PREFENDER: A Prefetching Defender against Cache Side Channel Attacks as A Pretender

Luyi Li†, Jiayi Huang‡, Lang Feng†*, Zhongfeng Wang†*
†*School of Electronic Science and Engineering, Nanjing University*
‡*Department of Electrical and Computer Engineering, University of California, Santa Barbara*
luyli@smail.nju.edu.cn; jyhuang@ucsb.edu; flang@nju.edu.cn; zfwang@nju.edu.cn;

***Abstract*—Cache side channel attacks are increasingly alarming in modern processors due to the recent emergence of Spectre and Meltdown attacks. A typical attack performs intentional cache access and manipulates cache states to leak secrets by observing the victim's cache access patterns. Different countermeasures have been proposed to defend against both general and transient execution based attacks. Despite their effectiveness, they all trade some level of performance for security. In this paper, we seek an approach to enforcing security while maintaining performance. We leverage the insight that attackers need to access cache in order to manipulate and observe cache state changes for information leakage. Specifically, we propose PREFENDER, a secure prefetcher that learns and predicts attack-related accesses for prefetching the cachelines to simultaneously help security and performance. Our results show that PREFENDER is effective against several cache side channel attacks while maintaining or even improving performance for SPEC CPU2006 benchmarks.**

***Index Terms*—Security, Cache Side Channel Attacks, Prefetcher**

I. INTRODUCTION

The threats of cache side channel attacks [1], [2] to modern processors are growing rapidly due to the vulnerabilities of advanced microarchitecture optimizations. For example, recent Spectre [3] and Meltdown [4] attacks can exploit the transient execution enabled by branch prediction, speculative execution, and out-of-order execution to create cache timing side channels for information leakage. These vulnerabilities were found in processors from Intel, AMD, and ARM that are used in billions of devices. Even worse, many new attack variants have been introduced in the past few years, making cache side channel attacks a serious problem [5].

Cache side channel attacks can exploit the timing variation of shared cache accesses for information leakage [6]. For example, the timing difference of a fast cache hit and a slow cache miss can reveal a victim's cache footprint [3], [4]. Different countermeasures have been proposed for either general or transient execution based attacks through isolation [7], conditional speculation [8], stateless mis-speculative cache accesses [9], and noise injection [10], [11]. Despite their effectiveness, they either trade some level of performance for security or ignore the performance effect.

*Corresponding authors.

This work was supported in part by the National Natural Science Foundation of China under Grant 61774082, in part by the Fundamental Research Funds for the Central Universities under Grant 2021300341, and in part by the Key Research Plan of Jiangsu Province of China under Grant BE2019003-4.

In this work, we seek an approach to enforcing security while maintaining performance. Note that victims need to access cache to change cache states while attackers need to access cache to manipulate and observe cache state changes. If we can learn their access patterns, we can predict the future accesses and prefetch data to change the cache state, thereby confusing the attacker. From a performance perspective, effective prefetching can also save execution time for benign programs.

We propose PREFENDER, a prefetching defender to defeat cache side channel attacks while preserving performance benefits for benign programs. Specifically, we design a low-cost Data Scale Tracker to track the address calculation of memory instructions to guide prefetching address prediction targeting victim accesses. We also propose an Access Pattern Tracker to learn the cache access pattern targeting attacker accesses by leveraging the insight that only a few `loads` are intensively used for attack. This helps correlate even intentional random attacker accesses for address prediction. Furthermore, effective prefetching also maintains or improves performance. This work includes the following main contributions:

- We propose PREFENDER, the first work that can defeat cache side channel attacks while maintaining or even improving performance, to the best of our knowledge.
- We propose a new approach to analyzing cache access patterns, and design Data Scale Tacker (DST) and Access Pattern Tacker (APT) to realize the runtime analysis for effective prefetching.
- Our detailed security and performance evaluation shows that PREFENDER is not only effective for cache side channel defense, but also highly compatible with other prefetchers for performance improvement.

II. BACKGROUND AND THREAT MODEL

A. Cache Side Channel Attacks

Cache side channel attacks can infer secrets of victim programs from the cache state changes made by the victim's execution. Such an attack typically consists of three phases. In phase 1, the attacker initializes the cache state by performing cache accesses. In phase 2, the victim may access the cache and change the cache state. In phase 3, the attacker measures the cache state change to infer the victim's secret.

