cortex-A520 in Pixel 8.

```
1  uint64_t delayed_idx;  // generate store address
2  uint8_t array1[64], array2[256 * 4096], temp;
3  void stld(int new_value) {
4      array1[delayed_idx] = new_value;  // delayed store
5      temp = array1[0];  // load to be tested
6      // cache side channel to test SSB
7      temp &= array2[temp * 4096];
8  }
9  void experiment(int raw_control, int old_value) {
10     // flush addresses in array2 for cache side channel
11     flush_all(array2);
12     // SA: 0, DA: 10, Trigger: 0
13     delayed_idx = raw_control;
14     // Train: 0, Trigger: 240
15     array1[0] = old_value;
16     // delay the address of the store by cache flushing
17     flush(&delayed_idx);
18     stld(0);
19 }
```

Fig. 1. C implementation of our microbenchmark to reverse-engineer the MDP. Data dependence in `stld` (lines 4-5) is controlled by `raw_control`. We use `flush` to delay the store, and use the cache side channel to test whether `old_value` is loaded before the updated of `new_value`.

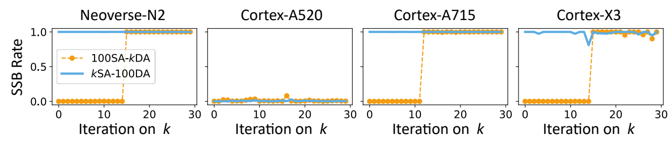


Fig. 2. Speculative store bypass rate after executing 100 `SA` and k `DA` (orange lines) or k `SA` and 100 `DA` (blue lines). The different SSB rates under two experiments and a dynamic change in 100`SA`−k`DA` indicate the MDP is used.

B. State Machine of MDP

After validating the existence of SSB on Armv9 CPUs, we further investigate the MDP implementation details on the Neoverse-N2 and Cortex-X3 CPUs. As shown in Fig. 2, after a number of `SA` are executed to train the MDP, at least 15 `DA` are required to activate SSB on both CPUs, suggesting that the MDP uses a counter to hold the state, where `SA` and `DA` update the counter differently. To further investigate the state machine, we use the setup in Fig. 1 and vary the `SA` and `DA` sequences used to train the MDP. Similarly, during training, we keep `old_value` as 0. After training, we set `old_value` to 1 and run `experiment` to observe whether the MDP predicts SSB or load blocking (BLK).

We design 8 experiments shown in Table I, where Exp. 1 and 2 show that a single `SA` is sufficient to switch the MDP state from BLK to SSB. Exp. 3 and 4 indicate that at least 14 `DA` are required to revert the state to BLK. Besides, Exp. 5 and 6 show that a maximum of 15 `DA` is sufficient to transition the state, regardless of the number of `SA`. From these results, we infer that the MDP uses a 4-bit saturating counter. When the counter is cleared, SSB is predicted. Otherwise, BLK is predicted. Executing an `SA` while the counter is cleared sets it to 14, and subsequent `SA` increment it to saturation (i.e., 15). Each `DA` decreases the counter by 1 until it reaches 0.

To further investigate the effect of `SA` when the counter is not cleared, we conduct Exp. 7 and 8. We initialize the counter to 4 using 1 `SA` and 10 `DA`, then run an additional `SA` and measure how many `DA` are needed to revert the state

TABLE I
EXPERIMENTS ON MDP STATE MACHINE

Exp.	Training	Pred.	Exp.	Training	Pred.
1	15 DA	SSB	2	15 DA, 1 SA	BLK
3	1 SA, 13 DA	BLK	4	1 SA, 14 DA	SSB
5	10 SA, 14 DA	BLK	6	10 SA, 15 DA	SSB
7	1 SA, 10 DA, 1 SA, 4 DA	BLK	8	1 SA, 10 DA, 1 SA, 5 DA	SSB

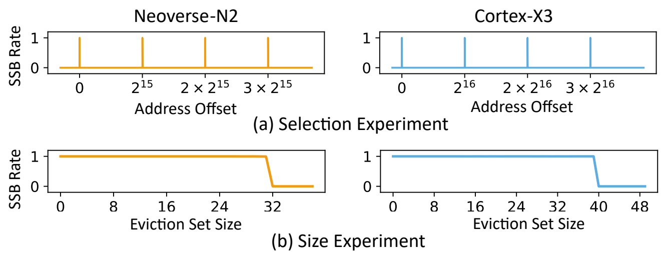


Fig. 3. Experiments on MDP selection function (a) and prediction table size (b) on Neoverse-N2 and Cortex-X3 CPUs.

to SSB. The results show that 5 DA are required instead of 4, indicating that running an SA at counter value 4 increases it to 5. Therefore, we infer that SA decrements the counter by 1 when the counter is not saturated or cleared.

Based on these findings, we construct the state machine shown in Fig. 6. To validate the accuracy of the inferred state machine, we exhaustively test all 2^{30} training sequences with all permutations of 30 SA and DA. Using the state machine to predict the outcome of each, and validating with the code shown in Fig. 1, we achieve a 100% match, indicating that our reverse engineering is accurate with high confidence.

C. Organization of MDP

In this section, we analyze the organization of the MDP. We first investigate how different store-load pairs share MDP state by extending code in Fig. 2 by extracting the machine code of the stld function, creating two versions named stld_1 and stld_2. We fix the address of stld_1 and use just-in-time code generation to dynamically place stld_2 across a large address range. For each location of stld_2, we train the MDP using stld_2 to alternate between SSB and BLK states, and observe the effects on stld_1. If both training outcomes are reflected in stld_1, we infer that the two functions share the MDP counter in the same entry. We then record the relative address offset between stld_1 and stld_2 where such sharing occurs.

As shown in Fig. 3, on Neoverse-N2, the training results are shared if and only if the instruction virtual addresses (i.e., IPs) of the stld differ by a multiple of 2^{15} bytes. We further insert varying numbers of nop instructions between the store and load in both functions and find that sharing still occurs as long as the load IPs have the same lower 15 bits. We conclude that, on Neoverse-N2, the MDP entries are selected by the lowest 15 bits of the load IP. Similarly, on Cortex-X3, we find that the lowest 16 bits are used for the entry selection.

Next, we investigate the number of entries in the MDP. Based on the selection function, we construct eviction sets containing k distinct stld functions, each with a unique load IP in the relevant low-bit range (i.e., 15 bits for Neoverse-N2 and 16 bits for Cortex-X3), ensuring no entry sharing with stld_1. By default, the MDP predicts SSB, and thus we first train the MDP state at stld_1 to BLK, then use the eviction set with a size k to train the related entries to BLK, and finally run stld_1 again to observe if its prediction has reverted to SSB. If so, the entry used by stld_1 is evicted.

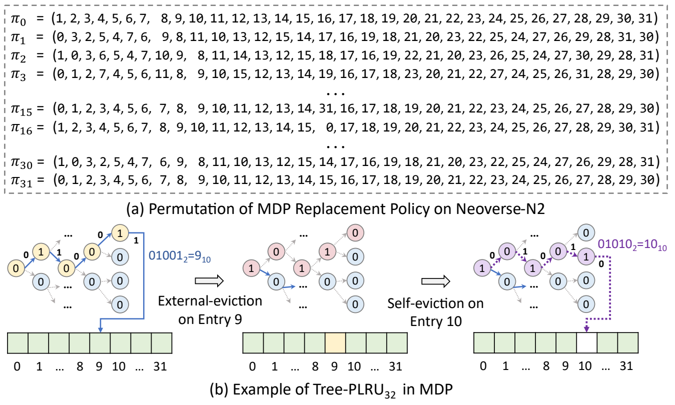
As shown in Fig. 3, we find that, on Neoverse-N2, the prediction at stld_1 is overwritten when $k = 32$, indicating that the MDP contains 32 entries. On Cortex-X3, the threshold is 40 entries. To further validate associativity, we generate 2^{20} random eviction sets of size 32 on Neoverse-N2, and always observe the eviction. Therefore, the MDP in Neoverse-N2 is fully-associative, using the lowest 15 bits of the load IP as the tag for matching, without using any bits for indexing. We observe the same conclusion on Cortex-X3, except the tag consists of the lowest 16 bits of the load IP.

D. Replacement Policy of MDP

In the organization experiment, we also observe that if each entry in the eviction set is trained into the SSB state using the sequence 1 SA 15 DA, we can no longer observe eviction. This leads us to conclude that the MDP employs two types of eviction mechanisms. The first is external-eviction triggered when the entry table is full, and the second is self-eviction triggered when the internal counter of an entry reaches 0. In this section, we focus on analyzing the replacement policy used during these two eviction mechanisms.

