Uncovering and Exploiting AMD Speculative Memory Access Predictors for Fun and Profit

Chang Liu[†], Dongsheng Wang[†‡], Yongqiang Lyu[†], Pengfei Qiu[§],
Yu Jin[§], Zhuoyuan Lu[§], Yinqian Zhang[¶], Gang Qu[‖]

[†]Tsinghua University, cliu21@mails.tsinghua.edu.cn, {wds, luyq}@tsinghua.edu.cn
[‡]Zhongguancun Laboratory
[§]Beijing University of Posts and Telecommunications, {qpf, jinyu, luzhuoyuan}@bupt.edu.cn
[¶]Southern University of Science and Technology, yinqianz@acm.org
[‖]University of Maryland, College Park, gangqu@umd.edu

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00014

***Abstract*—This paper presents a comprehensive investigation into the security vulnerabilities associated with speculative memory access on AMD processors. Firstly, employing novel reverse engineering techniques, our study uncovers two key predictors, namely the Predictive Store Forwarding Predictor (PSFP) and the Speculative Store Bypass Predictor (SSBP), along with elucidating their internal structures and state machine designs. Secondly, our research empirically confirms that these predictors can be deliberately manipulated and altered during transient execution, resulting in secret leakage across security domains. Leveraging these discoveries, we propose innovative attacks targeting these predictors, including an out-of-place variant of Spectre-STL and an entirely new form of Spectre attack named Spectre-CTL. Finally, we establish experimentally that enabling Speculative Store Bypass Disable alleviates the vulnerabilities. However, this comes at the expense of significant performance degradation.**

I. INTRODUCTION

Speculative execution is an essential approach that effectively reduces performance penalties caused by pipeline stalls. Branch prediction is a prominent example of speculative execution, whereby unresolved branches are speculatively executed using branch predictors. This enables earlier execution of instructions following the branch, provided that the prediction is accurate. It's worth noting that branch prediction is just one instance of speculative execution, as there exists another significant category called speculative memory access. In this type, operations such as data store, data load, or micro assists are performed before the data address is generated. This ensures that slow memory access does not hinder the execution of subsequent instructions.

However, the improper implementation of speculative memory access can potentially lead to significant security issues, such as data leakage and data injection. One type of speculative memory access is facilitated by the Line Fill Buffer (LFB), which effectively handles cache misses. LFB holds the load operations that miss in the cache, making the associated cache line available for other loads. On Intel CPUs, the LFB performs speculative data forwarding. Even if only part of the data address matches the tag, the LFB speculatively sends the ready data to a load. Such speculative data forwarding can lead to data leakage, and certain attacks exploit this behavior in LFB to forward secret data through carefully constructed faulty loads, as demonstrated in RIDL [48] and Zombieload [44]. These vulnerabilities are commonly known as Microarchitectural Data Sampling (MDS) vulnerabilities. Intel processors have been proven vulnerable to transient execution attacks exploiting variants of such MDS vulnerabilities, such as Cacheout [49], Fallout [14], Crosstalk [42], and LVI [12]. Fortunately, AMD processors have been found to be immune to these MDS vulnerabilities [8].

Another important category of speculative memory access involves store-to-load forwarding (STLF) and store bypassing, which is the focus of this paper. STLF accelerates loads that share the same data address as preceding stores; store bypassing permits out-of-order execution of loads when preceding stores have not retired. False STLF can give rise to a new type of transient attack referred to as Spectre-STL [26], affecting both Intel and AMD processors.

In Spectre-STL, a slow store can be bypassed by a load that follows it, allowing the data of the store to be transmitted to the load before the data address is generated, even if the store and load have different target addresses. As noted in a previous study [29], Spectre-STL is an *in-place* attack, which requires an attacker to repeatedly execute a store-load instruction pair with the same data address for lots of times before triggering a false store-to-load forwarding on that store-load pair. This means the attack requires a shared address space between the adversary and the victim, which makes it less practical. As such, it is yet unclear whether practical transient execution attacks exploiting speculative memory access are feasible on AMD processors.

Inspired by an AMD white paper that describe predictive forwarding, which suggests the potential for an *out-of-place* attack due to the limited size of predictors [6], the goal of this research is to conduct a comprehensive investigation into the security vulnerabilities associated with speculative memory access on AMD processors. Towards this end, we first reverse engineer the predictors employed in the speculative memory access on AMD processors. We uncover the involvement of

Corresponding authors: Yongqiang Lyu and Pengfei Qiu.

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00014

two distinct predictors serving different purposes. The first predictor, comprising three counters, determines whether the data from a store can be forwarded to a subsequent load before the store's data address is generated. As this behavior corresponds to Predictive Store Forwarding (PSF) [6], we name this predictor as PSFP (PSF Predictor). The second predictor, comprising two counters, governs whether a load can be executed ahead of a slower preceding store and whether the data is fetched from the store buffer or the data cache. As this behavior corresponds to Speculative Store Bypassing (SSB) [2], we name this predictor as SSBP (SSB Predictor). We further study the state machines and the structures of these predictors, and identifies the hash function that is used to select the predictor entries.

Based upon these reverse engineering efforts, we conduct a systematic analysis of the security of these predictors. While our experimental results confirm some of the statement provided in the AMD public document [6], such as both predictors are isolated between two hyperthreads of the same physical core and PSFP is flushed during context switches, our analysis uncovers several new vulnerabilities. First, SSBP lacks adequate isolation among processes, enabling cross domain data leakage. Second, SSBP and PSFP can be trained *out-of-place* and trigger false predictions, leading to new variants of Spectre attacks. Third, SSBP and PSFP can be updated during the transient execution, leading to transient attacks without the requirements of cache and shared memory.

To effectively showcase the threats posed by these newly discovered vulnerabilities, we present a series of security attacks. Specifically, we develop an out-of-place Spectre-STL attack by training PSFP with an out-of-place store-load pair. Moreover, we present a novel Spectre attack that exploits SSBP to trigger transient execution and recover secrets fetched in the transient window. We call it Spectre-CTL (Cache-To-Load) since the speculative load fetches data from the cache or memory during the transient execution. We also show that the vulnerabilities lead to application fingerprinting across security domains. Finally, our experimental results suggest that while Speculative Store Bypassing Disable (SSBD) can mitigate most of the reported vulnerabilities, it comes at the cost of significant performance degradation.

Responsible Disclosure. All vulnerabilities discussed in this paper have been disclosed to AMD's security team. AMD has officially acknowledged our findings and confirmed the existence of these vulnerabilities. Given our study, AMD has emphasized the critical importance of enabling SSBD to mitigate data leakage through these predictors.

Contributions. This paper makes several contributions:

- It presents the first comprehensive reverse engineering effort of speculative memory access predictors, namely PSFP and SSBP, on AMD processors.
- It performs a systematic security analysis on these predictors and identifies several vulnerabilities that can be exploited in typical settings of transient execution attacks.
- It proposes several novel attacks exploiting these vulnerabilities on AMD processors, including the first out-of-place Spectre-STL attack and a new Spectre-CTL attack.

II. BACKGROUND

A. Store Queue and Predictive Store Forwarding

Most modern CPUs that support out-of-order execution are designed based on Tomasulo algorithm [47]. A store queue, also known as a store buffer, is used to hold the address and data of a store that has been issued but not yet completed. The store queue guarantees that memory writes are performed in order, and prevents pipeline stalls caused by slow memory writes. By asynchronously handling memory writes while executing other independent instructions, the store queue effectively hides the latency of memory writes. On AMD CPUs, the size of the store queue varies across different microarchitectures. For instance, the store queue has up to 48 entries on AMD 17th family CPUs [4], and up to 64 entries on AMD 19th family CPUs [3].

Store-to-load forwarding, based on the store queue, is a widely recognized technique that speeds up memory access in read-after-write (RAW) scenarios. When a load instruction has the same address as a preceding store instruction, the load can retrieve the data directly from the store after the address of the store has been generated but before the store completes. To further improve the performance for store-to-load forwarding, AMD implements a technique called Predictive Store Forwarding (PSF). PSF uses a predictor to anticipate whether a load has the same address as a preceding store. If the prediction indicates a match, the data from the store is directly forwarded to the load even before the data address of the store is generated [6]. The design of this predictor, which we refer to as PSFP, has not been publicly disclosed. We uncover its design and functionality in our study.

B. Predictive Store Bypass

Read-after-write does not always occur for every contiguous store-load pair. If the RAW does not occur, the load should bypass the store queue and obtain data from cache or memory. However, in certain cases, the data address of the store is generated slow, and the CPU cannot determine whether an RAW occurs for the store-load pair in a short time. For a correct execution, the CPU has to stall the load until the data address of the preceding store is generated, which causes the performance losses.

In order to avoid such stalls and speed up the load, a store bypass predictor known as memory disambiguation unit is involved on Intel and ARM CPUs [34], [41] to predict whether the load is *aliasing* with the store (i.e., the store and load target the same address). The common design of this predictor is shown in Fig 1. The predictor consists of a buffer with numerous entries, and each entry contains a counter-based state machine (f_1) that predicts whether a load is aliasing with its preceding stores. The load selects the entry based on its instruction address (f_2). The update of the chosen state machine depends on whether an RAW occurs (f_3).

