PREFENDER: A Prefetching Defender Against Cache Side Channel Attacks as a Pretender

Luyi Li , Jiayi Huang , *Member, IEEE*, Lang Feng , *Member, IEEE*, and Zhongfeng Wang , *Fellow, IEEE*

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 mostly trade some level of performance for security, or have restricted security scope. 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 CPU 2006 and 2017 benchmarks.

Index Terms—Security, cache side channel attacks, prefetcher.

I. INTRODUCTION

OVER the last few decades, continuing optimization of microarchitecture has led to a dramatic increase in its complexity, which might unfortunately be accompanied by many potential security vulnerabilities. As a result, the cache side channel attacks [1], [2] become serious threats to modern processors. For example, it is possible for Spectre [3] and Meltdown [4] attacks to steal almost any data in the memory, by leveraging vulnerabilities of the out-of-order execution and the speculative execution. More seriously, these two attacks can threaten most of the modern commercial processors from Intel, AMD, and ARM. Lots of variants of cache side channel attacks

have also been found in recent years [5], so the defense methods are urgently needed to enforce the security of the processors.

Cache side channel attacks exploit the cache state changes for information leakage [6]. For example, the attacker can infer the cache footprint of the victim program by the time differences between cache hits and cache misses when accessing the data [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], noise injection [10], [11], prefetching [12], [13], etc. However, these countermeasures either incur performance overhead, or have limited scope of security, such as only defending against the attacks conducted cross-core, so they failed to benefit both security and performance.

In this paper, we propose an approach to defeating the cache side channel attacks while maintaining or even improving the performance. During the attack, the attacker obtains the cache state changes made by the victim by accessing the cache. If the access patterns of both the attacker and the victim can be learned, the processor can prefetch the data that can further change the cache state to confuse the attacker. Besides, effective prefetching can help performance if the prefetcher is able to predict the access patterns of the benign programs.

We propose PREFENDER, a prefetching defender to defeat cache side channel attacks while preserving performance benefits for benign programs. Specifically, three low-cost designs are proposed, which are called Scale Tracker (ST), Access Tracker (AT), and Record Protector (RP). Scale Tracker is able to prefetch the data that the victim may access, by tracking the target address calculation history of the memory instructions. Access Tracker can learn the cache access patterns of the attackers and prefetch data for confusion, even if the attackers perform intentional random accesses. Record Protector can link Scale Tracker and Access Tracker to prevent noisy instructions and accesses from affecting PREFENDER, and further enhance the robustness of PREFENDER. Furthermore, effective prefetching of PREFENDER also maintains or improves performance. The contributions of this work are as follows:

- PREFENDER is proposed, where a novel address prediction and a noise preventing approaches for prefetching are proposed. PREFENDER can prevent wide range of general access-based cache timing side channel attacks including both single-core and cross-core attacks, while maintaining the performance.
- A new approach to analyzing cache access patterns is proposed. Scale Tacker and Access Tacker are
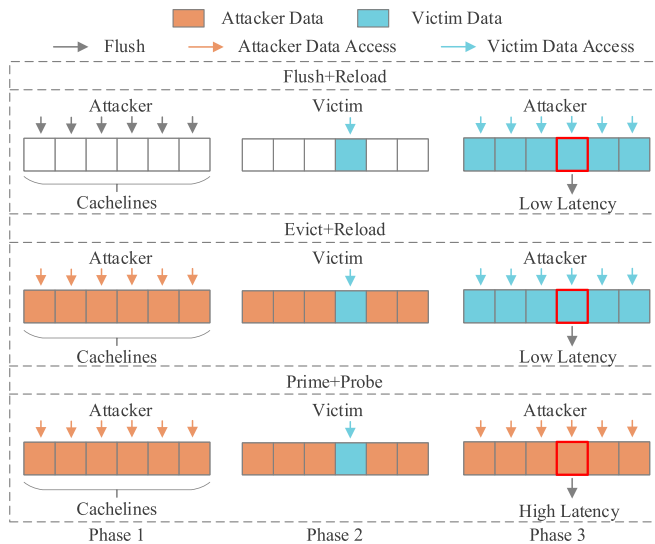
Fig. 1. The examples of Flush+Reload, Evict+Reload, and Prime+Probe. The secret can be revealed by the only low (or high) latency eviction cacheline.

designed to realize the runtime analysis for effective prefetching.