We construct a set of stld functions where the load in each function selects a unique MDP entry. These functions are labeled as $f_0, f_1, ..., f_{k-1}$, where $k \geq 2s$, and s is the total number of MDP entries. For each function f, we perform 10 SA to ensure its corresponding entry is fully trained and inserted into the MDP. After filling the MDP with the execution order f_{s-1} to f_0, we introduce operations O that may affect replacement order, including O_i^1: updating the counter value of entry i without triggering self-eviction, and O_i^2: actively removing entry i via self-eviction. We then execute entries f_s to f_{2s-1} to trigger external-evictions. Finally, we probe from f_0 to f_{s-1} to determine which entries have been evicted. This allows us to reconstruct the replacement order of the original entries.

Following previous research [1], [72], we represent the replacement ordering as a permutation $\Pi_s = (\pi_{-1}, \pi_0, \pi_1, ..., \pi_{s-1})$, where each π_i is a permutation of the remaining entries after applying O_i^1 or O_i^2, and a higher index indicates a higher eviction priority. Notably, π_{-1} denotes the default replacement order without any operations applied. On both Neoverse-N2 and Cortex-X3, we observe that $\pi_{-1} = (0, 1, 2, ..., s-1)$, where each $i \in \pi_{-1}$ denotes the entry corresponding to f_i, and the values in the right have a higher eviction priority. This result indicates FIFO (i.e., First-In, First-Out) policy, where newer entries


$$\pi_0 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)$$
$$\pi_1 = (0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14, 17, 16, 19, 18, 21, 20, 23, 22, 25, 24, 27, 26, 29, 28, 31, 30)$$
$$\pi_2 = (1, 0, 3, 6, 5, 4, 7, 10, 9, 8, 11, 14, 13, 12, 15, 18, 17, 16, 19, 22, 21, 20, 23, 26, 25, 24, 27, 30, 29, 28, 31)$$
$$\pi_3 = (0, 1, 2, 7, 4, 5, 6, 11, 8, 9, 10, 15, 12, 13, 14, 19, 16, 17, 18, 23, 20, 21, 22, 27, 24, 25, 26, 31, 28, 29, 30)$$
$$\dots$$
$$\pi_{15} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 31, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30)$$
$$\pi_{16} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)$$
$$\dots$$
$$\pi_{30} = (1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14, 17, 16, 19, 18, 21, 20, 23, 22, 25, 24, 27, 26, 29, 28, 31)$$
$$\pi_{31} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30)$$

(a) Permutation of MDP Replacement Policy on Neoverse-N2

(b) Example of Tree-PLRU$_{32}$ in MDP

Fig. 4. The Tree-PLRU$_{32}$ replacement policy in the Neoverse-N2 MDP. (a) shows the effective permutation caused by replacement due to self-eviction. (b) shows the replacement process when the 9th entry is external-evicted due to a full table, and the 10th entry is self-evicted due to counter clear.

have lower eviction priority. Furthermore, for operation O^1 and its permutation Π_s^1, we observe $\pi_i = \pi_{-1}$ for all $\pi_i \in \Pi_s^1$. Therefore, counter value updates alone do not alter replacement priority, confirming that FIFO is used as long as no self-eviction occurs.

For operation O^2 and its permutation Π_s^2, we observe $\pi_i = \pi_{-1} - \{i\}$ on Cortex-X3, which means removing entry i leaves the ordering of remaining entries intact. Hence, Cortex-X3 strictly follows FIFO, regardless of self-eviction. On Neoverse-N2, however, Π_s^2 undergoes a non-trivial shuffle, as illustrated in Fig. 4(a). This behavior resembles a Tree-PLRU replacement policy [1], [72]. Given the MDP on Neoverse-N2 contains 32 entries, we hypothesize it employs a 5-level binary Tree-PLRU structure with 31 internal nodes. We construct the theoretical Tree-PLRU$_{32}$ permutation Π_{32}^{tp32} and confirm that for every $\pi_i^2 \in \Pi_s^2$ and corresponding $\pi_i^{tp32} \in \Pi_{32}^{tp32}$, we have $\pi_i^2 = \pi_i^{tp32} - \{i\}$. This confirms our hypothesis.

In specific, the Tree-PLRU$_{32}$ consists of 31 bits representing internal nodes of a binary tree. As shown in Fig. 4(b), on external-eviction, the MDP traverses the tree from the root. Each bit determines the direction (i.e., 0 = left, 1 = right). After 5 nodes are accessed, one entry is selected for replacement. The bits along this traversal path are then flipped to update the replacement priority. This replacement scheme approximates FIFO behavior when no self-eviction occurs. However, when entry i is self-evicted, the path from the root to entry i is extracted, and the 5 corresponding internal bits are set to the bitwise complement of the binary representation of i. This modifies the tree state deterministically and causes a specific permutation effect, as observed in Fig. 4(a). Finally, accessing an entry and updating its counter value without an self-eviction has no effects on the bits in the binary tree.

E. Update Condition of MDP

In this section, we study the conditions under which the MDP counter is updated. In previous sections, we demonstrate that non-speculative store-load pairs within a single process can trigger MDP updates. To further investigate other conditions, we design the experiments listed in Table II.

TABLE II
EXPERIMENTS ON MDP UPDATE CONDITION

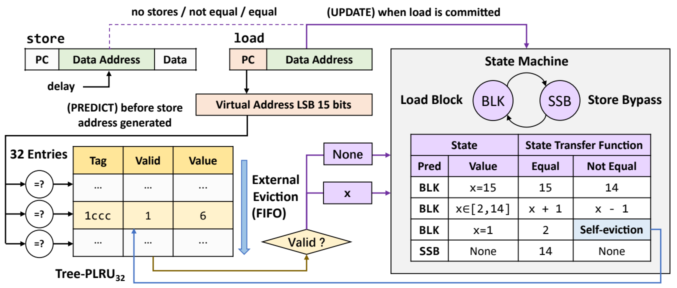
Exp.	Training		Trigger and Test		
	Process	Inst.	Process	Inst.	Pred.
1	Process-1	10 `SA`	Process-1	14 `L`	BLK
2	Process-1	10 `SA`	Process-1	15 `L`	SSB
3	Process-1	10 `SA`	Process-2	14 `DA`	BLK
4	Process-1	10 `SA`	Process-2	15 `L`	SSB

```
1  uint64_t delayed_bound = 1;   // branch condition
2  uint8_t temp, array1[2] = {16, 0}, array2[20];
3  void branch(int x) {
4      // delayed branch to trigger transient execution
5      if (x < delayed_bound) {
6          // normal: temp = 0, transient: temp = 1
7          temp = array1[x];
8          // delayed store, address from array1[x]
9          array2[temp] = 0;
10         // load with address &array2[0]
11         temp &= array2[0];
12     }
13 }
```

Fig. 5. C implementation of our experiment to test the MDP update by transient store-load pairs.

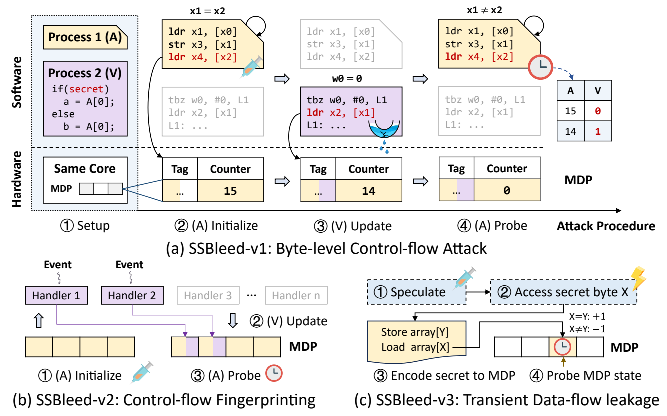
In Exp. 1 and 2, we first train the MDP to the state BLK with 10 `SA`, then execute another load without any preceding store (denoted as `L`) in the same process. We ensure that this load shares the same MDP tag as the one in `stld`. We find that after more than 14 `L`, the MDP state changes to SSB. This shows that even without a preceding store, a load can decrement the MDP counter if it is greater than zero. Next, in Exp. 3 and 4, we place the training and triggering code in different processes, while keeping the MDP tags of loads the same. We execute process 1 to train the MDP and process 2 to trigger prediction. Results show that as long as the tags match, different processes can share MDP state, i.e., the counter value.

The above experiments focus on the effect of non-speculative loads with or without preceding delayed stores. Finally, we test whether a load executed transiently can cause persistent MDP updates. We use the branch-based setup shown in Fig. 5 to trigger transient execution. Specifically, we set `x` to 0 and repeatedly execute the `branch` function to train the conditional branch (line 4), then set `x` to 1 to trigger misprediction, and evict `delayed_bound` from the cache to delay branch resolution. During the transient execution caused by the mispredicted branch, the store and load (lines 6 and 7) access the same address, and the store address is delayed by another load (line 5). If the load updates the MDP, the MDP will insert a new entry. We then use the `stld` function with the same load tag to probe the MDP after transient execution. Results show that a transient load can update the MDP, indicating that the MDP is updated before the load is committed.