One widely used attack is timing-based side channel attack. Fig. 1 shows the Flush+Reload [2], Evict+Reload [12], and Prime+Probe [6] attacks. For the Flush+Reload attack, in phase

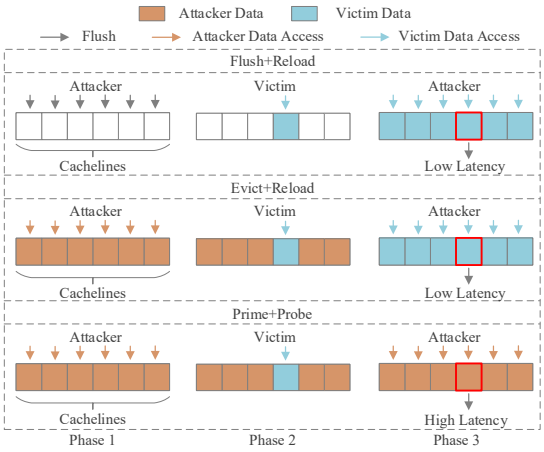

Fig. 1. The examples of Flush+Reload, Evict+Reload, and Prime+Probe. The secret can be revealed by the only low (or high) latency eviction cacheline.

1, the attacker flushes all the cachelines that may be accessed by the victim (through shared libraries). These cachelines form an *eviction set*, and each of them is called an *eviction cacheline*. In phase 2, the victim loads the secret-dependent data to the cache. In phase 3, the attacker accesses the eviction set and times each eviction cacheline's access latency. Then, the low hit latency of the secret-dependent cacheline reveals the secret. For example, assuming the cacheline size is 64 bytes, if the victim loads a secret-dependent shared data `array[s×64]` in phase 2, where `s` is the secret. During phase 3, the attacker accesses `array[768]` with a low cache hit latency; it can infer the secret is `s=768/64=12`. Compared with Flush+Reload, Evict+Reload and Prime+Probe differ in the way of phase 1 and phase 3 but share the same key idea.

B. Threat Model

Our threat model includes the cache timing side channel attacks described in Section II-A. The attackers can manipulate the state of any cacheline (usually the eviction set) and measure the cacheline access latency. The victim may either share data with the attacker through shared libraries (for flush-based attacks) or conflict with the attacker's eviction set. The attacker can exploit the timing difference of eviction cachelines' access latencies to infer the secret of the victim.

III. Related Work

To mitigate cache side channel attacks caused by speculative execution, prior work constrains the execution of speculative loads, such as Conditional Speculation [8] and SpecShield [13]. Another category such as InvisiSpec [9] and SafeSpec [14] design a shadow structure to temporarily hold the data brought by speculative loads during transient execution. However, they are ineffective in defending against general cache side channel attacks. As a result, new cache policies were introduced. DAWG [7] dynamically partitions cache ways to avoid cache sharing among different security domains. SHARP [15] designs a new cache replacement policy to prevent the spy from evicting the victim's data. All these approaches pay some level of performance for the security strength.

Prefetch-guard [10] and Reuse-trap [11] propose several methods to detect the spy and leverage prefetching to obfuscate the spy based on previously recorded information. Prefetch-guard needs a large amount of history tracking registers to record all the flushed memory lines and replaced memory lines, which may cost high hardware overhead. Reuse-trap needs to know the victim's process ID in advance to record the victim's cache misses, which may cause software modifications. Last but not the least, they both overlook the potential performance gains that can be achieved with prefetcher.

By contrast, PREFENDER is a completely hardware-based and resource-efficient method without modifying any policy of speculative execution or cache in the modern processors. On the premise of ensuring security , it further achieves a performance enhancement through accurate runtime analyses and well-designed hardware prefetching strategies.

IV. PREFENDER Design

In this section, we present the overview of our proposed system with PREFENDER shown in Fig. 2 and detail the design of Data Scale Tracker (DST) and Access Pattern Tracker (APT).

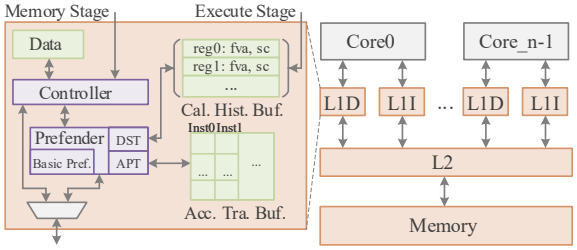

Fig. 2. The overall design architecture of our system.