- An approach is proposed to protect PREFENDER from being affected by the noisy memory instructions and accesses. To realize this, Record Protector is designed to link the scale tracker and the access tracker to help identify the cache accesses from the attackers.
- The detailed experiments show the effectiveness and the robustness for defeating cache side channel attacks. Besides, PREFENDER also brings the performance improvement, and is highly compatible with other prefetchers.

For the following sections, Section II introduces the background and the threat model. The related work is discussed in Section III, and the details of PREFENDER are proposed in Section IV. Then the experiments are described in Section V. Finally, Section VI concludes the paper.

II. BACKGROUND AND THREAT MODEL

A. Cache Side Channel Attacks

Cache side channel attacks are to detect the cache state changes caused by the victim's memory accesses and further infer the sensitive information of the victim from these changes. In a cache side channel attack, a round of attack is typically made up of three phases. During the first phase, the attacker initializes the cache states. For example, the attacker usually uses flush instructions to invalidate the cachelines or loads irrelative data to evict the original cachelines. Then, in the second phase, the attacker does nothing but wait for the victim to be executed. During the execution, the victim accesses its data and causes changes in the cache. In the last phase, the attacker measures which cache state is different from the initialized state and therefore deduces what data the victim has accessed.

One kind of widely used cache side channel attacks is the timing-based attack, where the cache states (hit or miss) can be identified by access latencies. Fig. 1 illustrates three attacks, including Flush+Reload [2], Evict+Reload [14], and

Prime+Probe [6]. Take Flush+Reload as an example, which is based on page sharing between the attacker and the victim. In phase 1, the attacker flushes all the cachelines that may be accessed by the victim. Each cacheline is called an *eviction cacheline*, and they compose an *eviction set*. In phase 2, the victim loads the data that are related to the secrets, which is also called secret-dependent data. In phase 3, the attacker accesses the eviction set and measures the access latency of each eviction cacheline. If the attacker detects a low latency, i.e., a cache hit, the secret might be inferred from the address of this cacheline. For example, assuming the cacheline size is 64 bytes, if the victim loads a secret-dependent data `array[s×64]` in phase 2, where `s` is the secret. During phase 3, `array[768]` will be accessed with a cache hit; the attacker can infer the secret is `s=768/64=12`.

Compared with Flush+Reload, Evict+Reload mainly differs in the way of phase 1. In Evict+Reload, the attacker loads some irrelative data to evict the cachelines instead of flush instructions. In contrast, in Prime+Probe, the attacker and the victim do not share memory pages. Therefore, the attacker has its own data which maps to the same cache sets with the victim's data. In phase 1, the attacker evicts the cachelines by loading its own data. In phase 2, the victim accesses its data and evicts the attacker's data. In phase 3, the attacker re-accesses its data and detects if there is a high latency, i.e., a cache miss. This cache miss can reveal the victim's secret. The three attacks share the same key idea, which is to leverage the access latency to identify the secrets.

B. Prefetching

It is widely known that the memory wall is one of the major bottlenecks of modern processors. One approach to reducing the memory access latency is prefetching, which refers to predictively loading data into the cache in advance. If the processor requests the data later, it will encounter a cache hit and the access latency is reduced. This technique is usually implemented by the hardware module named prefetcher. Such typical examples include Tagged Prefetcher [15], Stride Prefetcher [16], Feedback Directed Prefetcher [17], Address Correlation Based Prefetcher [18], [19], etc.

C. Threat Model

We refer to work [20] to categorize the attacks. The cache timing side channel attacks that are access-based (types 2 and 4 [20]) are included in our threat model, which contains all the attacks described in Section II-A. Besides, both single-core and cross-core attacks are included. In these attacks, the attacker is able to modify the states of any cachelines (usually the eviction cachelines) and measure their access latencies. The data at the eviction cachelines are either shared or conflict between the victim and the attacker. Besides, the attacker needs and is able to access multiple eviction cachelines and leverages the timing difference between their access latencies to infer the secret of the victim[1].

---

[1] As each cacheline is not necessarily accessed multiple times in our threat model, the attack in works Prefetch-guard [10], PrODACT [21], and Reuse-trap [11] is out of the scope.