State		State Transfer Function	
Pred	Value	Equal	Not Equal
BLK	x=15	15	14
BLK	x∈[2,14]	x + 1	x - 1
BLK	x=1	2	Self-eviction
SSB	None	14	None

Fig. 6. Overview of the reverse-engineered MDP on Neoverse-N2 CPU. The MDP on the Cortex-X3 CPU shares a similar design but has different parameters in tag, table size and replacement policy.



```
if(secret)
  a = A[0];
else
  b = A[0];
```

```
ldr x1, [x0]
str x3, [x1]
ldr x4, [x2]
```

```
tbz w0, #0, L1
ldr x2, [x1]
L1: ...
```

Fig. 7. Three variants of SSBleed that exploit MDP to implement cross-process control-flow attack (a), kernel control-flow fingerprinting (b), and transmit data-flow information during transient execution (c).

F. Summary of Reverse-engineering of MDP

In this section, we discover that Armv9 implements SSB using an undocumented MDP, and present the first detailed reverse engineering of the state machine, organization, replacement policy, and update conditions. Fig. 6 systematically illustrates our findings using the Neoverse-N2 MDP as an example. The MDP has a prediction table containing 32 entries. Each entry includes a tag field, a valid bit, and a value field. The tag field stores the lowest 15 bits of the load's IP, allowing the MDP to make predictions for different loads. The value field is a 4-bit counter. The counter gives the prediction according to a state machine, and is updated based on the data dependence between the load and preceding stores.

In particular, when the counter reaches 0, self-eviction is triggered. In addition, if the table is full and a new entry is inserted, external eviction occurs. Neoverse-N2 adopts the Tree-PLRU$_{32}$ replacement policy. Under external eviction, it behaves like FIFO, while self-eviction alters the replacement priority of entries. Cortex-X3 has a similar MDP design, but it has 40 entries and only uses FIFO as the replacement policy.

IV. SSBleed: Overview and Primitives

The reverse engineering results in Section III show that the MDP is shared between different security domains. In this section, we show the feasibility of MDP side-channel attacks using unpriveleged coarse-grained timers.

A. Overview of SSBleed

Section III-E shows that the MDP is shared across processes and privileges on the same CPU. Moreover, a single load is sufficient to update the MDP, and loads executed transiently can also persistently update it. Therefore, the MDP can be used to construct various control-flow and data-flow attacks. In this work, we implement three side-channel attacks, named SSBleed, as shown in Fig. 7.

SSBleed-v1 uses the MDP to perform cross-process, byte-level control-flow attacks. The attack target is a secret-dependent branch in the victim code, which includes a load (①) in the branch path. The attacker first executes a store-load pair with the same MDP tag as the target load, training the counter to 15 (②). If the victim then executes the target load, the counter of the corresponding MDP entry is decremented by 1 (③). Finally, the attacker probes the MDP state in its own address space to determine whether the target load has been executed, thereby inferring the branch direction (④).

SSBleed-v2 uses the MDP to monitor kernel behavior from user space. Specifically, the attacker identifies loads executed in different interrupt handlers and selects a set of loads with distinct MDP tags. These tags are inserted into MDP in advance (①). Then, if an external interrupt or syscall triggers a context switch (②), the attacker monitors changes in MDP counter values or the eviction of MDP table entries to determine which interrupts or syscalls occur during a time window (③).

SSBleed-v3 uses the MDP to transmit secret data obtained via transient execution or to verify address validity. For example, the attacker first trains a conditional branch to induce misprediction and trigger transient execution (①). During the transient execution, a secret byte is accessed (②). The attacker encodes this byte into the load address (i.e., the array index) of a store-load pair and adjusts the store address to see whether a new MDP entry is updated due to address matching (③). If so, the leaked byte value equals the index of the array (④).

B. Threat Model

We follow the standard threat model for single-core and user-space attackers [16], [20], [31], [48], [66]. We assume a user-level attacker who shares the same core with the victim. The attacker can execute unprivileged code. On Arm CPUs, the attacker cannot access the privileged timer `PMCCNTR` for cycle-level timing and is restricted to the user-accessible timer `CNTVCT_EL0`. On Microsoft Azure instances, this timer operates at a default frequency of 1 GHz, with a precision of 1 ns. On the Pixel 8 Pro, however, the default frequency is 24.5 MHz, with a precision of about 40 ns.

For SSBleed-v1, we assume the attacker knows a secret-dependent branch in the victim's code through binary analysis and has identified at least one load on the branch path. We assume the attacker is capable of preempting the victim to achieve the inter-process interleaving shown in Fig. 7(a), which has been demonstrated as practical in prior work [66], [92]. For simplification, we follow the experimental setup in [16] where the victim voluntarily yields control for each execution round. For SSBleed-v2, we assume the attacker can pin their process across multiple cores. For SSBleed-v3, we assume the attacker can inject code into the victim's address space (e.g., via JIT or shared libraries), which is a common assumption in transient attacks [37]–[39]. Finally, we assume the victim system is deployed with existing mitigations enabled, including address space layout randomization (ASLR) and SSBS, and that other system settings remain at default.

C. Prime and Probe Primitives

During reverse engineering, we detect whether a load triggers SSB by observing cache states. However, as a defense against transient attacks such as Spectre-v4 [12], Armv9 CPUs implement the SSBS to block the data forwarding by a load performing SSB, and thus eliminate the cache state changes caused by the load [5]. Therefore, we require a method to probe the MDP state even when SSBS is enabled.

Inspired by prior work [48], [59], we design the probing primitive using `mul` instructions to induce port contention, as shown in Fig. 8,. This not only delays the address generation of the store, but also creates a timing side channel independent of the cache to observe the MDP state. To validate the feasibility of this primitive, we measure the timing under both MDP states with SSBS enabled. We use a sufficient number of `mul`s to ensure that the timing difference is observable with coarse-grained timers. As shown in Fig. 9, both CPUs show measurable timing differences between the BLK and SSB states. Under `CNTVCT_EL0`, Neoverse-N2 shows a difference of about 10 ticks, and Cortex-X3 shows a difference of 2 ticks.

Further, this primitive can be used to both prime and probe the MDP. Similar to Fig. 1, we mark $\texttt{x0} = \texttt{x1}$ as `SA` and $\texttt{x0} \neq \texttt{x1}$ as `DA`. We fill an MDP entry with 1 `SA` and initialize the counter to 14. During probing, we execute multiple `DA` in sequence and measure their execution times. According to the state machine, if the counter value is k, then the first k `DA` will be in BLK state, and the rest in SSB. We observe noticeable slowdowns for the first k `DA`, allowing us to infer k.

To further confirm that the timing differences are caused by the MDP, we conduct the following experiments. First, to exclude the effects of prefetchers [7], we execute additional loads before probing to ensure that both the store and load in the `mdp_probing` hit the cache. Second, to exclude the effects of other predictors, such as value predictors [37] and address predictors [38], we randomly generate different values for `x0`, `x1` and `[x1]`, while keeping $\texttt{x0} \neq \texttt{x1}$ for each probing round. Third, to exclude the effects of speculation in the arithmetic unit [54], we replace the `mul` instructions with other arithmetic operations. Finally, we insert barrier instructions (i.e., `dsb`) between the store and load. The timing difference persists across the first three experiments, but does not exist for the last experiment, demonstrating that the MDP is the root cause of the observed timing variation.

```
mdp_probing:
    // avoid semantic error of multiplication
    mov x2, #0x1
    // contention 1: delay store address
    .rep 50
    mul x0, x0, x2
    .endr
    str x2, [x0]     // store
    ldr x3, [x1]     // load
    // contention 2: delay memory dependency resolution
    .rep 50
    mul x3, x3, x2
    .endr
    ret
```

Fig. 8. A64 assembly implementation of the probe primitive to probe the MDP current state.

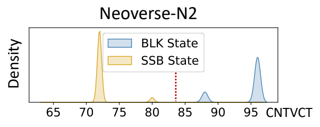


Fig. 9. Timing difference between two MDP states created via port contention.

V. SSBleed: Case Studies

In this section, we demonstrate the effectiveness and practicality of SSBleed through five real-world case studies shown in Table III. We first show how SSBleed breaks secret-dependent branches to leak RSA keys, proving its efficiency in both repeatable unbalanced branch scenarios (Section V-A) and single-trace balanced scenarios (Section V-B). We then show that SSBleed can leak branchless control flows (Section V-C), enabling real-time interrupt monitoring. Finally, we demonstrate SSBleed's capability to leak data flows (Section V-D to V-E) by improving transient attacks. Notably, all of the experiments are conducted with SSBS enabled.