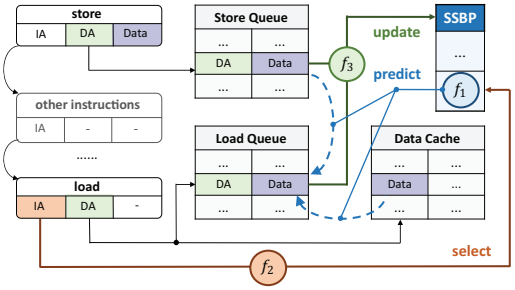


Fig. 1. The common structure and organization of the store bypass predictor on modern CPUs.

AMD claims that the speculative store bypass (SSB) technique is implemented on their processors [2]. However, apart from a brief patent [40], AMD has not publicly disclosed the presence of a predictor similar as memory disambiguation unit that is specifically designed for a *predictive* store bypass. We uncover this predictor in our study, which we refer to as SSBP. Our study reveals that the structure and organization of SSBP share similarities with the illustration shown in Fig 1, but the key functions f_1 and f_3 are significantly different from those disclosed on Intel and ARM CPUs. We present our findings in the subsequent sections of this paper.

III. Reverse Engineering PSFP and SSBP

In this section, we present our effort to uncover the design and organization of PSFP and SSBP. Particularly, PSFP (Predictive-Store-Forwarding Predictor) is used in the predictive store forwarding and SSBP (Speculative-Store-Bypass Predictor) is a predictor used in the speculative store bypass.

A. Experiment Setup

Our experiments is conducted on 4 AMD CPUs including AMD Ryzen 9 5900X, AMD EPYC 7543, AMD Ryzen 5 5600G, and AMD Ryzen 7 7735HS.

We start with a simple microbenchmark shown in Listing 1. The microbenchmark includes a function named *stld*, written in amd64 assembly. This function employs a store-load pair to trigger the utilization of speculative memory access predictors, PSFP and SSBP. This setup allows SSBP to determine whether the load can be executed without waiting for the store, and PSFP to determine whether the store data can be forwarded to the load before the data address is resolved.

```
1  stld:
2  .rep 20
3  imul $1, %rdi      ; delayed store DA generation
4  .endr
5  mov $0x0, (%rdi)   ; store
6  mov (%rsi), %rax   ; load
7  .rep 20
8  imul $1, %rax      ; data-dependent calculations
9  .endr
10 ret
```

Listing 1. A microbenchmark for reverse engineering the predictors.

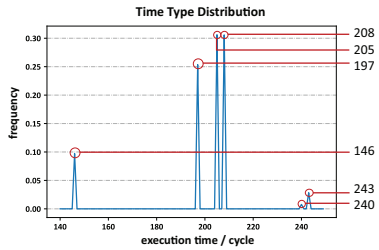
In Listing 1, register `rdi` holds the data address of the store, and `rsi` holds the data address of the load. To facilitate the observation of time differences under different prediction outcomes, we use another 20 `imul` instructions to delay the address generation of the store data. As the execution port is limited [1], [10], even minor pipeline stalls during the load operation lead to substantial time differences. Similar to the previous work [23], we use `RDPRU` instruction to obtain cycle-level execution time of our microbenchmark. `RDPRU` demonstrates a remarkably stable timing. The noise rate consistently remains below 1%. Consequently, all experiment results reported below represent a stable time reading from `RDPRU` that do not require special noise reduction.

For ease of representation, we denote an aliasing *stld* with the same value in `rdi` and `rsi` as a, and a non-aliasing *stld* with the different values in `rdi` and `rsi` as n. According to the document [7], the difference between the two values is greater than 4, so that the CPU treats them as different data addresses. Additionally, we indicate the multiple execution times of *stld*s with a number ahead. For example, sequence $(7n, a)$ means we execute 7 non-aliasing *stld*s and then execute an aliasing *stld*.

B. State Machine

1) Execution type: In order to analyse different execution types resulting from various predictions, we measure the execution time of each *stld* in sequence $(40n, 40a, 40n, 40a)$. Fig 2 displays the time distribution, revealing six types of execution time. By comparing the prediction outcomes and the ground truth, we further classify the execution type into 8 categories. Type A, B and C occur when the prediction of the store-load pair as aliasing is correct, while type D, E and F occur when the prediction of the store-load pair as aliasing is incorrect. Type G occurs when the prediction as non-aliasing is incorrect, and type H occurs otherwise.

To further analyse the cause of different execution time, we use some events in Performance Monitor Counters (PMC) [5]. Some of the typical events are listed in Fig 2. For type A, B, E and F, the prediction is aliasing, and the first event presents that the load is stalled until the data address of the store is generated. On the other hand, for type G and H, the prediction is non-aliasing, and the load bypasses the store without any stalls. For type A, B and G, the truth is aliasing, and the second event presents that the load fetched its data from the store queue. For type E, F and H, the truth is non-aliasing, and the load fetches the data from the data cache or memory. Type C is quite special and no PMC events reveal its unique behavior. Since the prediction is aliasing, and the execution of type C is less than type A and type B, we infer that predictive forwarding occurs in this type, which will be demonstrated in the following section. For type D and G, the execution takes more than 240 cycles, since a rollback is triggered, and the CPU has to fetch and dispatch the instructions following the



Execution Type of Different Predict Cases

	predict as aliasing	predict as non-aliasing
truly aliasing	A (208) B (205) C (197)	G (240)
truly non-aliasing	D (243) E (208) F (205)	H (146)

Some PMC Events in Different Time Types

Performance Monitor Counter Event	A	B	C	D	E	F	G	H
Dynamic Tokens Dispatch for SQ1 Stall Cycles	42	42	42	42	42	42	21	21
Store to Load Forwarding	7	7	6	7	6	6	7	6
Ld Dispatch	41	41	41	44	41	41	44	41
L1 TLB Hits for Instruction Fetch 4K	83	83	83	105	83	83	105	83
Retired Ops	200	200	200	201	200	200	201	200

Fig. 2. Execution time and execution type analysis of the store-load pair in repeated sequences $(40n, 40a)$.

load after flushing the pipeline. The other three events in Fig 1 demonstrate that the rollback happens.

2) Method to reverse engineer the state machine: Having identified the 8 execution types, we proceed to reverse engineer the state machine of these predictors. Specifically, we investigate the execution types for each $stld$ when provided with a sequence of arbitrary n and a.

We assume the predictor is designed as a finite state machine, which has several states and each transition between two states have one input and one output. The input is a $stld$ function (i.e. n or a), and the outcome is an execution type with a corresponding typical execution time (i.e. A to G). For ease of representation, we denote the state machine as ϕ. For example, given a state s, the execution types for sequence $(7n, a)$ are $(7H, G)$, which is presented as $\phi_s(7n, a) = (7H, G)$. For brevity, we omit s in the rest of this paper.

Initially, we model the state machine using a single counter, which is initialized to 0. Next, we detect the execution types of a sequence by measuring the CPU execution time and adjust the state machine to align with the actual execution outcomes. For example, we observe that $\phi(n, a, 7n) = (H, G, 4E, 3H)$, which means that the counter in our model, we denote as C_0, is updated to 4 after an input of a, and it decreases by 1 after an input of n. The corresponding outcomes for inputs n and a when $C_0 = 0$ are H and G, respectively. Additionally, when $C_0 > 0$, the outcome for input n is E.

The state machine is updated and becomes more and more complicated as we consider more sequences. In some cases, the structure of the state machine has to be updated in accordance to our new findings. For example, we observe that $\phi(a, 4n, a, 4n, a, 16n) = (G, 4E, G, 4E, G, 15F, H)$. This sequence shows that another counter is used to record how many types G have happened. However, we cannot model this behavior using only one counter. Therefore, we introduce another counter, which we denote as C_4. When C_4 reaches 3, no less than $(15n)$ has to be executed to make the prediction as aliasing back to non-aliasing.

3) Counter-based state machine: By collecting numerous sequences and modifying the state machine model, we finally get the state machine shown in TABLE I. The input of the state machine is a store-load pair, either non-aliasing (n) or aliasing (a), the outcome is an execution type, and the counters is updated according to current state and the input. The state machine consists of 5 counters with 7 states. We classify the states based on the outcomes and update ways of the state machine model. The state machine can successfully model the behavior of more than 99.8% sequences generated randomly.

According to TABLE I, counter C_0 and C_3 determine whether the prediction is aliasing or non-aliasing. The prediction is non-aliasing only when both C_0 and C_3 are equal to 0. In this situation, we have $\phi(n) = (H)$ and $\phi(a) = (G)$. For the latter, a rollback occurs and changes the prediction from non-aliasing to aliasing. Counter C_1 records how many types D have occurred. A block state is triggered after type D occurs twice. In the block state, the prediction will always be aliasing and both SSB and PSF are disabled. Counter C_2 determines whether to make store forwarding aggressive (i.e. forwarding before the address of the store data is generated). The store forwarding becomes aggressive after executing at least $(4a)$. In the PSF-enabled states, we have $\phi(a) = (C)$ and $\phi(n) = (D)$. For the latter, a rollback occurs and changes the prediction from aliasing to non-aliasing. Counter C_4 records how many types G have occurred. To change the prediction from aliasing to non-aliasing, at least $(4n)$ is required when C_4 is smaller than 3. Otherwise, at least $(15n)$ is required if C_4 reaches 3.