A. Overview

According to Section II-A, a cache side channel attack has three phases. If one of the phases is interfered with, we can defend against the attack. PREFENDER is designed in the L1Dcache for phase interference by prefetching the eviction cachelines. Specifically, PREFENDER includes *Data Scale Tacker (DST)* and *Access Pattern Tacker (APT)* to interfere with the second and third phases, respectively. In PREFENDER, we also support a basic prefetcher (Basic Pref. in Fig. 2), which is a conventional prefetcher such as the Tagged or Stride prefetcher. All the three including DST, APT, and the basic prefetcher can prefetch data. Note that the basic prefetcher can only help with performance, while DST and APT can enforce security and also improve performance to some extent.

DST aims to predict the eviction cachelines during the victim execution (phase 2). The prediction is based on the arithmetic calculation histories of the victim instructions, which are stored in the *Calculation History Buffer* (Cal. Hist. Buf. in Fig. 2). After a victim instruction loads a secret-dependent cacheline (e.g., an eviction cacheline in Flush+Reload), DST will predict another eviction cacheline and prefetch it. The prefetched eviction cacheline can be disguised as a secret-dependent cacheline brought by the victim to fool the attacker, which is illustrated in the example on the top of Fig. 3.

APT aims to predict the access patterns of the eviction cachelines during the attacker measurement stage (phase 3). The patterns are stored in the *Access Trace Buffer* (Acc. Tra. Buf. in Fig. 3). As shown in the example at the bottom of

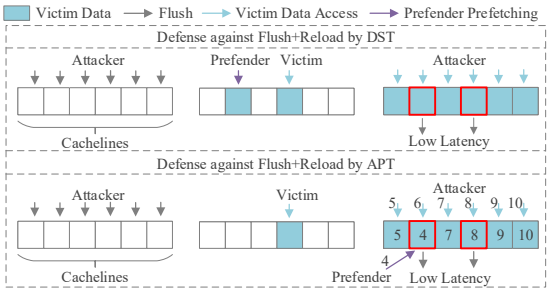

Fig. 3. The example of the defenses against Flush+Reload attacks (The number near an arrow represents the access time, and the number inside each rectangle represents the first time when the corresponding cacheline is accessed).

Fig. 3, APT seeks to prefetch the cacheline before the attacker accesses and measures it, which also misleads the attacker.

Since the key idea of DST and APT is to learn cache access patterns for prefetching, effective prefetching on benign loads can also improve performance while enforcing security. However, there are two major challenges for effective prefetching for DST and APT, respectively.

C1. During phase 2, the victim may only access one secret-dependent eviction cacheline. Even though more secret-dependent cachelines are accessed, they may not be simply contiguous. How to effectively predict the access pattern given limited accesses (even single access) is challenging, which we overcome with DST.

C2. During phase 3, the attacker tends to randomly measure the eviction cachelines to bypass prefetchers, such as Stride prefetcher. Learning the eviction set given a random access pattern is challenging, which we tackle using APT.

B. Data Scale Tacker

DST predicts the eviction cachelines by tracking how the `load`'s target address is calculated. For example, if the target address is calculated by `128×i` for a `load`, where `i` is an integer variable, the target address can only be `0`, `128`, `256`, etc. After address translation, if the target physical address of the `load` is `paddr`, we can infer that this `load` may access addresses `paddr-128`, `paddr`, `paddr+128`, etc., if they are in the same page. Therefore, we can leverage this pattern to predict the eviction cachelines. The key question is how to track and learn `128` as in this example, which we call a *scale*.

Since the target address of a `load` is usually calculated and stored in registers, DST needs to track how the register values are calculated. The best way is to record all the calculations related to each register. But this is impractical due to high hardware cost. So we only consider two common calculations: addition (and subtraction) and multiplication (and shifting).

We use two values to track the history for each register r: A fixed value fva_r and a scale sc_r, which are stored in the calculation history buffers. Essentially, we maintain fva_r to help track the scale. If the register r's calculation history only depends on constant values (immediate numbers), the calculated result is assigned to fva_r. Otherwise, fva_r is not applicable (NA); for example, when the register value depends on a loaded memory value, fva_r is NA.

Scale sc_r is tracked to predict the cache access pattern. In common cases, array access address in a loop is usually calculated as `base+scale×i` (e.g., `base+128×i`), where `base` is the base address, `i` is an index, and `scale` is the stride. Such an address calculation typically propagates through several registers and finally to the one that is used by a `load` instruction to access the array. Our goal is to track the scale value with the help of fva_r by propagating them from source registers to destination registers. With sc_r, we can predict the nearby cachelines in the access pattern even with a single access.