A. Attack on RSA Signature Implementation of WolfSSL

Attack Target. WolfSSL accelerates RSA signing using the Chinese Remainder Theorem (CRT). Given a private key D, it precomputes $dP = D \bmod (P-1)$ and $dQ = D \bmod (Q-1)$, where P and Q are two large primes. It then computes the modular exponentiations $h^{dP} \bmod P$ and $h^{dQ} \bmod Q$ for a message hash h, and combines the results to produce the ciphertext. When hardware acceleration is disabled, the function shown in Fig. 10 is called to perform the modular exponentiations, which involves multiple iterations.

In each round, the variable `y` scans a bit of the secret keys, and selects an operation mode based on the bit: skipping leading zeros (mode 0, line 13), performing squaring (mode 1, lines 14 to 16), or performing both squaring and multiplication (mode 2, lines 17 to 24). We observe that accessing variables or calling functions causes execution of loads to access the data from the stack, and identify one load in mode 0, one in mode 1, and four in mode 2. These loads leak the branch targets, directly leaking dP and dQ from `y`.

Attack Setup. We use the default gcc compilation options and a 4096-bit key. Similar to prior state-of-the-art works [16], [24], [25], we fix the window size (i.e., `winsize` in Fig. 10)

TABLE III
OVERVIEW OF 5 SSBLEED CASE STUDIES ON CONTROL FLOW AND DATA FLOW SIDE-CHANNEL ATTACKS

Attack Information		Attack Target				Attack Feature*					
Section	**Case Study**	**Variant**	**Victim**	**Function**	**Loads**	**Leakage**	**BL**	**SP**	**ST**	**SdB**	**BB**
V-A	RSA Signature	v1	WolfSSL v5.7.6	Exponent Modular	6	Control Flow	✔	✗	✗	✔	✗
V-B	RSA Key Generation	v1	MbedTLS v3.6.2	Inverse Modular	9	Control Flow	✔	✗	✔	✔	✔
V-C	Interrupt Detection	v2	Linux 6.8.0-1027-azure	Interrupt Handler	1	Control Flow	✔	✗	✔	—	—
V-D	SpectreRewind-MDP	v3	5.15.137-android14-11	Attacker Injected	1	Data Flow	✔	✔	—	—	—
V-E	Tiktag-MDP	v3	5.15.137-android14-11	Attacker Injected	1	Data Flow	✔	✔	—	—	—

* Column 6 (**Loads**) shows the number of vulnerable loads used in the attack. Column 7 (**Leakage**) indicates the leakage type. Column 8 (**BL**) denotes byte-level granularity. Column 9 (**SP**) indicates speculative execution. Column 10 (**ST**) indicates a single-trace control flow side-channel attack. Column 11 (**SdB**) indicates the attack target is the secret-dependent branches. Column 12 (**BB**) indicates the attacked branch is balanced at the source code level.

```
 1 // X is one of the RSA keys (dP, dQ)
 2 digidx = X->used - 1; // get the bit number of X->dp
 3 // 0: skip leading 0s; 1: handle bit 0; 2: handle bit 1
 4 mode = 0;
 5 bitcnt = 1;  // to iterate within each bit block
 6 for (;;) {   // access one bit (from MSB) in each round
 7     if (--bitcnt == 0) {
 8         if (digidx == -1) break;
 9         buf = X->dp[digidx--];
10         bitcnt = 60;
11     }
12     y = (int)(buf >> 59) & 1;
13     buf <<= 1;
14     // secret-dependent branch 1
15     if (mode == 0 && y == 0) continue;
16     // secret-dependent branch 2
17     if (mode == 1 && y == 0)
18        ... // square function
19     mode = 2;
20     // the branch below is always taken when winside=1
21     if (1 == winsize) {
22         for (x = 0; x < winsize; x++)
23             ... // square function
24         ... // multiply function
25         mode = 1;
26     }
27 }
```

Fig. 10. Part of the `mp_exptmod_fast` function in WolfSSL v5.7.6. During RSA signing, dP and dQ are used as exponents. The variable `y` scans each bit from the most significant bit. Load instructions (lines 15, 20, 22) leak `y` via conditional branches (lines 13, 14), which further leaks dP and dQ.

to 1, ensuring that every bit affects the control flow. When `winsize` is greater than 1, cache side channels are required to leak bits within each window, which has been shown to be feasible [49]. Using SSBleed, we track updates to the MDP entries in each iteration to monitor the execution of 6 loads.

Experimental Results. We randomly generate 1000 RSA keys of 4096 bits and use SSBleed to leak dP and dQ. We evaluate the leakage accuracy and the number of incorrectly recovered bits, as shown in Fig. 11. Without interference from other user processes, each attack takes approximately 9.29 seconds. A single run achieves 94% accuracy, and averaging over 5 repetitions achieves 100% accuracy. We also evaluate the impact of compute-intensive, memory-intensive, and I/O-intensive workloads on attack accuracy. Compute and memory-intensive scenarios generated using `openSSL` and `stress-ng` have minimal effect, requiring only 10 repetitions to reach 100% accuracy. In contrast, I/O-intensive scenarios generated by `iperf3` significantly lower

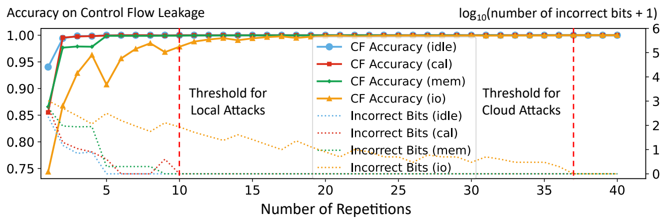


Fig. 11. Evaluation of attacks on the RSA signature implementation in WolfSSL. We test 1000 RSA keys across 4 environments, measuring control flow recovery accuracy (solid line) and incorrect bits (dotted line) under varying repetitions. In non-I/O-intensive settings, 10 repetitions suffice for 100% accuracy; otherwise, about 37 repetitions are needed to suppress noise.

the accuracy, as frequent context switches may exhaust and evict MDP entries. Under this condition, a single trace may contain around 1000 bit errors, and more than 37 repetitions are needed to fully recover the key.

Discussion. Instruction-level control-flow attacks on RSA modular exponentiation typically rely on specific microarchitectural features, such as Intel's branch predictors [82], [86] and prefetchers [16], and AMD's scheduler [24]. Other attacks assume privileged attackers with access to page tables [51], interrupts [58], or performance counters [24]. On Armv9 CPUs, the microarchitectural details mentioned above remain largely unexplored, and we assume a non-privileged threat model. To the best of our knowledge, this is the first byte-level control-flow attack demonstrated on Armv9.

B. Attack on RSA Key Generation of MbedTLS

Attack Target. MbedTLS uses the modular inverse algorithm for RSA key generation. After obtaining the large primes P and Q, MbedTLS computes the private key D by calling a modular inverse function, where $D = E^{-1} \bmod L$, E is the public exponent, and $L = \frac{(P-1)(Q-1)}{\gcd(P-1,Q-1)}$.

The latest version of MbedTLS implements modular inversion using the BEEA algorithm, as shown in Fig. 12. It performs multiple iterations on the initial inputs E and L, stored in variables u and v. Each iteration consists of a u-loop (lines 2–5), a v-loop (lines 6–9), and a sub-step (lines 10–16). The u-loop and v-loop (collectively referred to as x-loop) repeatedly right-shift u and v until their least significant bit is set to 1. The sub-step is a balanced branch (i.e., both

```
1  do {  // input: u = E, and v = (P-1)(Q-1)/gcd(P-1,Q-1)
2      while((TU.p[0] & 1) == 0) // u-loop: if u % 2 == 0
3          mbedtls_mpi_shift_r(&TU, 1); // u = u >> 1
4      while((TV.p[0] & 1) == 0) // v-loop: if v % 2 == 0
5          mbedtls_mpi_shift_r(&TV, 1); // v = v >> 1
6      // sub-step: if u >= v
7      if (mbedtls_mpi_cmp_mpi(&TU, &TV) >= 0)
8          mbedtls_mpi_sub_mpi(&TU, &TU, &TV); // u -= v
9      else  // u < v
10         mbedtls_mpi_sub_mpi(&TV, &TV, &TU); // v -= u
11 } while (mbedtls_mpi_cmp_int(&TU, 0) != 0);
```

Fig. 12. Part of them `mbedtls_mpi_inv_mod` function in MbedTLS v3.6.2. During RSA key generation, the private key D is computed as $E^{-1} \bmod L$. `ldr` instructions (lines 3, 7, 11, 14) leak the number of iterations of the u-loop (line 2) and v-loop (line 6), and the execution path of the sub-step (line 10), which further leaks P and Q.