C. Selection of Predictor Entries

In the previous experiments, we use a $stld$ with fixed instruction addresses, and uncover the combined state machine of the speculative memory access predictors. However, a predictor commonly consists of numerous *entries*. Each entry contains a state machine with several counters, and is selected by the instruction address, such as the branch predictor [22] and memory disambiguation unit [30]. In this section, we study how the speculative memory access predictors are organized on AMD CPUs. Specifically, we focus on how $stld$s with different instruction addresses select the entries, and we show our discovery that the counters can be further divided into 2 groups that have different organizations.

1) IPA-dependent selection: On Intel CPUs, the instruction virtual address (IVA) of the load determines which entry to select [41]. However, we cannot observe a similar design on AMD CPUs. To identify the selecting function, we design the following experiments in the Linux kernel.

First, we fix the instruction address of $stld$ and change the data address of the store and load randomly. We find that the same entry is always selected and updated, which indicates that the selection is independent with the data address of a store-load pair. Second, we use the `fork` function to create a new child process with the same address layout with its father process. Due to the Copy-on-Write [11], the $stld$ of the father process and child process share the same IVA and

TABLE I
STATE MACHINE OF SPECULATIVE MEMORY ACCESS PREDICTORS

State Machine	Non-aliasing Store-load Pair (n)		Aliasing Store-load Pair (a)	
	Type	Counter Update	Type	Counter Update
[Initialize] $(C_0 = 0, C_1 = 0, C_2 = 0, C_3 = 0, C_4 = 0)$	**H**	No Changes	**G**	$C_0 \leftarrow 4, C_1 \leftarrow 16, C_2 \leftarrow 2,$ $C_3 \leftarrow$ if $C_4 < 3$ then 0 else 15, $C_4 \leftarrow C_4 + 1$
[Block] $(C_0 > 0, C_2 = 0, C_3 = 0)$	**E**	No Changes	**A**	No Changes
[Load From Cache] $(C_0 = 0, C_2 > 0, C_3 = 0)$	**H**	No Changes	**G**	$C_0 \leftarrow 4, C_1 \leftarrow 16, C_2 \leftarrow 2,$ $C_3 \leftarrow$ if $C_4 < 3$ then 0 else 15, $C_4 \leftarrow C_4 + 1$
[Load From Store Buffer, PSF Enabled, S1] $(C_0 > 0, C_1 \leq 12, C_2 > 0, C_3 = 0)$	**D**	$C_0 \leftarrow C_0 - 1,$ $C_1 \leftarrow C_1 + 4,$ $C_2 \leftarrow C_2 - 1$	**C**	$^*C_0 \leftarrow$ if $C_1\&3 = 3$ then $C_0 + 1$ else $C_0,$ $C_1 \leftarrow C_1 - 1$
[Load From Store Buffer, PSF Disabled, S1] $(C_0 > 0, C_1 > 12, C_2 > 0, C_3 = 0)$	**E**	$C_0 \leftarrow C_0 - 1,$ $C_1 \leftarrow C_1 + 4$	**A**	$C_0 \leftarrow$ if $C_1\&3 = 3$ then $C_0 + 1$ else $C_0,$ $C_1 \leftarrow C_1 - 1$
[Load From Store Buffer, PSF Disabled, S2] $(C_1 > 12, C_3 > 0$ or $C_0 = 0, C_1 \leq 12, C_3 > 0)$	**F**	$C_0 \leftarrow C_0 - 1,$ $C_1 \leftarrow C_1 + 4,$ $C_3 \leftarrow C_3 - 1$	**B**	$C_0 \leftarrow$ if $C_1\&3 = 3$ and $C_0 > 0$ then $C_0 + 1$ else $C_0,$ $C_1 \leftarrow C_1 - 1,$ $^{**}C_3 \leftarrow$ if $C_0 > 0$ then $C_3 - 1$ else $C_3 + 16$
[Load From Store Buffer, PSF Enabled, S2] $(C_0 > 0, C_1 \leq 12, C_2 > 0, C_3 > 0)$	**D**	$C_0 \leftarrow C_0 - 1,$ $C_1 \leftarrow C_1 + 4,$ $C_3 \leftarrow C_3 - 2$	**C**	$C_0 \leftarrow$ if $C_1\&3 = 3$ and $C_0 > 0$ then $C_0 + 1$ else $C_0,$ $C_1 \leftarrow C_1 - 1,$ $C_3 \leftarrow$ if $C_0 > 0$ then $C_3 - 1$ else $C_3 + 16$

* $C_0 \leq 4$ always holds. ** $C_3 \leq 32$ always holds.

physical address (IPA), and we observe that the *stld* in these processes select the same entry. Third, in the child process, we write some dummy data to the page that contains the *stld* by calling the `mprotect` function, which makes this page executable. As a result, although the *stld* of the father process and child process still have the same IVAs, the IPAs are now different since the kernel remaps the page of the child process. This time, we cannot observe the selection collision , which indicates that the physical address has an effect on the selection. Finally, we use a shared memory that holds the *stld* by calling the `mmap` function in two processes. Now the *stld* has the same IPA and different IVAs in these processes, and we observe the collision. Therefore, we can conclude that the selection of the entry depends on the IPA of the *stld*.

2) Hash function: Since the IPA is up to 48 bits, the size of the predictors is too large if the whole IPA is used to select the predictors, and a hash function may be used to compress the IPA before selecting. To prove this, we use a code sliding method to collect the collision addresses (i.e. *stld* at these addresses select and update the same entry), as shown in Fig 3. We first fix a *stld* at an address. Then we obtain the machine code of the *stld*, and fill the machine code into a set of contiguous pages that are mapped using `mmap` function. After that, we execute the *stld* at the fixed address using the sequence $(7n, a, 7n, a, 7n, a)$, and then execute the sliding code using the sequence $(15n)$. We check whether the collision occurs by observing $\phi(15n)$. If $\phi(15n) = (15F)$, the collision occurs. Otherwise, the collision does not occur, and we add the entry address of the *stld* one byte, so that the IPA moves one byte within the page for the next attempt.

By collecting numerous IPAs that select the same entry, we reverse engineered the hash function. We observe that different bits of two colliding addresses at a stride of 12 exhibit identical

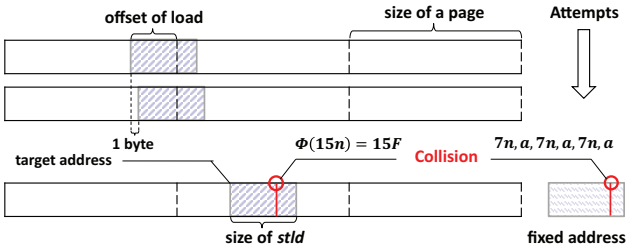


Fig. 3. Code sliding to find collision for the predictors.

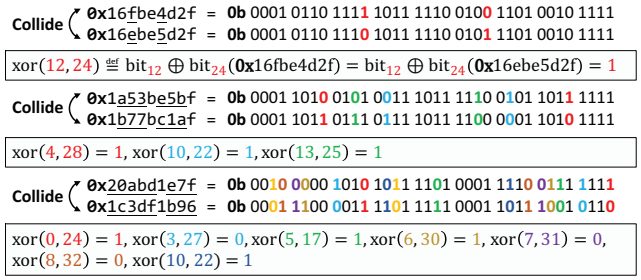


Fig. 4. Mathematical characteristics of the colliding address pairs.

XOR values. For instance, the XOR values of the 12th bit and the 24th bit in the first two colliding address pairs depicted in Fig 4 are both 1. We hypothesize that bits at intervals of 12 can be grouped together to determine the hashed value. We verity our hypothesis through an extensive examination of colliding addresses. Based on our analysis, the hash function consists of 12 `xor` operations, with each `xor` being performed on 4 bits of the IPA at a stride of 12 bits. For example, one of the output bits is the result of `xor` on the 1st, 13th, 25th and 37th

bits of a given IPA. In addition to the hash function, we find that the IPA of the load inside the $stld$, instead of the IPA of $stld$ entry, determines whether the collision occurs.

3) IPA dependence for different counters: In the analysis of hash function, we only study the collision of C_3 because of the sequence we choose. Now we extend our study to other counters. To better label the $stld$s with different IPAs and hashed values, we denote them as n_x^y and a_x^y, where x and y represent the hashed value of the load IPA and the store IPA, respectively. For example, n_0^0 and n_1^0 have the same hashed value of the store IPA and different hashed values of the load IPA. Particularly, n and a represent n_0^0 and a_0^0.

We present some of the important experiments and their corresponding results in TABLE II. Each experiment is conducted with carefully constructed $stld$ sequences that modify the tested counters. For example, when studying the selection mechanism of C_3, we use the sequence $(7n, a, 7n, a, 7n, a, 6a_0^1, 35n)$. The prefix $(7n, a, 7n, a, 7n, a)$ sets C_3 to 15. The following a_0^1 has the same hashed value of the load IPA and different hashed value of the store IPA with a. If we observe that $\phi(6a_0^1, 35n) = (6F, 9F, 26H)$, we can conclude that C_3 is selected by the load IPA only. Otherwise, if we observe that $\phi(6a_0^1, 35n) = (6H, 15F, 20H)$, we can conclude that the selection of C_3 dose not depend on the load IPA only. Since the result shows that $\phi(6a_0^1, 35n) = (6F, 9F, 26H)$, we can conclude that C_3 is selected solely by the hashed value of the load IPA.