Complicated access patterns such as `128×i+32×j+imm` can exist, where `i` and `j` are indices and `imm` is an immediate. Given an `imm`, if there is a pair of `i` and `j` makes the result to be `652`, there may be another pair (e.g., `i` increments 1) to make the result as `652+128`. The `128` can be sc_r in this calculation. Similarly, `32` and any multiples of them like `256`, `512`, etc., can also be sc_r. Note that a more complicated access pattern that involves multiplications of several registers' scales (such as $(128i_0i_1i_2+32j_0\times16j_1)\times(48k_0+imm)$) can also be handled.

TABLE I

The rules to calculate fva_{rd} and sc_{rd}. (rd is the destination register; "-" means not applicable. †The rule is also for subtraction when $+$ is replaced by $-$. ‡The rule is also for shifting when $\times$ is replaced by $>>$ or $<<$.)

	Conditions				Results	
Instruction	Arg. a	Arg. b	fva_{rs_0}	fva_{rs_1}	fva_{rd}	sc_{rd}
`load rd a`	imm_0	-	-	-	imm_0	1
	$imm(rs_0)$	-	-	-	NA	1
`add rd a b`†	rs_0	imm_0	NA	-	NA	sc_{rs_0}
	rs_0	imm_0	Valid	-	$fva_{rs_0}+imm_0$	1
	rs_0	rs_1	Valid	Valid	$fva_{rs_0}+fva_{rs_1}$	NA
	rs_0	rs_1	NA	Valid	NA	sc_{rs_0}
	rs_0	rs_1	Valid	NA	NA	sc_{rs_1}
	rs_0	rs_1	NA	NA	NA	$min(sc_{rs_0}, sc_{rs_1})$
`mul rd a b`‡	rs_0	imm_0	NA	-	NA	$sc_{rs_0}\times imm_0$
	rs_0	imm_0	Valid	-	$fva_{rs_0}\times imm_0$	1
	rs_0	rs_1	Valid	Valid	$fva_{rs_0}\times fva_{rs_1}$	NA
	rs_0	rs_1	NA	Valid	NA	$sc_{rs_0}\times fva_{rs_1}$
	rs_0	rs_1	Valid	NA	NA	$fva_{rs_0}\times sc_{rs_1}$
	rs_0	rs_1	NA	NA	NA	$sc_{rs_0}\times sc_{rs_1}$
Otherwise	-	-	-	-	NA	1

We propose a set of rules to calculate sc_r (and fva_r, which can help calculate sc_r) as shown in Table I. The fixed value and scale of the destination register rd are calculated using the source operand and the propagated values of the source registers. The fixed and scale values are initialized to NA and 1 respectively upon starting a program.

For data movement instructions, if an immediate number is loaded to rd, fva_{rd} is set to the number. If a value is loaded from memory to rd, va_{rd} and sc_{rd} are reinitialized since we conservatively regard the loaded value as an unknown variable.

For addition, when va_{rd} is calculated by one immediate number and one register rs_0, if rs_0's fva_{rs_0} is NA, sc_{rd} is the same as sc_{rs0} since adding the immediate number as the offset has no effect on the scale. If fva_{rs_0} is valid, fva_{rd} is the addition of fva_{rs_0} and the immediate number since both are fixed values. When adding two registers, if only one of them has a valid fixed value, the scale of the destination register is the same as the scale of the source register without a valid fixed value. If neither of the source registers has a valid fixed value, the scale of the destination register can be the minimum scale of the two registers. The reason is that when the values of two registers are added, both scales can be used as the new scale. However, using the minimum one can reduce the possibility of making the scale larger than a page.

For multiplication, the calculations of fva_{rd} and sc_{rd} are similar to those of addition, except the consideration of multiplicative factors due to multiplication. If any other calculations

are involved, to be conservative, the destination register of the calculation is reinitialized.

The prefetching policy of DST is as follows. When an instruction `load rd imm(rs)` (or the equivalent instructions such as `mov rd 12(rs)`) is executed, assuming the calculated address is $addr$, the candidate prefetching addresses are $addr + sc_{rs}$ and $addr - sc_{rs}$. If sc_{rs} is larger than the cacheline size, one of the candidate addresses currently not in the L1Dcache and is in the same page as $addr$ is chosen for prefetching. Note that, to be conservative, all the `load` instructions are assumed to be vulnerable and applied with DST prefetching. For implementation, since DST prefetches data in the same page, the bitwidth for storing and calculating fva and sc can be small (Section V-E).