TABLE I
COMPARISONS WITH RELATED WORK IN THREAT MODEL, APPROACH, AND PERFORMANCE OVERHEAD

	Conditional Speculation [8]	NDA [22]	SpecShield [23]	InvisiSpec [9]	SafeSpec [24]	MuonTrap [25]	SpecPref [26]	Catalyst [27]	StealthMem [28]	DAWG [7]	CEASER [29]	RPcache [30]	SHARP [31]	PREFENDER
Threat Models	Speculative Execution Attacks							Cache Timing Side Channel Attacks						
Approaches	Speculation Restriction			Shadow Structures to Hold Speculative Data				Cache Partition			New Cache Replacement Policy			Prefetch
Performance Overhead	13%-54%	11%-125%	10%-73%	21%-72%	-3%	4%	1.17%	0.70%	5.90%	15%	1%	0.30%	0%	-1.69% /-6.28%

III. RELATED WORK

A. Cache Side Channel Attacks

Cache side channel attack is one of the most powerful micro-architectural side channel attacks, where the attacker can directly detect the cache states and obtain accurate timing information for inferring the secrets. Many researchers have studied various types of effective attack methods.

Kosher [32] firstly mentioned that the timing difference in the cache can be exploited to extract cryptographic secrets. Osvik [6] proposed Evict+Time and Prime+Probe methods to attack the AES algorithm [33]. In 2014, Yarom [2] proposed a more powerful and more fine-grained method, called Flush+Reload. This method utilizes the flush instruction supported by some architectures, for example, clflush in x86. Moreover, since it is based on shared memory, it has much lower noise and finer granularity, e.g., a single cacheline. The Flush+Reload has more variants, one of which is Evict+Reload [14]. The Evict+Reload is applicable to devices that do not support a flush instruction because it replaces the flush behavior with the cache eviction.

Research on cache side channel attacks continues to spring up, especially after Spectre [3] and Meltdown [4] attacks were reported. These attacks exploit one of the most important microarchitectural optimizations, speculation, to get sensitive data. They and their variants show that many critical microarchitectural components, including Branch Target Buffer (BTB) [3], Return Stack Buffer (RSB) [34], Floating Point Unit (FPU) [35], Page-table Entry [5], Intel SGX enclave [36], may inadvertently leak their internal states, including potential secrets while running. However, even if these attacks are based on different hardware components, most of them still leave the secrets in the cache and use cache side channel attacks such as Flush+Reload, Evict+Reload, Prime+Probe, etc., as mentioned before, to extract the information. Therefore, PREFENDER has a broad defense scale because it is able to defend against the cache side channel attacks that exploit the timing difference of cache access latency, including both traditional and transient execution based ones.

B. Microarchitectural Defenses

Many countermeasures have been proposed to defend against cache side channel attacks, including software and hardware approaches. Software defenses are more compatible with current platforms, but they may not fundamentally defeat the attacks, and they can incur high performance overhead. Therefore, microarchitectural defenses are further proposed in many studies.

The comparisons between PREFENDER and related work are shown in Tables I and II. Cache side channel attacks can be combined with transient speculative execution for data leakage, such as Spectre [3] attacks. To mitigate cache side channel attacks caused by transient execution, some of the prior work restricts speculation by constraining the execution of speculative loads, such as Conditional Speculation [8], NDA [22] and SpecShield [23]. They seek to identify the dangerous load instructions that can be potentially exploited by attackers and then delay their execution until all the past instructions are guaranteed to be safe. However, this method may lead to high overhead if they fail to accurately detect the dangerous loads. Another category, such as InvisiSpec [9], SafeSpec [24] and MuonTrap [25], designs a shadow structure to temporarily hold the data brought by speculative loads during transient execution, but they require many modifications to the existing hardware systems. Although SafeSpec [26] achieves a 3% performance improvement by avoiding cache pollution, its threat model is attacks on transient speculative execution, which are different from our threat model on cache timing side channel attacks. SpecPref [26] also aims at speculative execution vulnerabilities and prefetchers, but the role of the prefetchers in SpecPref is the source of the data leakage instead of the way of defense, which is a different threat model.

The above defenses only prevent data leakage caused by transient execution. They are ineffective in defending against other traditional cache side channel attacks. For the traditional ones, some new cache policies were introduced. Catalyst [27] and StealthMem [28] partition the cache into different regions for private data and shared resources, respectively. For Catalyst [27], software modifications are needed. DAWG [7] achieves a higher granularity, which dynamically partitions cache ways to avoid cache sharing among different security domains. However, these methods require programmers to rewrite the source codes to flag the sensitive data. In contrast, the key idea of CEASER [29] and RPcache [30] is to randomize the cache mapping algorithm in order to prevent the attacker from evicting the cache. SHARP [31] also designs a new cache replacement policy to prevent the eviction and flush from forcing out dedicated cachelines. It requires operating system support to handle interrupts generated by alarm counters and does not defend against single-core attacks in the private cache. Indicated in Table I, almost all the approaches for cache timing side channel attacks pay some level of performance for the security strength, or are not able to defeat general cache side channel attacks. To sum up, it is always a challenging task to design both efficient and effective defenses for both security and performance.