The other experiments in TABLE II can be analysed in a similar way. We find that C_0, C_1 and C_2 are selected by the hashed values of both the store IPA and the load IPA, while C_3 and C_4 are selected by the hashed value of the load IPA only. Therefore, The first 3 counters are in the same entry, and support the predictive store forwarding, while the last 2 counters are in another entry, and support the predictive store bypassing. Thus, we conclude that C_0, C_1 and C_2 belong to PSFP and C_3 and C_4 belong to SSBP.

D. Organization of Predictors

We further study the organization of PSFP and SSBP respectively.

1) Organization of PSFP: We already know that the hashed values of both the store IPA and the load IPA are used to select the PSFP entry, and each entry consists of 3 counters. Besides, according to the document [6], PSFP is flushed during a context switch. As a result, it is reasonable to assume that PSFP has a small size. Otherwise, the performance overhead of flushing this predictor would be too high to be acceptable.

To reverse engineer the size of PSFP, we use the sequence $(7n, a, 7n, a, 7n, a, 40n_0^{j_0}, a_{i_1}^{j_1}, a_{i_2}^{j_2}, a_{i_3}^{j_3}, ..., a_{i_k}^{j_k}, 5n)$, where each i and j are non-zero and differ from each other. The prefix $(7n, a, 7n, a, 7n, a)$ initializes an entry of the PSFP, which we denote as the *base entry*. The $stld$ whose hashed values of the store and load IPAs are both 0 selects the base entry. For the base entry, the prefix sequence sets its C_0 to 4. Next, $(40n_0^{j_0})$ is executed to clear C_3 selected by both $n_0^{j_0}$ and n (i.e. n_0^0), which avoids the effects of SSBP. Note that $n_0^{j_0}$ have no effects on the base entry, because the hashed values of the store IPA are different from a and it selects a new PSFP entry instead of the base entry.

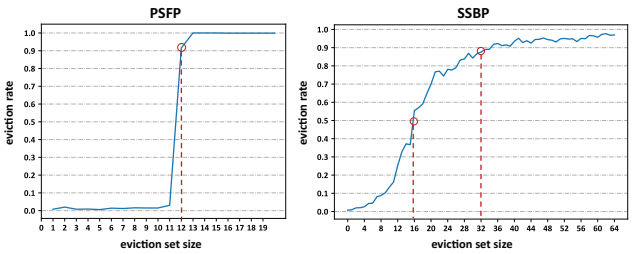


Fig. 5. Eviction rate of PSFP and SSBP under different eviction sizes.

Then we randomly use k different $stld$s to prime the PSFP and try to evict the base entry. By changing k, we effectively build an eviction set with different sizes and observe which size of the eviction set is capable of evicting the base entry. Finally, we observe the execution types of the last $(5n)$ by measuring the execution time. If we observe that $\phi(5n) = (4E, H)$, the base entry is not evicted. Otherwise, we have $\phi(5n) = (5H)$ and the base entry is evicted.

The experiment results are shown in Fig 5. When the eviction size is less than 11, the base entry is not evicted, while when the eviction size is larger than 11, the base entry is consistently evicted. Therefore, we can conclude that the size of PSFP is 12. Since the hashed values in the eviction set are random, it is likely that PSFP is implemented as a 12-entry fully associative buffer, with two 12-bit tags corresponding to the hashed value of the store and load IPAs.

2) Organization of SSBP: We design similar experiments to study the organization of SSBP. The results are shown in Fig 5. Unlike PSFP, SSBP has a complex selection mechanism, and we cannot determine the exact size due to the absence of an abrupt change in the eviction rate. However, we observe some typical changes corresponding to the eviction set size. For example, the eviction rate exceeds 50% when the eviction size is 16, and reaches 90% when the eviction size is 32.

3) Summary: We summarize the organization of PSFP and SSBP in Fig 6. The 48-bit IPAs of the store and load serve as the input to a hash function, resulting in a 12-bit compressed output. The PSFP is 12-way fully associative, consisting of 3 counters C_0, C_1 and C_2, with the hashed IPAs of both the store and load serving as the tags. The SSBP consists of 2 counters C_3 and C_4, and has a more complex selecting function F_2. These 5 counters are combined to form a prediction regarding whether the store-load pair is aliasing and whether to forward the store data to the load before the data address is generated. According our experiments, all 4 AMD Zen 3 CPUs in our study share the same design of PSFP and SSBP.

IV. Security Analysis of PSFP and SSBP

In this section, we conduct an in-depth analysis of the security of PSFP and SSBP. We conduct empirical experiments

TABLE II
SOME IMPORTANT EXPERIMENTS FOR STUDYING THE COUNTER ORGANIZATION

Counter		Experiments							Dependence	
C_0	**seq**	$7n\ 1a$	$7n\ 1a$	$7n\ 1a$	$4a\ 1n\ 4a$	$1n\ 3a$	$6n_0^1$	$35n$	**store IPA**	**load IPA**
	type	7H 1G	4E 3H 1G	4E 3H 1G	4B 1D 4B	1D 3B	6H	5E 30H	✔	✔
C_1	**seq**	$7n\ 1a$	$6n_0^1$	$35n$	$7n1a$	$6a_0^1$	$35n$	-	**store IPA**	**load IPA**
	type	7H 1G	1G 4E 1C	4E 31H	7H 1G	6E	32F 3H	-	✔	✔
C_2	**seq**	$5a\ 1n$	$7n_1^0\ 5a_1^0$	$1n_1^0\ 42n$	$5a\ 1n$	$7n_1^0\ 5a_1^0$	$1n_1^0$	$35n$	**store IPA**	**load IPA**
	type	1G 4E 1D	7H 1G 4E	1D 4E 38H	1G 4E 1D	7H 1G 4E	1D	35E	✔	✔
C_3	**seq**	$7n\ 1a$	$7n\ 1a$	$7n\ 1a$	$6a_0^1$	$35n$	-	-	**store IPA**	**load IPA**
	type	7H 1G	4E 3H 1G	4E 3H 1G	6F	9F 26H	-	-	✗	✔
C_4	**seq**	$4n$	$7n_0^1\ 1a_0^1$	$39n$	$7n_0^1\ 1a_0^1$	$39n$	$7n_0^1\ 1a_0^1$	$35n$	**store IPA**	**load IPA**
	type	4H	7H 1G	39H	7H 1G	39H	7H 1G	15F 20H	✗	✔

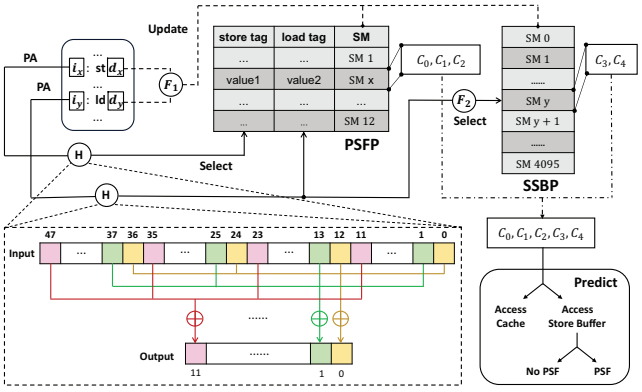


Fig. 6. Overview of the organization of PSFP and SSBP.

TABLE III
CPU AND KERNEL INFORMATION IN VULNERABILITIES ANALYSIS

Processor	Microcode	Kernel
AMD Ryzen 9 5900X (Zen 3)	0xA201205	Linux 5.15.0-76-generic
AMD EPYC 7543 (Zen 3)	0xA001173	Linux 6.1.0-rc4-snp-host-93fa8c5918a4
AMD Ryzen 5 5600G (Zen 3)	0xA50000D	Linux 5.15.0-76-generic
AMD Ryzen 7 7735HS (Zen 3+)	0xA404102	Linux 5.4.0-153-generic

on four platforms (TABLE III) to answer the following questions:

1) Are the predictors well isolated between security domains, e.g., user-kernel isolation and host-VM isolation?
2) Can the predictors be trained out-of-place deterministically? In other words, can we find a collision between the predictions of different store-load pairs?
3) Can the predictors trigger a transient window with attacker-controlled values?
4) Can the predictors be updated during the transient execution?

A. Breaking Isolation

In the in-place experiments, we use a shared executable page between two different security domains. We specifically consider three security domains: a user process in the host OS, a process inside a VM, and a kernel thread. We repeat the following experiments for all three pairs of security domains. We fill a $stld$ in the shared page, train the predictors using this function in a domain, and probe it using this function in the other domain. For PSFP, we use the sequence $(7n, a, 7n, a, 7n, 5a, n, 4a, n, 3a)$ to train the predictor because it sets C_0 to 5 and clears C_3. Then we probe PSFP with the sequence $5n$. For SSBP, we use the sequence $(7n, a, 7n, a, 7n, a)$ to train the predictor, and probe it with sequence $(32n)$. In the out-of-place experiments, we use PTEditor [39] to get the IPA from a given IVA, and find collisions between 2 $stld$s in different address space. Then we use the same sequences mentioned before to observe the state changes of these predictors.