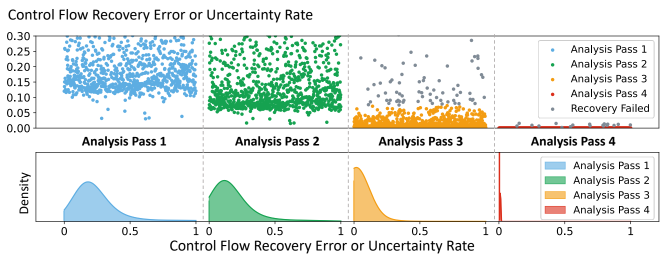


Fig. 13. Evaluation of attacks on RSA key generation in MbedTLS. Control flow recovery improves with each analysis pass. Out of 1000 RSA keys, 615 are successfully recovered. The remaining 385 fail due to early termination in pass 3 or persistent noise that cannot be reduced in pass 4.

branches execute the same code) that subtracts the smaller of u or v from the larger one and updates the corresponding variable. Each direction of the u-loop, v-loop, and sub-step contains three loads that access the return values and invoke `MBEDTLS_MPI_CHK` to verify the results. These loads leak the direction of three secret-dependent branches (lines 2, 6, and 10), which can be used to reconstruct L and ultimately recover the primes P and Q.

Attack Setup. Aligned with prior state-of-the-art works [46], [58], we use the latest version of MbedTLS to generate 2048-bit RSA keys. After leaking the control flow, we derive P and Q from L based on methods established in previous research [46]. Compared to modular exponentiation in Section V-A, RSA key generation invokes the inverse modular function only once to compute the private key, meaning that only a single trace is available for recovery. This significantly increases the attack difficulty. However, our results show that by using SSBleed and exploiting mathematical properties to reduce measurement noise, an effective userspace attack is still achievable.

Noise Reduction. For attacks on the modular inversion algorithm, 100% accuracy from a single trace is required. Otherwise, the correct L cannot be recovered. To achieve this, we perform four analysis passes, exploiting control-flow correlations to reduce noise and obtain the final result.

In the first pass, we identify the control flow in each iteration by analyzing which of the 12 loads are executed. In the second pass, we exploit a property: between any two adjacent sub-step operations, there must exist an x-loop, and only one of u-loop or v-loop is executed. Based on this property, we segment control flow using sub-step as boundaries and correct the control flow between adjacent sub-steps to either u-loop or v-loop. In the third pass, we exploit another key property: if the sub-step branch is taken, the next iteration must execute a u-loop. Otherwise, it must execute a v-loop. This correlation enables us to refine the classification of the previous sub-step based on the following x-loop, and vice versa.

In the final pass, we enumerate the remaining uncertain control-flow cases, i.e., those in which the observed loads does not match any known pattern. If the number of possibilities is fewer than 64, we exhaustively enumerate all options and retain only those leading to valid prime values for P and Q. If the candidates exceeds this threshold, we terminate early, as the noise cannot be effectively eliminated. In practice, the attacker can relax this constraint and increase enumeration time to improve the likelihood of successful key recovery.

Experimental Results. We randomly generate 1000 RSA keys of 2048 bits and use SSBleed to leak the control flow of the modular inversion. After four passes of analysis, we successfully recover 615 keys, i.e., we achieve 100% single-trace control flow recovery accuracy for these keys. On average, 10 control-flow candidates are evaluated per key. Each candidate takes approximately 9.19 seconds to verify, resulting in an average recovery time of around 90 seconds per key.

We further evaluate the control-flow error rate across all 1000 traces after each analysis pass, as shown in Fig. 13. Without the noise reduction process, the initial trace error rate ranges from 15% to 20%. The error rate decreases after each pass of analysis, with the third pass contributing the most significant improvement. Among the unrecovered keys, 369 are terminated early during the third pass due to an excessive number of candidates. The remaining 16 keys are not leaked successfully due to unrecoverable noise. In practice, attackers can increase the enumeration threshold to improve key recovery success at the cost of longer runtime.

Discussion. Attacks on RSA modular inversion require high noise resilience, as only a single trace is available. Existing attacks [46], [51], [58] often rely on privileged access to page tables to infer the number of x-loops, since x-loops are unbalanced branches that trigger additional page accesses when taken. Compared to these attacks, SSBleed observes load executions to leak x-loop counts. For sub-step identification, prior works typically require specific compiler options to emit store [58] or store-load pairs [46], enabling microarchitectural side channels to track memory accesses. SSBleed only relies on the presence of a single load along the branch path, demonstrating strong noise resistance and significantly expanding the attack surface of real-world code.

C. Interrupt Detection

Interrupt Side Channels. Interrupts are a well-established shared resource for side channel attacks [14], [60], [61], [67], [76], [88], [90]. They can be detected by an unprivileged

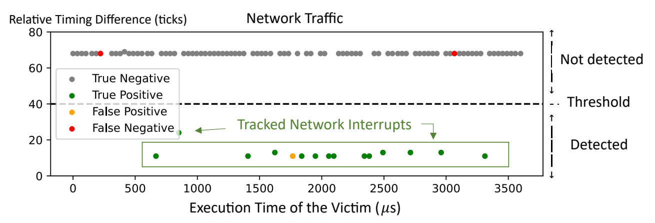


Fig. 14. Evaluation of interrupt side channels. This side channel successfully distinguishes network interrupts from other events, including memory allocation, file operations, and idle states.

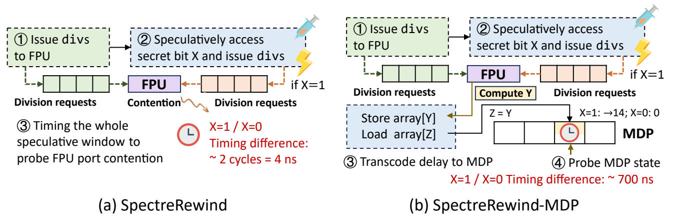


Fig. 15. Basic principle of SpectreRewind (a) and our improvement (b). Using MDP, we transcode transient contention on the division unit into persistent MDP state changes, amplifying the timing difference from 4 ns to 700 ns, making it distinguishable in user space.

attacker via architectural or microarchitectural changes. Such attacks can leak user activities such as keystrokes [60], [61], [67], [76] and network traffic [14], [88], [90]. Among existing interrupt detection techniques, only LeakIDT [76] can spy on specific types of interrupts, and the rest merely detect the presence of an interrupt. However, LeakIDT [76] depends on x86-specific features (e.g., interrupt descriptor table), rendering it inapplicable to ARM architectures.

Attack Setup. We demonstrate that SSBleed goes beyond basic interrupt detection by accurately identifying specific types of interrupts among a wide range of context-switching events. Specifically, on Linux kernel versions listed in Table III, we trigger different types of interrupts and syscalls, and profile all MDP tags to identify load that are executed only in specific interrupt handlers. As a case study, we focus on the network interrupt handler, identifying a set of unique load tags used exclusively by that handler. Using the process in Fig. 7(b), we initialize and probe the corresponding MDP entries from a user process. If we observe changes in any of the selected tags, we infer that the network interrupt occurs and that the handler is executed.

Experimental Results. Over a time window of 3600 μs, we randomly trigger 16 interrupts and 25 noisy context switches while using SSBleed to continuously monitor the execution of the network interrupt handler. The experiment consists of 100 probes, corresponding to a detection resolution of 27.78 kHz. As shown in Fig. 14, SSBleed accurately detects network interrupts with μs-level granularity while suppressing noise from other context switches, which is fast enough to track network packets in real world applications [14]. The results show an accuracy of 97%.

D. Improvement of SpectreRewind

SpectreRewind. SpectreRewind [23] is the first transient attack on Arm CPUs that does not rely on caches. As shown in Fig. 15(a), the attacker first trains the branch predictor to trigger a misprediction, and then executes a sequence of `fdiv` instructions to delay branch resolution. During the transient execution, the attacker accesses a secret bit and, depending on whether the bit is 1, issues another sequence of `fdivs` to introduce contention. Finally, the total execution time of the transient execution is measured: longer latency indicates that the secret bit is 1. We reproduce SpectreRewind on Cortex-X3 and observe that the timing difference caused by contention is below the resolution of `CNTVCT_EL0`, making the attack infeasible in user space.

Attack Improvement. In this work, we improve SpectreRewind using the MDP. As shown in Fig. 15(b), within the transient window, we insert a store-load pair with the same address. We use `fdivs` to delay the store's address generation. If the secret bit is 1, contention on the division unit delays the store address, activating the MDP and inserting a new entry for the load. If the bit is 0, the store is not delayed and the MDP is not updated. After the transient execution, the attacker probes the MDP and infers the secret bit.

Experimental Results and Discussion. We implement the enhanced SpectreRewind on the Cortex-X3. Using our MDP probing primitive, we distinguish between counter values 14 and 0, with a timing difference of 16 ticks, clearly observable in user space. Our proof-of-concept achieves over 98.7% accuracy and a leakage rate of 12.8 kbps with only a single attack (i.e., repetitions to reduce noise are not required). Similar to other transient attacks [26], [37], [38], practical exploit scenarios include leaking secrets from browser space or bypassing ASLR. We leave these exploitation as future work.