Besides the above studies with different approaches from PREFENDER, there are also multiple studies using prefetchers for defense, as summarized in Table II. Prefetch-guard [10], PrODACT [21] and Reuse-trap [11] propose several methods to detect the spy and leverage prefetching to obfuscate the spy based on previously recorded information, sharing the same

TABLE II
COMPARISONS WITH RELATED WORK USING PREFETCHING. (''-'' STANDS FOR THAT THE INFORMATION IS NOT MENTIONED IN THE CORRESPONDING WORK.)

			Prefetch-guard [10]	PrODACT [21]	Reuse-trap [11]	Disruptive Prefetching [12]	BITP [13]	PREFENDER
Threat Models			Access-Based Cache Attacks (Types 2 and 4 [20])					
			Flush+Reload					
		Single-Cacheline	√	√	√	-	-	×
		Multi-Cacheline	×	×	×	-	-	√
			Evict+Reload					
		Single-Cacheline	√	-	√	-	√	×
		Multi-Cacheline	×	-	×	-	√	√
			Prime+Probe					
		Single-Cacheset	√	√	√	-	√	√
		Multi-Cacheset	×	×	×	√	√	√
			Timing-Based Cache Attacks (Types 1 and 3 [20])					
		Evict+Time Cache Collision Attack	×	×	×	×	√	×
			Single/Cross-Core Attacks					
		Single-Core	√	√	√	√	×	√
		Cross-Core	√	√	√	√	√	√
Techniques		Considering Random Access Pattern	×	×	√	×	×	√
		Defense Granularity	Cacheline	Cacheline	Cacheline	Cacheset	Cacheline	Cacheline
		Handling Benign Noise Accesses	×	×	×	×	√	√
		No Software Modification	×	×	×	√	√	√
Performance & Hardware		Hardware Overhead	High. One conflict miss tracker and one flush instruction tracker per cache set.	High. One conflict miss tracker per cache set.	High. One reuse distance counter per cache set.	Low. One marked bit per cache set, randomization and set-balancer logic.	Low. BACK-INV command tracker.	Low. ST+AT+RP, detailed in Section V-E.
		Performance Improvement	-	-	-	0% (SPEC 2006)	1.10% (SPEC 2006)	1.69% (SPEC 2006) 6.28% (SPEC 2017)

idea with PREFENDER. However, their threat model is different from ours. They focus on covert channel attacks. One key feature their defenses are based on is that the attacker needs to access one cacheline multiple times. As this assumption is not included in our threat model, they cannot defeat the targeting attacks of this paper. In addition, the attacker in our threat model might access the caches randomly to mislead the prefetchers, and this is not handled by the studies [10], [11], [21]. Moreover, no techniques are proposed in these studies to handle the noise from the benign memory accesses. These studies need software modifications and can be intrusive. For hardware consumption, since they need one tracker for each cache set, the hardware overhead can be highly increased with the growth of the cache size. Besides, Reuse-trap needs to know the victim's process ID in advance to record the victim's cache misses, which may cause software modifications. Finally, they still trade some level of performance and fail in gaining performance improvement that can be achieved with prefetching. Disruptive Prefetching [12] also modifies the prefetchers to defeat cache side channel attacks. But it manipulates in a granularity of cacheset instead of cacheline, and only Prime+Probe is discussed, so the security is restricted. Meanwhile, it may cause cache pollution due to its random prefetching policy. BITP [13] prefetches the data when identifying cross-core back-invalidation-hits in multicore systems. So, it targets cross-core attacks but not single-core attacks. In contrast, PREFENDER can also be applied to single-core attacks as it is able to filter the benign memory accesses in the single-core executions. Both BITP and PREFENDER improve performance, and PREFENDER achieves higher improvement.