Our experiments confirm that PSFP is well isolated. However, SSBP is not isolated between two security domains, allowing one domain to leak data from another domain. Furthermore, we find that PSFP is flushed during a context switch due to a system call or the `yield` function, which matches the information provided in the official document [6]. However, SSBP is not affected by the context switch, and retains the legacy data from the previous process. Additionally, both SSBP and PSFP are flushed if the process is suspended due to a `sleep` function.

We also study the isolation between two Simultaneous Multi-Threading (SMT) threads by running two processes in two hyperthreads, and find that both SSBP and PSFP are partitioned amongst SMT threads, and thus the activity of one SMT thread does not influence the other thread's predictors. To investigate how CPUs manage the resources of these predictors between the SMT threads, we repeat the experiment mentioned in Section III-D after switching the CPU from SMT mode to single-thread mode. We do not observe a significant change in the eviction size, suggesting that the predictors might be duplicated resources [46].

Vulnerability 1: SSBP is not isolated between two security domains, which means the data from a security domain may be leaked to another domain.

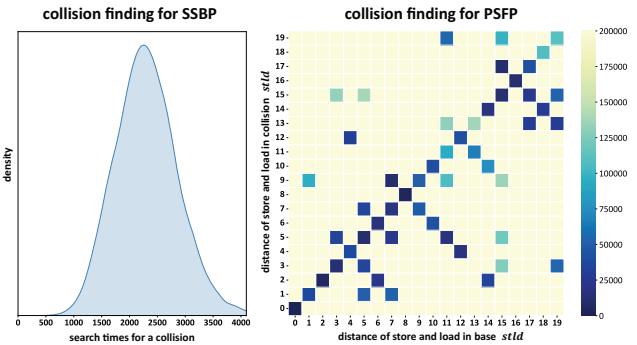


Fig. 7. Collision finding for PSFP and SSBP.

B. Finding Collisions

According to Section III-C, a specially designed hash function takes an IPA as the input and compresses it into a 12-bit value before selecting the predictors. We show that it is easy to find hash collisions for PSFP and SSBP.

1) Collision For SSBP: According to Section III-C, two $stld$s select the same SSBP entry when the hashed values of their load IPA are the same. Assume the hash function is implemented as the way in Fig 6, and we need to find a collision with a given IPA, whose physical page frame is F and page offset is O. Now we prove that the collision for this given IPA can always be found in any executable pages.

The hashed value of the given IPA has 12 bits, and we denote it as h. The ith bit of h can be calculated as:

$$h_i = O_i \oplus F_i \oplus F_{i+12} \oplus F_{i+24}$$

for each i from 0 to 11, where $\oplus$ is an `xor` operation. For a random page $\mathbb{P}$ with its physical page frame denoted as F', the ith bit of its hashed value h' is:

$$h'_i = x \oplus F'_i \oplus F'_{i+12} \oplus F'_{i+24}$$

where x is ith bit of its page offset. Given that the page frame of $\mathbb{P}$ is fixed, $F'_i \oplus F'_{i+12} \oplus F'_{i+24}$ is a fixed value. Therefore, for each i from 0 to 11, it is satisfiable for the ith bit in the page offset of $\mathbb{P}$ to make $h_i = h'_i$. In other words, it requires at most 4096 attempts to find an IPA in any pages that selects the same SSBP entry with another IPA.

To further verify it, we measure the distribution of the number of attempts to find a collision, and the results are shown in the left part of Fig 7. The figure shows that the distribution of the number of attempts follows a Gaussian distribution with an approximate average of 2200.

2) Collision For PSFP: Unlike SSBP, PSFP is selected using both the store and load IPAs. The hashed values of the store and load IPAs serve as the tags to select the PSFP entry, making it much more difficult to find a collision for PSFP.

An intuitive idea is that the distance between the store IPA and the load IPA matters whether a collision can be found. To prove it, we measure the average number of attempts required to find a collision for different IPA distances. A portion of the results is shown in Fig 7. The collision can always be found when the IPA distances are the same for two $stld$s, but the collision may not be found (i.e. number of attempts are more than the upper threshold) if the distances are different. Therefore, it is better to keep the distance equal so that the collision of PSFP can be found deterministically.

> **Vulnerability 2**: Collisions for PSFP and SSBP can be deterministically found, and at most 4096 attempts are required to find a collision for SSBP, which means out-of-place attacks are feasible using these predictors.

C. Transient Execution

Based on the state machine and organization of the speculative memory access predictors, we can train any entries of these predictors to any states and trigger the mispredictions. In this section, we study the behavior of the CPU when a misprediction of PSFP or SSBP occurs.

As shown in Fig 8, we delay the data address generation of the store by performing time-consuming calculations or loading the data address from memory (1). This allows the predictors to be used to predict whether the load can bypass the store and whether the data of the store can be forwarded to the load before its address is generated. For simplification, assume that the DPA of the store is 0xaa, and the data is 0xdd. The DPA of the load is 0xaa (2a) or 0xbb (2b) in different cases. The memory 0xaa contains the value 0xcc.

By training the predictors, we can trigger a misprediction of PSFP (3a) or SSBP (3b). In the misprediction of PSFP, the DPA of the store is predicted as 0xbb, and a predictive store forwarding is performed (4a). In the misprediction of SSBP, the DPA of the store is predicted as another value that is not equal to 0xbb, and then a speculative store bypass is performed to load the data from the data cache or memory (4b).

Before the data address of the store is generated, the CPU does not stall the following instructions, but continues to consume the incorrectly loaded data (5). Since the CPU will find the misprediction and reissue the load later, the execution is referred to as the *transient execution*. To observe the loaded data in the transient window, we use the cache side channel [50] to recover it. When the data address of the store is generated, the CPU identifies a misprediction and triggers a rollback to eliminate the effects of the transient execution (6). After the rollback, we recover the data in the transient window by timing the cache access.

The results indicate that 0xbb is loaded in the transient window triggered by PSFP, and 0xcc is loaded in the transient window triggered by SSBP. Therefore, both predictors can be misused to trigger the transient execution, during which an unexpected value is loaded and consumed.

D. Transient Update

A lot of studies focus on searching microarchitecture covert channels that can be used to recover the data in a transient window. The new covert channels are proposed to bypass the

Vulnerability 3: Both PSFP and SSBP can be misused to trigger the transient execution with an incorrect loaded value, which means an attacker can control any malicious data as an address to fetch secrets using PSFP and SSBP.

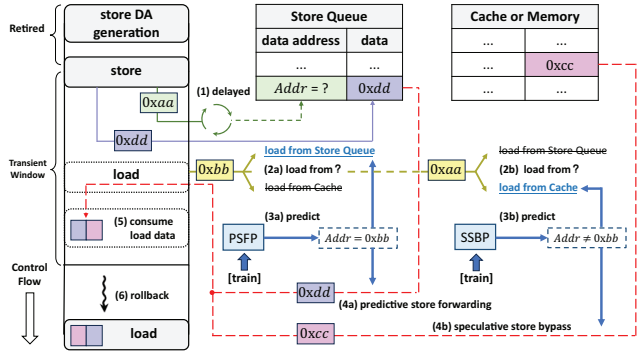


Fig. 8. Transient Execution of PSFP and SSBP.

cache-related defenses [9], [43], make timing easier [34], [51], and expand available gadgets [14], [17]. This inspires us to study whether PSFP and SSBP can serve as the covert channels for transient execution attacks.

As shown in Fig 9, we trigger a transient window in different ways, including a branch misprediction, a faulty load, and a speculative memory access misprediction. In the transient window, we execute a store-load pair and try to update the states of PSFP or SSBP. After the transient execution, we probe the state of PSFP or SSBP using a *stld* that selects the same entry of PSFP or SSBP. The results show that both predictors can be updated in any kind of transient windows, and the update is not rolled back.

Vulnerability 4: Both PSFP and SSBP can be updated during the transient execution and the updates to these predictors are not rolled back, which means that these predictors can be used to construct covert channels for data transmition during transient execution.

V. EXPLOITATION

In this section, we propose novel attacks against AMD's SSBP and PSFP, which includes two new variants of Spectre attacks, out-of-place Spectre-STL and Spectre-CTL, on AMD Zen 3 processors. We also show that SSBP can be misused to perform application fingerprinting.

A. Threat Model

We assume that an attacker and a victim use the same AMD Zen 3 CPU, and the attacker aims to leak secrets, such as the secret data and secret-dependent control flow, from the victim. We do not make any special assumptions about the victim, who can be a normal application or a kernel thread running on any versions of operating system and microcode.

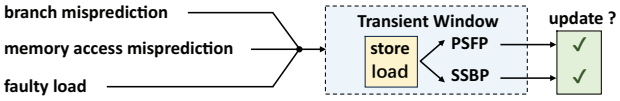


Fig. 9. Transient Execution of PSFP and SSBP.

For most of attacks, we assume that the attacker is a normal user without root privilege. The attacker can execute unprivileged instructions such as `mfence`, `clflush` and `rdpru`. `rdpru` provides the cycle-level timing method, allowing the attacker to measure the execution of any code. Without root privilege, the attacker cannot use both PTEditor and `pagemap` of Linux to get the physical address directly, but has to find the collision by probing the PSFP or SSBP counters.