E. Improvement of TikTag

TikTag. TikTag exploits transient execution to break Arm's Memory Tagging Extension (MTE) [39]. MTE appends a 4-bit tag to the upper bits of a pointer and verifies tag correctness during memory accesses [5]. On failure, it raises an exception to defend against memory corruption attacks. However, if the access occurs during transient execution, no exception is raised, and the validity of the address affects the length of the transient window. As shown in Fig. 16(a), the attacker speculatively accesses a guessed tagged address, followed by another access with a valid address. If the tag guess is correct, the second address is prefetched and cached [39]. Through a cache side channel, the attacker can enumerate all 16 tag values to infer the correct one. TikTag relies on the prefetch, requiring extensive pre-training to ensure both speculation and prefetch are triggered. We reproduce TikTag on Cortex-X3 and observe that it requires over 80 training iterations and 100 repetitions per guess to reach reliable inference.

Attack Improvement. We observe that TikTag's high overhead stems from prefetcher noise. To address this, we replace the prefetch-based channel with the MDP side channel.

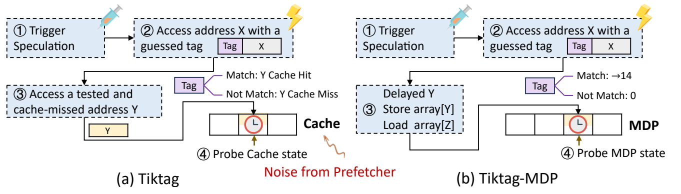


Fig. 16. Basic principle of TikTag (a) and our improvement (b). We replace the cache side channel with MDP to effectively suppress noise from the prefetcher, achieving high accuracy with fewer training and probing attempts.

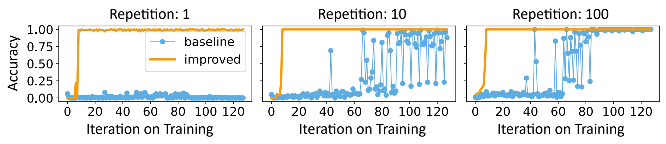


Fig. 17. Evaluation of the improved TikTag. Compared to the cache side channel (baseline), using MDP significantly improves accuracy with fewer repetitions and minimal training iterations.

As shown in Fig. 16(b), after accessing the guessed address, we insert a store-load pair. The store-load pair executes transiently and updates the MDP only if the guess is correct.

Experimental Results and Discussion. We implement the improved TikTag on Cortex-X3. As shown in Fig. 17, our method achieves 100% accuracy with only 8 pre-training iterations and a single trial. The 8 iterations are needed to induce branch misprediction. Reducing training and repetition times lowers the attack's detectability and complicates sampling-based defenses [53], [89].

F. Discussion

Other exploitations of SSBleed. As a new leakage source on Armv9, MDP can cross security-domain boundaries, such as sandbox and virtual machine, which gives it practical impact on real systems. At the system level, SSBleed-v1 can also break certain system-level defenses such as ASLR [74] and KASLR [35]. Moreover, by exploiting speculative execution, SSBleed-v3 can undermine Arm-specific pointer-safety defenses such as Pointer Authentication [62], enabling classic memory vulnerabilities to be re-exploited on Armv9 platforms. At the compiler level, SSBleed-v2 can be adapted to infer the function call chain, because compilers may insert additional loads at different instructions when handling function calls with more than eight arguments, large structs, or variadic arguments. At the application level, SSBleed-v1 and v3 can achieve effects on exfiltrating secrets from sandboxes and VMs [27], breaking other control-flow-dependent cryptographic algorithms such as post-quantum schemes [25], and leaking partial tokens fed to LLMs [36].

Comparison with other potential leakage sources. Although components such as branch predictors [82], [86] and prefetchers [16], [66] on x86 can enable similarly fine-grained control-flow attacks, additional reverse engineering efforts are required for Armv9 branch predictors and prefetchers. Similar attacks may also be implemented with other predictors such as address predictors [38] and value predictors [37], but a more comprehensive reverse engineering of these predictors is required on Armv9 CPUs. Finally, compared with established microarchitectural side channels, SSBleed achieves byte-granularity control-flow leakage, which is substantially finer than that of cache [44] (64 B) or TLB [62] (4 KB).

TABLE IV
EVALUATION OF PERFORMANCE OVERHEAD OF DEFENSE IN SPEC2017

Benchmark ID	500	502	503	505	508	510	519
Overhead / %	1.02	0.00	0.00	0.40	0.00	0.00	0.00
Benchmark ID	520	523	525	531	541	544	557
Overhead / %	1.83	0.86	0.72	0.42	0.00	0.00	1.15

VI. MITIGATION

Our study shows that existing defenses are ineffective against SSBleed. For example, although ASLR randomizes the load IP, it does not affect the lower 16 bits used by the MDP as the tag. In addition, although SSBS prevents speculative loads from leaking data [5], it does not block the attack primitives described in Section IV. A straightforward defense would be to adjust SSBS to either completely disable SSB or entirely disable the MDP, similar to the SSBS design on Armv8 CPUs [47]. While effective, this approach incurs significant performance overhead and would require a hardware or microcode update.

Our Defense. We propose a software defense that prevents cross-process and cross-privilege SSBleed attacks. Inspired by prior research [16], [82], [86], we implement a kernel patch to flush the MDP during context switches, thereby preventing MDP state from leaking across processes or privileges. Based on our reverse-engineering results, we prepare s store-load pairs in the kernel with distinct load tags, where s is the number of entries in MDP. We first execute each store-load pair with 3 `SA` to evict the existing MDP entries via external-eviction. Then, for each store-load pair, we follow with 15 `DA` to trigger self-eviction. This ensures a full flush of the MDP.

Results and Discussion. We deploy this defense on the Microsoft Azure D2ps v6 instance and observe that all cross-process and cross-privilege attacks are no longer effective. We also evaluate the performance overhead using SPEC2017, with results shown in Table IV. The geometric mean performance overhead is below 0.46%, which is acceptable in most scenarios. To further show the effectiveness, we use a Armv8 machine with the interface to fully disable the MDP, and evaluate the performance with MDP on and off using SPEC 2017. Disabling MDP caused an average overhead of 4.3% (500.perlbench_r reaches 12%), indicating that even if the MDP can be fully diablsed on Armv9, the performance overhead are non-negligible.

Other software defenses. For SSBleed-v1 and v2, the loads from different program paths can be merged to eliminate control-flow leakage [45]. For SSBleed-v3, the gadgets with store-load pairs can be detected with software analysis, and barriers can be inserted to block potential leakage [55], [91].

TABLE V
COMPARISON OF MDP SECURITY ON DIFFERENT ARCHITECTURES

Architecture	**MDP Features and Security**			
	State Machine	Entry Collision	Single Load	SSBD/SSBS Bypass
Intel [59]	✔	✔	✗	✗
AMD [48]	✔	✔	✗	✗
Armv8 [47]	✗	✔	✗	✗
Armv9 (This Work)	✔	✔	✔	✔

Other hardware defenses. SSBleed-v1 and v2 can be mitigated using partitioning [84], randomization [77], or context-switch flushing [75], while SSBleed-v3 can be mitigated by delaying speculative commits [87]. More broadly, secure microarchitectures should isolate components across domains and prevent speculative updates to shared states, though approaches incur notable performance overhead. We leave hardware implementation for future work.

VII. RELATED WORK

A. MDP Research on Other Architecture

Prior to this work, several studies have investigated MDP-like structures on other CPU architectures. Ragab et al. [59] present the first systematic reverse engineering of Intel's Memory Disambiguation Unit (MDU), demonstrating its exploitation in a cross-process Spectre-v4 attack. Liu et al. [46] further show that MDU is not isolated in Intel SGX, and propose a control-flow attack named MDPeek. Liu et al. [48] also identify two similar predictors on AMD Zen 3 CPUs, used for predictive store-to-load forwarding and SSB, respectively. Based on these predictors, they propose an out-of-place Spectre-v4 attack. Additionally, they find that Armv8 CPUs include an MDP without a complex state machine [47].

Compared to the previous studies, this work is the first to uncover and reverse-engineer the MDP on Armv9 CPUs. We find that the Armv9 MDP employs a novel fully-associative design and poses more severe security risks than similar predictors on other architectures, which is summarized in Table V. First, Armv9's MDP enables non-speculative side-channel attacks, giving it security implications beyond transient attacks. Second, while SSB predictors on Armv8 and x86 can be disabled via SSBS or SSBD [34], [47], [48], Armv9's MDP remains active even when SSBS is enabled, making SSBleed unmitigated by existing defenses. Third, MDPeek [46] requires a store-load pair inside SGX enclaves to detect load execution. In contrast, SSBleed detects a single load from user space, significantly expanding the attack surface for control-flow leakage.