C. Access Pattern Tracker

In phase 3 of the attack, all the eviction cachelines of the victim instruction are timed for their access latencies. To mislead the attacker, besides using the DST to interfere with phase 2, we propose an Access Pattern Tacker (APT) to directly interfere with phase 3. The key idea of APT is to learn the patterns of the timed accesses, and prefetch the cachelines before they are accessed and timed by the attacker.

According to challenge C2, attackers usually use random instead of sequential cacheline accesses to bypass prefetchers (e.g. Stride prefetcher), making it difficult to learn the access pattern. One key observation is that these random accesses are associated with only a few `load` instructions; this helps correlate the cache accesses that are initiated by the same `load`.

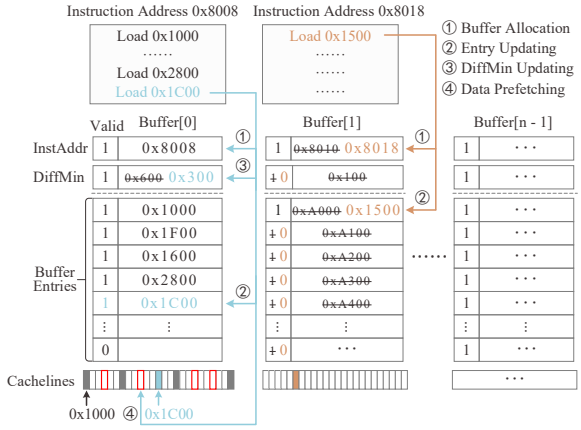

Fig. 4. An example of the access trace buffer.

To defend against the attacks, we propose APT, which consists of a set of access trace buffers. Each buffer is associated with a `load` instruction and records the block addresses accessed by the associated `load`. The purpose of the access trace buffer is to infer the access pattern of each `load`. The access pattern is estimated as an arithmetic sequence with a constant difference, which is estimated as the minimum difference between block addresses recorded in the buffer.

Fig. 4 shows the access trace buffer microarchitecture. Each buffer keeps the instruction address (`InstAddr`) of its associated `load`, and maintains a `DiffMin` register. Each entry in a buffer records the block address (`BlkAddr`) accessed by the `load`. The minimum difference between two block addresses among all the entries is stored in the `DiffMin` register. For every piece of information maintained by the buffer, a valid bit is used to indicate if the data is valid or not. When the buffers are reset, all the valid bits are set to 0. Note that we discuss the conceptual idea in this section. For implementation, we do not need to store a complete block address in each entry (Section V-E).

The APT policy can be described in the following 4 stages:

① **Buffer Allocation:** When a cache access is issued by a `load`, its instruction address is used to look up the buffers by comparing with their `InstAddrs`. If a hit happens, its associated buffer is activated. Otherwise, an empty buffer is allocated to the `load`; if all the buffers are occupied, we use the least recently used (LRU) replacement policy to select a buffer for allocation. Fig. 4 shows an example where cache access is issued by the `load` with `InstAddr 0x8008`, and its associated buffer (Buffer[0]) is activated. When a cache access is issued by the `load` with `InstAddr 0x8018`, a buffer miss happens and Buffer[1] is selected with LRU and allocated to the `load`.

② **Entry Updating:** For the selected buffer, if the accessed `BlkAddr` is not recorded in the buffer, it is stored into a new entry of the buffer by APT. Once all the entries are full, we apply LRU to find an entry for the new `BlkAddr`.

③ **DiffMin Updating:** After the number of valid entries of a buffer surpasses a threshold, each time this buffer is activated, `DiffMin` is calculated by APT. To reduce the possible hardware complexity, we only use a small number of entries for each buffer, such as 8. `DiffMin` is used to predict the difference between each two addresses to be timed by the attacker.

④ **Data Prefetching:** After the number of valid entries in a buffer surpasses a threshold, each time this buffer is activated, candidate prefetching addresses are calculated. Assuming the block address of the current `load` is `BlkAddr′`, the candidate prefetching addresses are `BlkAddr′+DiffMin` and `BlkAddr′-DiffMin`. APT checks if they exist in the activated buffer; then one of them that is not in the activated buffer nor L1DCache will be prefetched. For example, assuming the cacheline size is 256 bytes, in Fig. 4, the colored cachelines' block addresses are recorded in the buffer entries, where the cachelines and their corresponding block addresses have the same color. When Buffer[0] is activated and the latest block address `0x1C00` is stored in the buffer, `DiffMin` is updated to `0x300`. APT now predicts the eviction cachelines are `0x1C00` + `0x300`×`k`, where `k` is an integer. The eviction cachelines that are not currently accessed are shown with red margins. According to our data prefetching policy, the cacheline with address `0x1C00-0x300` is finally prefetched (indicated by the arrow near ④), since `0x1C00+0x300` is already in the cache.