B. Out-of-place Spectre-STL

Spectre-STL, also known as Spectre V4, has been discovered on AMD CPUs [13]. AMD further claims that Spectre-STL can be executed by misusing a predictor [6]. However, the exploitation is limited to a single process because the predictor is flushed during a context switch. In this paper, we proves that Spectre-STL's implementation is restricted to a single process by uncovering the design and security features of PSFP.

Beside being an inner-process attack, current research also suggests that Spectre-STL can only be exploited in-place [13]. This implies that the attacker needs to execute the same store-load pair multiple times within the victim's address space, so that a false store-to-load forwarding is performed by that particular store-load pair. While AMD claims that out-of-place exploitation is possible, since the associated PSFP is not publicly available, no research has yet discovered a method to implement Spectre-STL using a different store-load pair within the attacker's address space and under the attacker's full control.

In this paper, we first propose an out-of-place approach to misuse PSFP and implement Spectre-STL on AMD Zen 3 CPUs, which extends the attack surface. Both in-place and out-of-place Spectre-STL require the same gadget in the victim's address space as shown in Listing 2. In the gadget, a store that targets address `&array2 + (idx << 12)` is performed with the data `x`. Then three loads are performed following the store. The first load fetches the data from address `&array2`, and the fetched data serves as a new address in the second load to fetch another data stored in `&array1 + array2[0]`. The third load encodes the data fetched by the second load into a cache line, which is a common way to implement the Flush+Reload cache side channel [50].

For in-place attack, in order to train PSFP, the attacker sets `idx` to 0 and executes a lot of `victim_function`. For *out-of-place* attack, however, the attacker tries to find another store-load pair that is fully controlled in attacker space, and trains PSFP, so that only one execution of `victim_function` is required for leaking each secret. The

```
1 void victim_function(size_t x) {
2     array2[idx * 4096] = x;
3     temp = array2[array1[array2[0]] * 4096];
4 }
```

Listing 2. Gadget in Spectre-STL.

```
1 void victim_function() {
2     array2[idx] = 0;
3     temp = array2[array1[array2[idx2]]];
4 }
```

Listing 3. Gadget in Spectre-CTL.

code sliding mentioned in Section III-C is used to find the collision for PSFP. The attacker needs to carefully control the distance of store and load IPAs to be the same with the store-load pair in `victim_function`.

After training, PSFP will predict the store-load pair in `victim_function` as aliasing. The attacker now sets `idx` to another value, sets `x` to reach the address of secret, and executes the `victim_function`. The attacker flushes `idx` from the cache to delay the store, and creates a transient window, as shown in Fig 8. In the transient execution, `x` will be forwarded to the first load, and the second load fetches the secret from `&array1 + x`. The secret will be encoded to a cache line by the third load. Finally, the attacker uses Flush+Reload to recover the secret.

In our implementation, we use 16 pages to search for the PSFP collision, achieving a collision-finding rate of over 90%. We test the accuracy and bandwidth of out-of-place Spectre-STL by leaking 10,000 randomly generated bytes in a user process. The accuracy achieved is 99.95%, with the attack leaking an average of 416 bytes per second (B/s).

C. Spectre-CTL

Spectre-STL, as mentioned earlier, has several limitations. Firstly, the forwarded data is stored in a register, making it easy to being overwritten by other instructions. This necessitates the store operation to be in close proximity to the victim load to be effective. Secondly, Spectre-STL is constrained to operate within a single process. Even though our study extends the attack from in-place to out-of-place, the isolation of PSFP among different processes prevents its application across process boundaries. Thirdly, the recovery of the secret relies on a cache side channel, requiring the secret to be multiplied by a large value in the gadget so that it is distinguishable across different cache lines.

In this section, we present a novel Spectre Attack named Sepctre-CTL, which overcomes the limitations of Spectre-STL by leveraging SSBP. The gadget of Spectre-CTL is illustrated in Listing 3, where the secret address is not required in the gadget, and the secret is not required to multiply a large number, which makes it more feasible to find the gadget within the victim's code.

1) Spectre-CTL Attack in C Code: The attack process is shown in Fig 10. Similar to Spectre-STL, Spectre-CTL requires one store and three loads in the victim's address space. During the train phase, the attacker tries to discover two collisions with the first and the third load of the victim through code sliding. Upon finding these collisions, the attacker proceeds to train the relevant SSBP entries by clearing C_3 so that a misprediction as non-aliasing will occur. Then the attacker sets the first loaded data as the secret's address.

After training, the attacker executes the victim function with `idx=idx2`. The store is delayed by evicting `idx` from the cache, and SSBP gives a misprediction that the first load can bypass the store and fetch data from the cache or memory. In the transient window, the second load fetches the secret. Subsequently, the third load treats the secret as an address, updates the second SSBP entry. C_3 in this entry is updated to 15 if the secret is equal to `idx`, and remains 0 otherwise.

The leak phase is finished when the CPU detects the misprediction and triggers a rollback. Then the attacker probes the second SSBP entry in the recover phase. If the execution type F is observed, it indicates that the secret is equal to `idx`, signifying a successful recovery of the secret.

In Spectre-CTL, for each secret byte, the attacker is required to attempt at most 256 values of `idx` to successfully recover the secret. Due to the complex hash function, very little noise is induced in Spectre-CTL. We test the accuracy and bandwidth of Spectre-CTL by leaking 10,000 bytes randomly generated bytes. The accuracy achieved is 99.97%, with the attack leaking an average of 384 bytes per second (B/s).

Spectre-CTL is much more powerful than Spectre-STL, as it offers a broader scope of applicability and more extensive attack capabilities. Spectre-CTL can be implemented out-of-place and even across different processes, as SSBP is not isolated for individual processes. We successfully exploit the attack to leak secrets from another process or kernel thread. Moreover, despite being named Spectre-CTL, the first load that bypasses the store in the transient window can also fetch data from the memory if the cache miss occurs. This flexibility allows the attacker to control the secret address `array2[idx2]` through various data injection techniques, such as Rowhammer [25]. By incorporating such methods, the attacker's capabilities are further amplified, making the attack more formidable and posing a higher threat level to the targeted system's security.

2) Spectre-CTL Attack in Web Browser: In this section, we demonstrate that Spectre-CTL is a practical and powerful attack in web environments, by implementing the Spectre-CTL attack in Chrome version 86 on AMD Zen 3 CPUs.

Firstly, we verify that the SSBP state can be detected within a web browser. To accomplish this, we implement a high-resolution timer directly within the browser, capable of achieving a timing level at about 10 nanoseconds. This timer enables us to measure the execution time of $stld$ in the web context. We implement $stld$ by using WebAssembly for its flexibility. Our experiment demonstrates that SSBP side channel attack is practical in the context of web browser, and

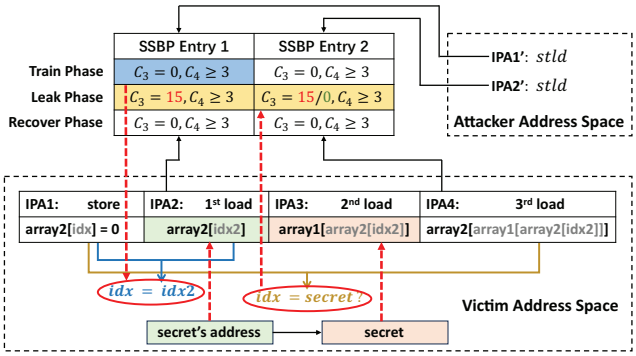


Fig. 10. Overview of the Spectre-CTL Attack.

```
function spectreCTL(trash) {
    spectreArgs[argsIdx[16 * 256]] = 0;
    return probeArray[((
        (spectreArray[spectreArgs[0]]) >> bit
    ) & 1 ) * 0x800];
}
```

Listing 4. JavaScript Gadget in Spectre-CTL.

is an alternative to the commonly used Evict+Reload covert-channel in the browser.

The prior work, leaky page [45], has already demonstrated that JavaScript can be used to implement Spectre-V1 [29], enabling the leakage of information from the browser's memory. In our research, we implement the Spectre-CTL attack in web by modifying the code in leaky page. In specific, we change the gadget from a branch bound check to a store-load pair, which is shown in Listing 4.

In the train phase, we set `spectreArgs[0]` to zero and set `argsIdx[16 * 256]` to a non-zero value so that the store and load is non-aliasing. We perform numerous non-aliasing store-load pairs to clear C_3 of the relevant SSBP entry, which ensures that SSBP predicts the store and load as non-aliasing. In the subsequent attack phase, we assign the address of the secret to `spectreArgs[0]` and set `argsIdx[16 * 256]` to zero. A misprediction occurs during the execution of `spectreCTL`, and the secret is fetched in the transient window. Our Spectre-CTL attack in the web browser has the capability to achieve a data leakage rate of approximately 170 B/s, with the accuracy as 81.1%.

D. Side Channel Impact of SSBP

In addition to transient attacks, SSBP can also be exploited to implement side-channel attacks in two ways. Firstly, because SSBP is not flushed during context switches, the control flow of the load instruction within one process, which has the potential to leak certain secrets [16], can be disclosed to another process via SSBP. Secondly, the hash function contains information about the physical address and may unintentionally leak address mapping from virtual to physical addresses, which is inaccessible to a regular user process in the user space. In this section, we use process fingerprinting to demonstrate the first kind of side channel impact of SSBP.