B. Other Optimization of Load Instruction on Arm CPUs

In addition to the MDP, Arm and Apple CPUs also employ various mechanisms to accelerate load execution. To reduce cache misses, Arm and Apple introduce several data prefetchers, including stride prefetcher [15], Spatial Memory Stream (SMS) prefetcher [66], and Data Memory-dependent Predictor (DMP) [13], [74]. In addition, Kim et al. [38] discover a Load Address Predictor (LAP) and a Load Value Predictor (LVP) [37] on Apple CPUs that optimize loads. These mechanisms expose various security vulnerabilities and have been exploited in side-channel attacks. By studying the MDP, this work broadens the attack surface and presents the first byte-level control-flow attack on Armv9, while also significantly enhancing the practicality of several existing transient attacks. The MDP exhibits better noise resistance when compared to the cache [44], prefetcher [39], [66] and other known side channels on Arm CPUs.

VIII. CONCLUSION

In this work, we discover that Armv9 CPUs implement Speculative Store Bypass (SSB) using a previously undocumented predictor, the Memory Dependency Predictor (MDP). We reverse-engineer its design, including the state machine, organization, and replacement policy. We also identify that the MDP is shared across processes and privilege levels, can be updated by loads without preceding stores, and can be updated during transient execution. Based on these findings, we propose SSBleed, the first non-speculative side-channel attack exploiting MDP. We demonstrate its effectiveness and practicality through 5 case studies targeting both control-flow and data-flow leakage. Finally, we propose a software defense that flushes MDP on context switches.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments. This work was partially funded by the National Natural Science Foundation of China (Grant No. xxxxxx) and the Singapore Ministry of Education (Grant No. xxxxxx). Chang Liu was supported by China Scholarship Council (Student No. 202406210249).

REFERENCES

[1] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 65–74.
[2] R. M. Al Sheikh and R. Damodaran, "Providing load address predictions using address prediction tables based on load path history in processor-based systems," 2023, uS Patent 11,709,679.
[3] AMD, "Security analysis of amd predictive store forwarind," 2021. [Online]. Available: https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf
[4] T. L. K. Archives, "Mds - microarchitectural data sampling," 2019. [Online]. Available: https://www.kernel.org/doc/html/next/admin-guide/hw-vuln/mds.html
[5] Arm, "Arm Architecture Reference Manual for A-profile architecture." [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest
[6] Arm, "Arm Cortex-X3 Core Technical Reference Manual." [Online]. Available: https://developer.arm.com/documentation/101593/0102
[7] Arm, "Arm Neoverse N2 Core Technical Reference Manual r0p3." [Online]. Available: https://developer.arm.com/documentation/102099/0003
[8] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 971–988.

[9] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, “Speculative interference attacks: Breaking invisible speculation schemes,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 1046–1060.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “Smotherspectre: exploiting speculative execution through port contention,” in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 785–800.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, “Fallout: Leaking data on meltdown-resistant cpus,” in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 769–784.

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” in *USENIX Security Symposium (USENIX Security)*, 2019, pp. 249–266.

[13] B. Chen, Y. Wang, P. Shome, C. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, “GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers,” in *USENIX Security Symposium (USENIX Security)*, 2024, pp. 1117–1134.

[14] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, “PrefetchX: Cross-core cache-agnostic prefetcher-based side-channel attacks,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 395–408.

[15] Y. Chen, A. Pashrashid, Y. Wu, and T. E. Carlson, “Prime+ reset: Introducing a novel cross-world covert-channel through comprehensive security analysis on arm trustzone,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[16] Y. Chen, L. Pei, and T. E. Carlson, “AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 16–32.

[17] Y. C. Chou, D. Chandra, M. Agarwal, and H. Jia, “Shared learning table for load value prediction and load address prediction,” 2024, uS Patent App. 18/764,611.

[18] Y. C. Chou, V. Gautam, W.-H. Lien, K. N. Kothari, and M. Agarwal, “Early load execution via constant address and stride prediction,” 2023, uS Patent 11,829,763.

[19] G. Z. Chrysos and J. S. Emer, “Memory Dependence Prediction Using Store Sets,” in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1998.

[20] S. Deng, B. Huang, and J. Szefer, “Leaky frontends: Security vulnerabilities in processor frontends,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 53–66.

[21] K. Evgeni, S. Guillermo, M. Idan, and D. Jacob, “Counter-based memory disambiguation techniques for selectively predicting load/store conflicts,” 2009, uS Patent 7,590,825.

[22] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[23] J. Fustos, M. Bechtel, and H. Yun, “SpectreRewind: Leaking secrets to past instructions,” in *Proceedings of the Workshop on Attacks and Solutions in Hardware Security (ASHES)*, 2020, pp. 117–126.

[24] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, “SQUIP: Exploiting the Scheduler Queue Contention Side Channel,” in *Symposium on Security and Privacy (SP)*, 2023, pp. 2256–2272.

[25] S. Gast, H. Weissteiner, R. L. Schröder, and D. Gruss, “CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP,” in *Network and Distributed System Security Symposium (NDSS)*, 2025.

[26] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, “Speculative probing: Hacking blind in the Spectre era,” in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 1871–1885.

[27] J.-C. Graf, S. Rüegge, A. Hajiabadi, and K. Razavi, “VMSCAPE: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments,” in *Symposium on Security and Privacy (SP)*, 2026.

[28] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks,” in *USENIX Security Symposium (USENIX Security)*, 2018, pp. 955–972.

[29] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+Flush: A Fast and Stealthy Cache Attack,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016, pp. 279–299.

[30] L. Hetterich and M. Schwarz, “Branch different-spectre attacks on apple silicon,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2022, pp. 116–135.

[31] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, “ShadowLoad: Injecting State into Hardware Prefetchers,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2025, pp. 1060–1075.

[32] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, “Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 321–347, 2020.

[33] Intel, “Fast Store Forwarding Predictor.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html

[34] Intel, “Speculative Store Bypass.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html

[35] H. Jang, T. Kim, and Y. Shin, “Sysbumps: Exploiting speculative execution in system calls for breaking kaslr in macos for apple silicon,” in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 64–78.

[36] G. Jia, A. Wong, and A. Khandelwal, “Found in translation: A generative language modeling approach to memory access pattern attacks,” in *USENIX Security Symposium (USENIX Security)*, 2025, pp. 7957–7975.

[37] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, “FLOP: Breaking the Apple M3 CPU via False Load Output Predictions,” in *USENIX Security Symposium (USENIX Security)*, 2025.

[38] J. Kim, D. Genkin, and Y. Yarom, “SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon,” in *Symposium on Security and Privacy (SP)*, 2025.

[39] J. Kim, J. Park, S. Roh, J. Chung, Y. Lee, T. Kim, and B. Lee, “ TikTag: Breaking ARM’s Memory Tagging Extension with Speculative Execution ,” in *Symposium on Security and Privacy (SP)*, 2025, pp. 4063–4081.

[40] S. S. Kim and A. Ros, “Effective context-sensitive memory dependence prediction,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 515–527.

[41] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[42] L. Li, H. Yavarzadeh, and D. Tullsen, “Indirector:High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor,” in *33rd USENIX Security Symposium (USENIX Security)*, 2024, pp. 2137–2154.

[43] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996, pp. 138—-147.

[44] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “ARMageddon: Cache attacks on mobile devices,” in *USENIX Security Symposium (USENIX Security)*, 2016, pp. 549–564.

[45] C. Liu, S. Feng, Y. Li, D. Wang, and T. E. Carlson, “HoBBy: Hardening Unbalanced Branches against Control Flow Attacks on Intel SGX and AMD SEV,” in *Design Automation Conference (DAC)*, 2025, pp. 1–7.

[46] C. Liu, S. Feng, Y. Li, D. Wang, W. He, Y. Lyu, and T. E. Carlson, “MDPeek: Breaking Balanced Branches in SGX with Memory Disambiguation Unit Side Channels,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2025, pp. 622–638.

[47] C. Liu, Y. Lyu, H. Wang, P. Qiu, D. Ju, G. Qu, and D. Wang, “Leaky MDU: ARM Memory Disambiguation Unit Uncovered and Vulnerabilities Exposed,” in *Design Automation Conference (DAC)*, 2023, pp. 1–6.

[48] C. Liu, D. Wang, Y. Lyu, P. Qiu, Y. Jin, Z. Lu, Y. Zhang, and G. Qu, "Uncovering and Exploiting AMD Speculative Memory Access Predictors for Fun and Profit," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 31–45.

[49] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Symposium on security and privacy (SP)*, 2015, pp. 605–622.

[50] Mbedtls, "Mbedtls version 3.6.2," https://github.com/Mbed-TLS/mbedtls/tree/v3.6.2, 2025. [Online]. Available: https://github.com/Mbed-TLS/mbedtls/tree/v3.6.2

[51] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level Attacks on Enclaves," in *USENIX Security Symposium (USENIX Security)*, 2020, pp. 469–486.