In this way, if a `load` is actively executed with several access patterns, the future target address of the `load` is predicted and prefetched by APT to mislead the attacker. We conservatively assume all the `loads` are vulnerable and apply APT to them. By increasing the number of the buffers, we can also increase the possibility of associating one buffer with an attacking `load`. Note that APT (or DST) only prefetch one cacheline for each `load` execution in order to reduce the risk of incurring performance overhead and avoid additional hardware cost.

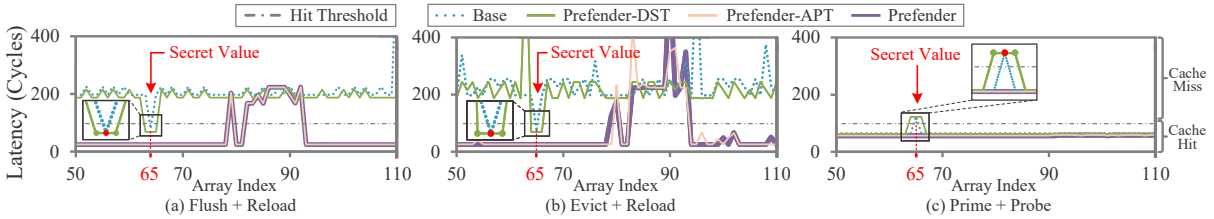

Fig. 5. The results of different attack methods. (Note that for PREFENDER-DST, the latency results of array indices 64-66 are the same in each figure.)

TABLE II
Performance improvement of SPEC CPU 2006 benchmarks. (†The basic prefetcher.)

Column ID		1	2	3	4	5	6	7	8	9	10	11
Prefetcher		PREFENDER			Tagged	PREFENDER (†Tagged)			Stride	PREFENDER (†Stride)		
# of Acc. Tra. Buf.		16	32	64	-	16	32	64	-	16	32	64
Benchmark	400.perlbench	0.677%	0.679%	1.110%	0.241%	0.427%	0.588%	0.324%	0.389%	1.117%	1.065%	1.536%
	401.bzip2	3.314%	3.346%	3.407%	4.428%	5.717%	5.728%	5.732%	1.777%	3.922%	3.959%	4.052%
	429.mcf	6.421%	8.562%	8.585%	8.636%	12.069%	12.228%	12.237%	13.233%	14.803%	17.684%	17.653%
	445.gobmk	-0.025%	-0.106%	-0.122%	1.318%	1.164%	1.103%	1.102%	0.363%	0.433%	0.379%	0.347%
	456.hmmer	0.830%	0.862%	0.891%	10.115%	10.128%	10.152%	10.158%	7.119%	6.417%	6.474%	6.512%
	458.sjeng	-0.354%	-0.355%	-0.366%	-0.437%	-0.613%	-0.615%	-0.609%	-0.016%	-0.300%	-0.303%	-0.322%
	462.libquantum	6.533%	6.533%	6.532%	4.852%	6.501%	6.501%	6.501%	7.555%	9.768%	9.770%	9.773%
	464.h264ref	0.269%	0.256%	0.408%	1.762%	1.707%	1.521%	1.804%	0.934%	0.724%	0.993%	0.793%
	471.omnetpp	-0.006%	-0.006%	-0.011%	0.112%	0.109%	0.109%	0.109%	0.229%	0.213%	0.213%	0.211%
	473.astar	0.033%	0.398%	-0.132%	0.183%	0.212%	0.415%	-0.176%	0.032%	0.059%	0.474%	-0.021%
	483.xalancbmk	0.702%	2.840%	3.941%	11.576%	11.577%	11.952%	10.592%	2.137%	2.771%	4.952%	5.683%
	999.specrand	0.000%	0.000%	0.000%	0.001%	0.001%	0.001%	0.001%	0.000%	0.000%	0.000%	0.000%
	Avg.	**1.533%**	**1.918%**	**2.020%**	**3.566%**	**4.083%**	**4.140%**	**3.981%**	**2.813%**	**3.327%**	**3.805%**	**3.851%**

V. EVALUATION

A. Experimental Setup

In our experiments, we use `gem5` simulator [16]. The baseline is configured with a 2GHz x86 out-of-order CPU with a 32KB L1Icache, a 64KB L1Dcache, and a 2MB L2cache. For security analysis, we test different Spectre attacks using Flush+Reload, Evict+Reload and Prime+Probe. For performance analysis, SPEC CPU2006 benchmark suite [17] is evaluated.