Compared with related work, PREFENDER is a completely hardware-based and resource-efficient method without
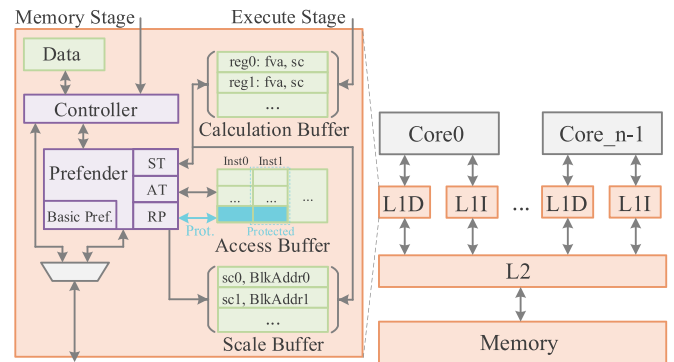

Fig. 2. The overall design architecture of our system.

modifying any policy of speculative execution or cache in modern processors. It can effectively defend against the multi-cacheline (cacheset) access-based cache attacks, as well as single-core and cross-core attacks. It also considers the random accesses from the attacks and the noise from the benign accesses, and the defense granularity is each cacheline. On the premise of ensuring security, it further achieves a performance enhancement better than prior work through accurate runtime analyses and well-designed hardware prefetching strategies.

IV. PREFENDER DESIGN

In this section, the overview of the proposed PREFENDER shown in Fig. 2 is first introduced, and the details of Scale Tracker (ST), Access Tracker (AT), and Record Protector (RP) are then elaborated.
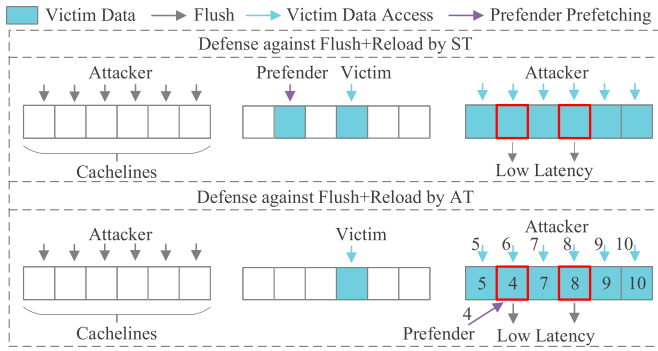
Fig. 3. The example of the defenses against Flush+Reload attacks (the number near an arrow represents the access time, and the number inside each rectangle represents the first time when the corresponding cacheline is accessed).
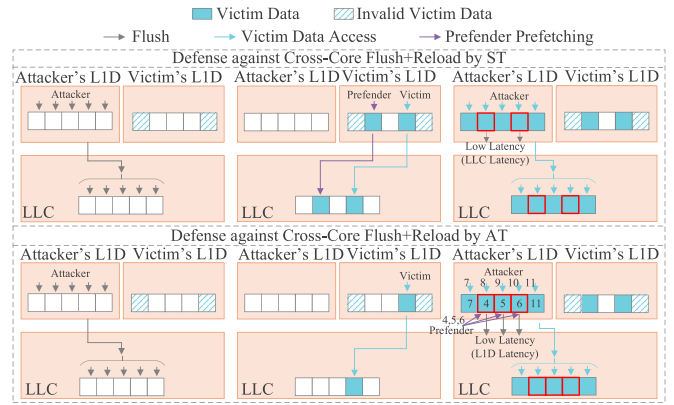

Fig. 4. The example of the defenses against cross-core Flush+Reload attacks (the number near an arrow represents the access time, and the number inside each rectangle represents the first time when the corresponding cacheline is accessed).

A. Overview

According to Section II-A, three phases need to be performed by a cache side channel attack so the attack can be defeated by interfering with one of the phases. PREFENDER is designed in each L1Dcache for interfering with the attacks by prefetching the eviction cachelines. Specifically, PREFENDER includes *Scale Tacker (ST)* and *Access Tacker (AT)* to interfere with the second and third phases, respectively. *Record Protector (RP)* can further protect PREFENDER from being interfered with by the noisy memory instructions and accesses, and enhance the robustness. A basic prefetcher (Basic Pref. in Fig. 2) is also supported, such as the Tagged or Stride prefetcher. The scale tracker, the access tracker, and the basic prefetcher are able to prefetch data, while the record protector can increase the accuracy of predicting the eviction cachelines. Note that the basic prefetcher can only help with performance, while the scale tracker, the access tracker, and the record protector can enforce security and also improve performance to some extent.