[52] K. H. Mose, S. S. Kim, A. Ros, T. M. Jones, and R. D. Mullins, "MASCOT: Predicting Memory Dependencies and Opportunities for Speculative Memory Bypassing," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2025, pp. 59–71.

[53] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2018, pp. 1–8.

[54] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing," in *Symposium on Security and Privacy (SP)*, 2023, pp. 1737–1752.

[55] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security Symposium (USENIX Security)*, 2020, pp. 1481–1498.

[56] L. E. Olson, Y. Eckert, and S. Manne, "Specialized memory disambiguation mechanisms for different memory read access types," 2016, uS Patent 9,524,164.

[57] A. Perais and A. Seznec, "Cost effective speculation with the omnipredictor," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018, pp. 1–13.

[58] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security Symposium (USENIX Security)*, 2021, pp. 663–680.

[59] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks," in *USENIX Security Symposium (USENIX Security)*, 2021, pp. 1451–1468.

[60] F. Rauscher and D. Gruss, "Cross-core interrupt detection: Exploiting user and virtualized ipis," in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[61] F. Rauscher, A. Kogler, J. Juffinger, and D. Gruss, "Idleleak: Exploiting idle state side effects for information leakage," in *Network and Distributed System Security Symposium (NDSS)*, 2024.

[62] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 685–698.

[63] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1998, pp. 127–137.

[64] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 361–374.

[65] S. Rüegge, J. Wikner, and K. Razavi, "Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions," in *USENIX Security Symposium (USENIX Security)*, 2025.

[66] T. Schlüter, A. Choudhari, L. Hetterich, L. Trampert, H. Nemati, A. Ibrahim, M. Schwarz, C. Rossow, and N. O. Tippenhauer, "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers," in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 975–989.

[67] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. L. N. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating software-based keystroke timing side-channel attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2018, p. 15.

[68] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 753–768.

[69] T. Sha, M. M. Martin, and A. Roth, "Nosq: Store-load communication without a store queue," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2006, pp. 285–296.

[70] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 131–145.

[71] S. Subramaniam and G. H. Loh, "Store vectors for scalable memory dependence prediction and scheduling," in *International Symposium on High-Performance Computer Architecture (HPCA).*, 2006, pp. 65–76.

[72] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 989–1007.

[73] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.

[74] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *Symposium on Security and Privacy (SP)*, 2022, pp. 1491–1505.

[75] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: Mitigating branch predictor side-channels," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 466–477.

[76] D. Weber, F. Thomas, L. Gerlach, R. Zhang, and M. Schwarz, "Indirect Meltdown: Building Novel Side-Channel Attacks from Transient-Execution Attacks," in *European Symposium on Research in Computer Security (ESORICS)*, 2023, pp. 22–42.

[77] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium (USENIX Security)*, 2019, pp. 675–692.

[78] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 3825–3842.

[79] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2023, pp. 49–61.

[80] wolfSSL, "wolfSSL Version 5.7.6-stable," https://github.com/wolfSSL/wolfssl/tree/v5.7.6-stable, 2025. [Online]. Available: https://github.com/wolfSSL/wolfssl/tree/v5.7.6-stable

[81] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *USENIX security symposium (USENIX security)*, 2014, pp. 719–732.

[82] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 770–784.

[83] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution," in *Symposium on Security and Privacy (SP)*, 2023, pp. 1220–1237.

[84] L. Yin, H. Wang, Y. Lyu, C. Hu, and D. Wang, "VeriCache: Formally Verified Fine-Grained Partitioned Cache for Side-Channel-Secure Enclaves," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2025.

[85] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1999, pp. 42–53.

[86] J. Yu, T. Jaeger, and C. W. Fletcher, "All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–14.

[87] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *Proceedings of the Annual*

International Symposium on Microarchitecture MICRO, 2019, pp. 954–968.

[88] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 7267–7284.

[89] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016, pp. 118–140.

[90] X. Zhang, Z. Zhang, Q. Shen, W. Wang, Y. Gao, Z. Yang, and J. Zhang, "Segscope: Probing fine-grained interrupts via architectural footprints," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 424–438.

[91] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking speculative load hardening to the next level," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 7125–7142.

[92] Y. Zhu, B. Chen, Z. N. Zhao, and C. W. Fletcher, "Controlled preemption: Amplifying side-channel attacks from userspace," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2025, pp. 162–177.

Artifact Appendix

A. Abstract

This artifact consists of two components. First, it includes the code used to reverse-engineer the MDP on Neoverse-N2 CPUs, corresponding to Section III and Section IV of the paper. Second, it contains the Proof-of-Concepts (PoCs) of the SSBleed attacks, including SSBleed-v1, SSBleed-v2, and SSBleed-v3, corresponding to Section V of the paper.

For the reverse engineering, the artifact demonstrates the methodology and results for analyzing the existence of the MDP (Section III-A), its state machine (Section III-B), its indexing mechanism and prediction table size (Section III-C), and its replacement policy (Section III-D). These findings serve as the building blocks of the SSBleed attacks. For the PoCs, the artifact demonstrates the feasibility of three SSBleed variants, including cross-process control-flow leakage using SSBleed-v1, interrupt detection using SSBleed-v2, and the update of MDP by transiently executed store–load pairs using SSBleed-v3.

B. Artifact check-list (meta-information)

- **Program:** Codes for reverse-engineering the MDP properties, and PoCs to demonstrate the effectiveness of SSBleed-v1, SSBleed-v2 and SSBleed-v3.
- **Run-time environment:** Microsoft Azure D2ps v6 instance, with Ubuntu 24.04, Linux kernel version 6.14.0, Python 3.12, gcc 13.3.0, make 4.3.
- **Hardware:** Arm Neoverse-N2 CPU.
- **Execution:** For the reverse engineering, python scripts are used to automatically run the tests and print the outputs or generate the figures. For the SSBleed-v1 and SSBleed-v3 PoCs, binaries are executed directly. For the SSBleed-v2 PoC, two semi-automated scripts are executed to generate the figure.
- **Output:** For the reverse engineering, four figures will be generated, corresponding to Fig. 2, Fig. 3 and Fig. 9. In addition, two text outputs are printed in the command line, corresponding to Table I and Fig. 4. For the PoCs, two text outputs are printed, demonstrating the effectiveness of SSBleed-v1 and SSBleed-v3. In addition, one figure will be generated, corresponding to Fig. 14, demonstrating the effectiveness of SSBleed-v2.
- **Disk space required (approximately):** 100 MB.
- **Time required to prepare the workflow (approximately):** 1 minute.
- **Time required to complete experiments (approximately):** 1 hour.
- **Publicly available?:** Yes.
- **Code license (if publicly available):** Apache-2.0 License.
- **Archived (provide DOI)?:** 10.5281/zenodo.17854610

C. Description

1) How to access: The PoCs can be accessed from Zenodo: https://zenodo.org/records/17854610 or from Github: https://github.com/CPU-Security/SSBleed.

2) Hardware dependencies: The PoCs depend on MDP properties specific to Arm Neoverse-N2 CPUs.

3) Software dependencies: A C compiler is required. For example, we use `gcc 13.3.0` with `make 4.3` to build the PoCs. No specific kernel or package dependencies and installations are required. To generate plots, matplotlib and seaborn packages of `python` are required. To perform SSBleed-v2, a kernel module is required.

4) Code organization: The code is organized in two main directories, corresponding to the reverse engineering and PoCs, respectively.

- **reverse-engineering**: contains the reverse engineering codes of MDP's existence, state machine, indexing mechanism, table size and replacement policy proposed in our paper.
- **proof-of-concepts**: contains the PoCs that demonstrate the effectiveness of SSBleed-v1, SSBleed-v2 and SSBleed-v3 attacks.

D. Installation

No specific installations are required. We recommend to use the `Makefile` in the artifact to build the executable files.

E. Evaluation and expected results

The reverse-engineering code includes tests for MDP existence, state machine, indexing mechanism, prediction table size, replacement policy, and probe primitives. These tests require compiling the source code first and then running Python scripts individually, each of which generates the expected outputs. The existence test, indexing mechanism test, prediction table size test, and probe primitive test each produce four figures,corresponding to Fig. 2, Fig. 3, and Fig. 9 in the paper. The state machine test produces the text output shown in Table I, and the replacement policy test produces the text output, corresponding to Fig. 4 in the paper.

The PoC code includes the SSBleed-v1, SSBleed-v2, and SSBleed-v3 PoCs. SSBleed-v1 and SSBleed-v3 are executed directly as binaries and print demo results, accuracy, and throughput of the cross-process control-flow leakage and transient-execution attacks to the terminal. SSBleed-v2 requires running two shell scripts sequentially: the first script obtains valid load tags for monitoring, and the second script performs interrupt detection. The final output corresponds to Fig. 14 in the paper.

For additional information on the building steps, running steps, and expected outputs, please refer to the `README.md` files in the respective subdirectories.