Building upon the baseline, PREFENDER can be combined with different basic prefetcher configurations, including PREFENDER only, PREFENDER with a Tagged prefetcher [18], and PREFENDER with a Stride prefetcher [19].

B. Security Evaluation

We evaluate the security effectiveness of PREFENDER by testing different side channel attacks. In phase 3, the attacker times the latencies of the cachelines by *randomly* accessing an array covering the eviction cachelines. The results are shown in Fig. 5. For Flush+Reload, the attacker successfully steals the secret value by detecting the only cache hit of the array accesses. After applying DST, two cachelines near the secret-dependent one are hit due to prefetching. After applying APT, most of the cachelines are hit because APT successfully learns the attacker's access pattern. When both DST and APT are implemented, their effect on cachelines are overlapped. Similar results are also illustrated for Evict+Reload. For Prime+Probe, the secret value is found by the attacker through the only miss of the array. Similarly, when DST is implemented, more misleading cachelines are prefetched to incur more misses. When APT is applied, all eviction cachelines are fetched and result in all hits; this also misleads the attacker. When both DST and APT are applied, only the effect of APT remains since APT performs prefetching (phase 3) after DST (phase 2). In conclusion, by successfully defeating the attacks in the threat model, PREFENDER can enforce the security the same as the previous work [8], [9], [13], [14].

C. Performance Evaluation

While enforcing security, PREFENDER can also maintain or even improve performance. The performance results of SPEC CPU 2006 benchmarks are shown in Table II, where "Basic Prefetcher" indicates the conventional prefetcher being used. The results show the improvement percentile compared with the baseline that has no prefetchers.

In Table II, the main results are at Columns 2, 6 and 10, with 32 access trace buffers. When we enable PREFENDER (Column 2), the system gains around 2% improvement on average, with the security enforcement. For Columns 6 and 10, when the conventional prefetchers are used, PREFENDER can further improve performance compared with the setups where PREFENDER is disabled (Columns 4 and 8, respectively), showing the ability of PREFENDER for maintaining performance.

PREFENDER improves overall performance in general with different impacts on each benchmark. For example, *401.bzip2*, *429.mcf* and *462.libquantum* have the most speedup with PREFENDER. In comparison, PREFENDER has almost no performance impact on *999.specrand*. For *445.gobmk*, *458.sjeng* and *471.omnetpp*, their performance has a slight drop (less than 0.5%) with PREFENDER. We further conducted a sensitivity study on the number of the access trace buffers. It shows that with the same basic prefetcher, more access trace buffers usually help improve performance. When more than 32 buffers are used, the improvements are marginal.

D. Analysis of Cache Miss and Defense

Prefetching can impact the cache miss rate. We tested the cache miss rate with PREFENDER enabled and disabled in L1Dcache. The results are shown in Fig. 6, where the shadowed bars represent the systems with PREFENDER. For each case in the benchmark, the cache miss rate results are normalized to

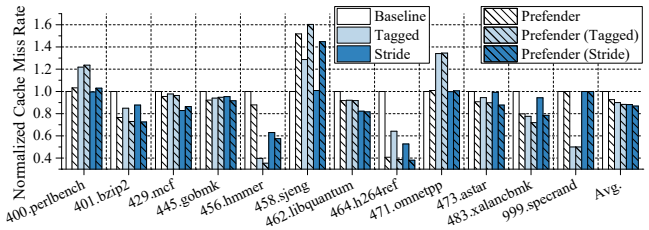

Fig. 6. The normalized cache miss rate of L1Dcache.

the baseline. In this experiment, PREFENDER is configured with 32 access trace buffers as Column 2, 6, 10 in Table II. When PREFENDER is enabled, the average cache miss rate reduces compared with that of the scenarios without PREFENDER. The cache miss rates of some cases with PREFENDER are higher than that without PREFENDER, leading to a slight performance drop for them, such as *458.sjeng*. For *400.perlbench* and *429.mcf*, PREFENDER can help improve performance with similar cache miss rates.