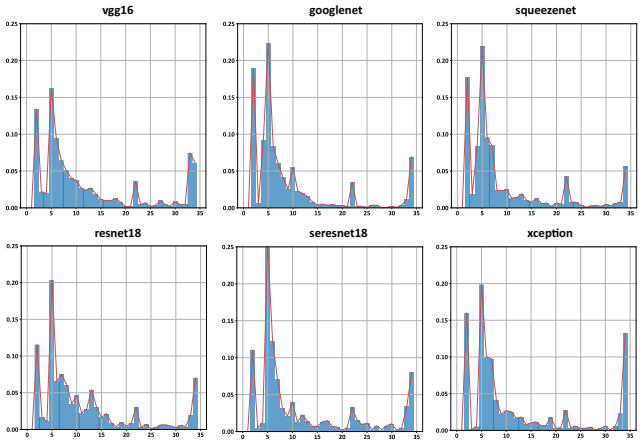


Fig. 11. Fingerprinting results of machine learning model using SSBP side channels.

As the SSBP is selected by the load IPA, and the physical address of the load is not controllable for an unprivileged attacker, it is impossible to observe the execution of a specific load. However, it is still effective to exploit SSBP to build the fingerprinting of a process. To achieve it, we use the code sliding to traverse the entire space of SSBP entries, which amounts to a total of 4096 entries. During each probe round, we collect the C_3 values of each entry, ranging from 0 to 35. Subsequently, we analyse the relative frequency of each data value and aggregate them into a vector containing 35 elements. Each element in the vector represents the relative frequency of the corresponding value, ranging from 0 to 1, and the sum of all elements in the vector totals to 1.

To demonstrate that the fingerprinting is practical and useful, we collect the fingerprinting of different machine learning models. The tested CNN models are running in a victim process, and the attacker binds the probe process on the same CPU. For each probe round, the attacker uses the `sleep` function to yield the CPU. Fig 11 displays the fingerprinting results for 6 distinct CNN models. Several noticeable features can be observed directly from the figure. For instance, the relative frequency of value 5 is distinguishable among vgg16 (0.16), googlenet (0.22), resnet18 (0.20), and sersnet18 (0.25). To quantify these differences and differentiate among the different models, we employ the support vector machine (SVM) provided by the `sklearn` module to classify the models based on their relative frequency vectors. This classification approach yields an accuracy of over 95.5%, indicating the effectiveness of our fingerprinting technique in successfully distinguishing among the various CNN models.

VI. DEFENSE

A. Disable Speculation with SSBD and PSFD

AMD has provided a system register `SPEC_CRTL` to control the speculative execution, including the speculative memory access [7]. In specific, the 2nd bit of `SPEC_CRTL`,

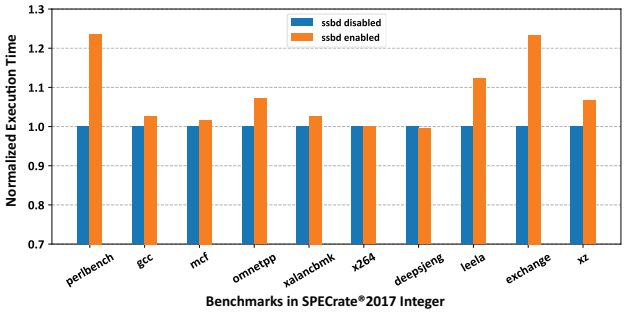


Fig. 12. Performance evaluation of SSBD on SPEC2017.

known as Speculative Store Bypass Disable (SSBD), determines whether the speculative store bypass is disabled. When this bit is set, any load is serialized and must wait until the preceding stores are fully resolved, which includes generating and translating the data addresses.

We conducted experiments using specific $stld$ sequences to investigate whether SSBD can effectively defend against the vulnerabilities mentioned in this paper. We observe that, for all sequences, we have $\phi(n) = E$ and $\phi(a) = A$. This behavior aligns with the block state presented in TABLE I, and thus we can conclude that SSBD fixes all of the SSBP and PSFP entries on the block state. Since the SSBP entries are at the block state, the attacker cannot detect timing differences among $stld$ sequences, and the side channels among different processes are prevented. In addition, the stores and loads are serialized, making it impossible to trigger an exploitable transient window, which prevents both Spectre-STL and Spectre-CTL attacks.

Unfortunately, enabling SSBD has considerable effects on CPU performance, as it introduces stalls for non-aliasing loads. As a result, SSBD is disabled by default in the Linux kernel. To assess the performance overhead of SSBD, we conduct an evaluation using SPEC2017 benchmarks. We executed 10 benchmarks from SPECrate with SSBD disabled or enabled for all CPUs on AMD Ryzen 9 5900X. The execution results can be seen in Fig 12. The evaluation reveals that, for most benchmarks, there is a significant performance overhead when SSBD is enabled. In some cases, the overhead exceeds 20%, as seen in benchmarks like $perlbench$ and $exchange$. In summary, despite the effectiveness of SSBD in mitigating vulnerabilities related to PSFP and SSBP, the notable performance degradation cannot be ignored.

Nevertheless, it is worth noting that Predictive Store Forwarding Disable (PSFD) might not mitigate these attacks. Specifically, AMD offers an additional control of predictive store forwarding, i.e., the 7th bit of `SPEC_CRTL` known as PSFD. However, in all experiment setups outlined in TABLE III, we find that the predictors continue to function even when PSFD is enabled, which suggests that the attacks proposed in this paper cannot be effectively mitigated. We will further investigate the implementation of PSFD and analyze the reasons in our future work.

B. Other Potential Mitigations

Although disabling speculation is a straightforward mitigation, the significant performance loss will hinder their adoption in production systems. We outline a few potential mitigation strategies below.

Develop a secure timer. Developing a more secure timer by introducing timing noise or reducing timing accuracy [18], [38] can effectively render timing differences unobservable, so that the predictor states cannot be probed.

Flush SSBP during context switch. Flushing SSBP during context switches can mitigate cross-process attacks that exploit SSBP, and the associated overhead can be controllable [16].

Randomize selection. Incorporating randomization into the organization of SSBP and PSFP can mitigate most out-of-place attacks because finding collisions between entries becomes more challenging, as demonstrated in secure cache and branch predictor designs [31], [35], [52].

VII. Related Work

A. Transient Execution Attacks

Since 2018, the year of Spectre [29] and Meltdown [33], a lot of transient execution attacks have been found on Intel, AMD and ARM processors. The related study mainly focuses on: (1) new ways to trigger the transient window, such as ret2spec [37], machine clear [41] and a series of MDS attacks [12], [14], [42], [44], [48], [49]; (2) new ways to recover data in the transient window, such as SmotherSpectre [10], mwait [51] and Speculative interference [9]. In this paper, our study covers both aspects and extends the Spectre attack with two new variants, including the out-of-place Spectre-STL attack and Spectre-CTL attack.

B. Side Channel Attacks

Side channel attacks on CPU microarchitecture have been widely studied. Vulnerabilities have been disclosed on a lot of CPU predictors and buffers, including decode string buffer [20], [43], branch predictor [22], [28], load store unit [15], [34], execution port [1], [23], translation look-aside buffer [24], [32] and cache [19], [36]. Due to the complexity of SSBP, there is little study focusing on this predictor, and our study first uncover the side channels that exploit SSBP.

C. Memory Disambiguation Units on Intel and ARM

Predictors used in speculative memory access are mentioned in both patents from Intel [30] and AMD [40], but limited information about the design and organization of these predictors are provided. The first reverse engineering effort on Intel's memory disambiguation units (MDU) is reported in a blog post [27], where well-designed microbenchmarks are used to demonstrate the existence of these predictors on x86 processors. Another post [21] conducts a similar experiment to reverse engineer the MDU design on Intel Skylake CPUs. Based on these findings, Ragab et al. [41] systematically analyses the design and organization of MDU on Intel CPUs. The study also finds that MDU can be misused to trigger

TABLE IV
CHARACTERIZATION OF MDU AND SSBP

Characterization	Intel [41]	ARM [34]	AMD (Our Work)
Feasible State Machine Size	4 bit	1 bit	6 bit (C_3) + 2 bit (C_4)
Selection	Lowest 8 bits of the load IVA/IPA	Lowest 16 bits of the load IVA	Hashed value of the whole load IPA

transient execution. However, it does not investigate whether MDU is effectively isolated among different security domains.

MDUs are also available on ARM processors. Liu et al. [34] uncovers the MDU design on ARM and utilizes MDU to construct side-channel attacks across different security domains. However, Liu et al. [34] did not discuss whether MDU can be used to trigger transient execution.

Our paper significantly extends the prior studies in two aspects. First, it investigates the SSBP design on AMD processors. While SSBP on AMD is similar to MDU on Intel and ARM, our work shows that the design of SSBP is considerably more complicated. TABLE IV provides characterizations of the SSB predictors on Intel, ARM and AMD CPUs. The size of the state machine of SSBP is larger than that of MDU, and the selection of SSBP depends on a complex hash function that considers the entire load IPA, rather than just a portion of the lowest bits of IVA. Second, our work performs a comprehensive analysis on the exploitability of SSBP in various attack settings. Specifically, it examines the capability of SSBP in performing both cross-domain attacks and transient execution attacks, which bridges the gaps between prior studies.