We further tested the number of the prefetches performed by PREFENDER, which is shown in Fig. 7. "PREFENDER-DST" and "PREFENDER-APT" represent the prefetches performed by DST and APT, respectively. It indicates that for most cases, APT prefetches more data than DST. This is because triggering DST for prefetching requires the value of a register to be calculated by $+$ and $\times$ only, and the scale is larger than the cacheline size. This happens less frequently than triggering APT, which mainly requires a `load` is frequently executed.

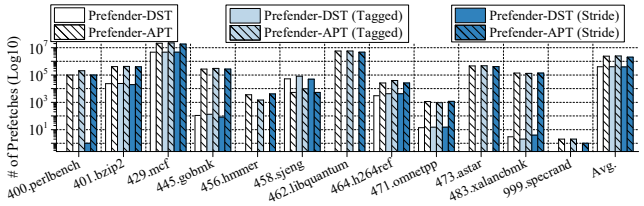

Fig. 7. The number of the prefetches performed by DST and APT under different scenarios.

E. Hardware Resource Consumption Analysis

We briefly analyze the hardware resource consumption. For SRAMs, for calculation history buffers, since the prefetching is performed within one page, 16 bits (even for a 64KB page) are enough for each value. The numbers of the buffers and the registers are the same, so it needs hundreds of bytes in total for dozens of registers. For access trace buffers, 32 buffers with 8 entries are used. Even if each entry, `InstAddr`, `DiffMin`, and the time for LRU are 64-bit, we only need <3KB SRAMs. For the datapath of PREFENDER, for DST, we need an adder, a multiplier, and a comparator, which are 16-bit. For APT, since the target is to prefetch eviction cachelines, at most 20 bits are enough for calculation even for a 1MB L1Dcache. Therefore, several 20-bit comparators and 20-bit subtractors are needed for each buffer, which is also at a reasonable level.

VI. CONCLUSION

In this paper, we propose PREFENDER, a secure prefetcher that is able to defeat cache side channel attacks while maintaining or even improving performance. In PREFENDER, we design Data Scale Tracker (DST) to predict the eviction cachelines during the victim's execution and Access Pattern Tracker (APT) to predict the access patterns during the attacker's measurement. Based on these predictions, the hardware prefetcher brings the eviction cachelines into the cache and thus obfuscates the attacker. The experiment proves the defense effectiveness of PREFENDER using Spectre attacks based on Flush+Reload, Evict+Reload, and Prime+Probe. In addition, the performance evaluation shows that PREFENDER can achieve a speedup for the SPEC CPU 2006 benchmark.

REFERENCES

[1] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptol. ePrint Arch.*, vol. 2002, no. 169, pp. 1–23, 2002.

[2] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," *USENIX Security Symposium*, pp. 719–732, 2014.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *IEEE Symposium on Security and Privacy*, pp. 1–19, 2019.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," *USENIX Security Symposium*, pp. 973–990, 2018.

[5] 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," *USENIX Security Symposium*, pp. 249–266, 2019.

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," *Cryptographers' track at the RSA conference*, pp. 1–20, 2006.

[7] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," *IEEE/ACM International Symposium on Microarchitecture*, pp. 974–987, 2018.

[8] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," *IEEE International Symposium on High Performance Computer Architecture*, pp. 264–276, 2019.

[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," *IEEE/ACM International Symposium on Microarchitecture*, pp. 428–441, 2018.

[10] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Defeating cache timing channels with hardware prefetchers," *IEEE Design & Test*, vol. 38, no. 3, pp. 7–14, 2021.

[11] H. Fang, M. Doroslovački, and G. Venkataramani, "Reuse-trap: repurposing cache reuse distance to defend against side channel leakage," *ACM/IEEE Design Automation Conference*, pp. 1–6, 2020.

[12] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," *USENIX Security Symposium*, pp. 897–912, 2015.

[13] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," *International Conference on Parallel Architectures and Compilation Techniques*, pp. 151–164, 2019.

[14] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," *ACM/IEEE Design Automation Conference*, pp. 1–6, 2019.

[15] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," *ACM/IEEE International Symposium on Computer Architecture*, pp. 347–360, 2017.

[16] The gem5 Simulator, http://www.gem5.org/Main_Page.

[17] SPEC CPU 2006 Benchmark, https://www.spec.org/cpu2006/.

[18] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[19] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," *ACM/IEEE Conference on Supercomputing*, pp. 176–186, 1991.