VIII. CONCLUSION

In this paper, we present our investigation efforts and research findings on the security of speculative memory access on AMD processors. Our study has led to better understanding of two predictors, namely PSFP and SSBP, in terms of both their internal organization and their security properties. Our study also presents novel out-of-place Spectre-STL attack and the first Spectre-CTL attack on AMD processors.

ACKNOWLEDGEMENT

This work was supported in part by the National Key Research and Development Program of China (Grant No.2021YFB3100902), the National Natural Science Foundation of China (Grant No. U23B2041, 62072263 and 62372258) and the Fundamental Research Funds for the Central Universities (Grant No. 2023RC71).

REFERENCES

[1] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 870–887. [Online]. Available: https://doi.org/10.1109/SP.2019.00066

[2] AMD. (2019) Speculation behavior in amd micro-architectures. [Online]. Available: https://www.amd.com/system/files/documents/security-whitepaper.pdf

[3] AMD. (2020) Software optimization guide for amd epyc™ 7003 processors. [Online]. Available: https://www.amd.com/system/files/TechDocs/56665.zip

[4] AMD. (2020) Software optimization guide for amd family 17h models 30h and greater processors. [Online]. Available: https://www.amd.com/system/files/TechDocs/56305.zip

[5] AMD. (2021) Processor programming reference (ppr) for amd family 19h model 21h, revision b0 processors (pub). [Online]. Available: https://www.amd.com/en/support/tech-docs/preliminary-processor-programming-reference-ppr-for-amd-family-19h-model-21h

[6] AMD. (2021) Security analysis of amd predictive store forwarind. [Online]. Available: https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf

[7] AMD. (2023) Amd64 architecture programmer's manual. [Online]. Available: https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5

[8] T. L. K. Archives. (2019) Mds - microarchitectural data sampling. [Online]. Available: https://www.kernel.org/doc/html/next/admin-guide/hw-vuln/mds.html

[9] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. V. Rozas, A. Morrison, F. McKeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. R. Alameldeen, "Speculative interference attacks: breaking invisible speculation schemes," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 1046–1060. [Online]. Available: https://doi.org/10.1145/3445814.3446708

[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 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 785–800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[11] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[12] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 54–72. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00089

[13] C. Canella, J. V. 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 *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 769–784. [Online]. Available: https://doi.org/10.1145/3319535.3363219

[15] A. Chakraborty, N. Singh, S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Timed speculative attacks exploiting store-to-load forwarding bypassing cache-based countermeasures," in *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, R. Oshana, Ed. ACM, 2022, pp. 553–558. [Online]. Available: https://doi.org/10.1145/3489517.3530493

[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 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 16–32. [Online]. Available: https://doi.org/10.1145/3575693.3575719

[17] M. H. I. Chowdhuryy and F. Yao, "Leaking secrets through modern branch predictors in the speculative world," *IEEE Trans. Computers*, vol. 71, no. 9, pp. 2059–2072, 2022. [Online]. Available: https://doi.org/10.1109/TC.2021.3122830

[18] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 2009, pp. 45–60. [Online]. Available: https://doi.org/10.1109/SP.2009.19

[19] Y. Cui, C. Yang, and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 2022, pp. 82–97. [Online]. Available: https://doi.org/10.1109/HPCA53966.2022.00015

[20] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 2022, pp. 53–66. [Online]. Available: https://doi.org/10.1109/HPCA53966.2022.00013

[21] T. Downs. (2021) Memory disambiguation on skylake. [Online]. Available: https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake

[22] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 693–707. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[23] 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 *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 468–484.

[24] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[25] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016, pp. 300–321. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15

[26] Z. He, G. Hu, and R. B. Lee, "New models for understanding and reasoning about speculative execution attacks," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 40–53. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00014

[27] Henry. (2014) Store-to-load forwarding and memory disambiguation in x86 processors. [Online]. Available: https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/

[28] 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 Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 1, pp. 321–347, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i1.321-347

[29] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[30] Krimer *et al.*, "Counter-based memory disambiguation techniques for selectively predicting load/store conflicts," 2009, uS Patent 7,590,825.

[31] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, pp. 21:1–21:25, 2020. [Online]. Available: https://doi.org/10.1145/3404189

[32] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "TLB poisoning attacks on AMD secure encrypted virtualization," in *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 2021, pp. 609–619. [Online]. Available: https://doi.org/10.1145/3485832.3485876

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[34] 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 *DAC '23: 60th ACM/IEEE Design Automation Conference, 2023*. San Francisco, USA: ACM, 2023.

[35] F. Liu and R. B. Lee, "Random fill cache architecture," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. IEEE Computer Society, 2014, pp. 203–215. [Online]. Available: https://doi.org/10.1109/MICRO.2014.28

[36] M. Luo, W. Xiong, G. Lee, Y. Li, X. Yang, A. Zhang, Y. Tian, H. S. Lee, and G. E. Suh, "Autocat: Reinforcement learning for automated exploration of cache-timing attacks," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 317–332. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10070947

[37] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[38] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*. IEEE Computer Society, 2012, pp. 118–129. [Online]. Available: https://doi.org/10.1109/ISCA.2012.6237011

[39] misc0110. (2018) Pteditor. [Online]. Available: https://github.com/misc0110/PTEditor/

[40] L. E. Olson, Y. Eckert, and S. Manne, "Specialized memory disambiguation mechanisms for different memory read access types," Dec. 20 2016, uS Patent 9,524,164.

[41] 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 *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 1451–1468. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ragab

[42] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1852–1867. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00020

[43] 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 *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 361–374. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00036

[44] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 753–768. [Online]. Available: https://doi.org/10.1145/3319535.3354252

[45] A. J. Stephen Röttger. (2021) A spectre proof-of-concept for a spectre-proof web. [Online]. Available: https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[46] M. Taram, X. Ren, A. Venkat, and D. M. Tullsen, "Secsmt: Securing SMT processors against contention-based covert channels," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3165–3182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/taram

[47] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[48] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 88–105. [Online]. Available: https://doi.org/10.1109/SP.2019.00087

[49] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 339–354. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00064

[50] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[51] 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*, 2023.

[52] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," in *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 1267–1272. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586178

APPENDIX A
ARTIFACT APPENDIX

A. Abstract

The artifact comprises two Proof of Concepts (PoCs): the out-of-place Spectre-STL attack detailed in Section V-B andthe Spectre-CTL attack detailed in Section V-C. These PoCs are implemented based on our reverse engineering analysis of SSBP and PSFP, including the state machine presented in Section III-B and the selection mechanism outlined in Section III-C. Building upon the reverse engineering and security analysis, we propose the attacks following the process illustrated in Fig 8. In the out-of-place Spectre-STL attack, we mistrain PSFP to trigger the store-to-load forwarding transient execution and use Flush+Reload cache side channel to recover the secret bytes fetched in the transient window. In the Spectre-CTL attack, we mistrain SSBP to trigger the store bypass transient execution and use SSBP as the covert channel (as shown in Fig 9) to recover the secret byte. The PoCs are easy to build and execute, requiring no special software environment. We validate the effectiveness of the PoCs in all the environments listed in TABLE III.

B. Artifact check-list (meta-information)

- **Algorithm:** Code sliding (Section III-C and Section IV-B) for collision finding, specific $stld$ memory access sequences (Section V) for predictors mistraining, and SSBP covert channel (Section IV-D) for secret recovering
- **Program:** Out-of-place Spectre-STL attack and Spectre-CTL attack PoCs
- **Compilation:** gcc
- **Run-time environment:** x86-64 Linux Kernel
- **Hardware:** AMD Zen3 CPUs
- **Execution:** Execute an executable file
- **Output:** Command line string
- **Experiments:** Leak secrets through out-of-place Spectre-STL attack and Spectre-CTL attack
- **Publicly available:** Yes
- **Code licenses (if publicly available):** Apache-2.0 License
- **Data licenses (if publicly available):** None
- **Archived**: DOI 10.5281/zenodo.10199277

C. Description

1) How to access: The PoCs can be accessed from Zenodo: https://zenodo.org/records/10199277 or from Github: https://github.com/CPU-THU/Spectre-V4-ng.

2) Hardware dependencies: The PoCs depend on SSBP and PSFP functionalities specific to AMD Zen 3 CPUs, and CPUs with a design similar to that of SSBP and PSFP are anticipated to execute the PoCs successfully. We have tested the PoCs successfully on four CPUs listed in TABLE III.

3) Software dependencies: A C compiler is required. For example, we use `gcc 9.4.0` with `make 4.2.1` to build the PoCs. No specific kernel or package dependencies and installations are required.

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 PoCs demonstrate the transient execution vulnerabilities of PSFP and SSBP. In the out-of-place Spectre-STL attack, the PoC demonstrates the step-by-step process of leaking the victim's secret string byte by byte by finding the collision of PSFP, mistraining PSFP, and triggering transient execution. In the Spectre-CTL attack, the PoC showcases the process of leaking the victim's secret string byte by byte by finding the collision of SSBP, mistraining SSBP, triggering transient execution, and recovering secrets using SSBP. For more detailed information on the attack implementation and expected results, please refer to the `README` files.