The scale tracker aims at predicting the eviction cachelines that might be accessed by the victim program during phase 2. The prediction is based on the arithmetic calculation histories of the victim instructions, which are stored in the *Calculation Buffer*. The scale tracker will predict and prefetch additional eviction cachelines after a victim instruction loads the data into an eviction cacheline in phase 2. The prefetched eviction cachelines can mislead the attacker since the attacker is unable to distinguish them from the cacheline loaded by the victim instruction. An example is shown at the top of Fig. 3.

The access tracker aims at predicting the attacker's access patterns of the eviction cachelines for measuring the access latency during phase 3. The access tracker leverages the insight that a few `load` instructions are intensively used for the attack and stores the attacker's access patterns in the *Access Buffer*. An example is shown at the bottom of Fig. 3, where the access tracker prefetches the eviction cacheline before the attacker accesses it and measures the access latency. This can also mislead the attacker.

Although the access tracker can interfere with phase 3 to mislead the attacker, since phase 3 is much longer than phase

2, there is more noise during the phase, which may affect the prediction of the access tracker. Because phase 2 is performed by the victim, the victim's access patterns learned by the scale tracker are regarded as trusted patterns, and can help correct the prediction of the access tracker. The record protector is designed to link the scale tracker and the access tracker to prevent the noise from affecting the access tracker. The record protector can record the victim's cache access prediction of the scale tracker into the *Scale Buffer*. If the attacker's access pattern in the access buffer matches a predicted victim's cache access in the scale buffer, the corresponding information in the access buffer is protected from being interfered with by the noise, and the prefetching is guided by the records in the scale buffer.

Note that ST and AT also work for cross-core attacks. An example is shown in Fig. 4. In this example, the programs of the attacker and the victim are on different cores with different L1D caches, but they share the same last level cache (LLC). For ST, after the attacker flush the eviction cachelines, when the victim accesses the data on another core, ST will prefetch the additional eviction cacheline similar as Fig. 3, both in victim's L1D cache and LLC. For phase 3, the cross-core attack originally can identify the only LLC hit to infer the sensitive information, but with ST, there are two LLC hits and the attacker is not able to distinguish which one is accessed by the victim's program. For AT, similar as the case of single-core attack, AT can directly prefetch the eviction cachelines into both attacker's L1D cache and LLC in phase 3. As the attacker keeps accessing the eviction cachelines, AT will keep prefetching, which can prefetch the cacheline in LLC accessed by the victim to L1D cache, and can directly mislead the attacker.

Since the key idea of the scale tracker, the access tracker, and the record protector is to correctly learn cache access patterns for prefetching, effective prefetching on benign loads can also improve performance while enforcing security. However, there are four major challenges for effective prefetching for PREFENDER.

C1. During phase 2, the victim may only access one eviction cacheline. Even though there are other eviction cachelines that may also be 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 the scale tracker.

C2. During phase 3, the eviction cachelines might be randomly accessed by the attacker. This can bypass some prefetchers such as Stride prefetcher. Predicting the eviction cachelines based on a random access pattern is challenging, which is tackled by the access tracker.

C3. During phase 3, there might be noisy memory instructions executed, so that the records of the attacker's access patterns in the access buffer are overwritten by the noisy instructions. In this case, the access tracker might be bypassed. We tackle this problem by using the record protector.

C4. During phase 3, if some non-eviction cachelines are also accessed by the same attacking instruction, the prefetching of the access tracker can be affected by these noisy accesses. We tackle this by using the record protector. Note that the challenge C3 is related to overwriting the attacker's recorded access behaviors, while the challenge C4 is about extra misleading behaviors.

B. Scale Tacker

The prediction of the Scale Tracker (ST) is based on the target address calculation of the victim load. For example, if the load's target address is calculated by $128 \times i+192$, where i is an integer variable, the target address can only be $192, 320, 448$, etc. After the virtual address is translated to the physical address, if paddr is the target physical address for this time, it can be deduced that paddr-128, paddr, paddr+128, etc., may also be accessed by this load if they are in the same page. In this way, we can predict the access pattern of the victim instruction in phase 2. The main goal is to learn the 128 as in the example, which is called the *scale* in our work.

The target address of a load is usually stored in the registers, so the scale tracker needs to track how the register values are calculated. This can be realized by recording all the calculation history of each register, but it can incur unacceptable hardware consumption. Therefore, only addition and multiplication (including subtraction and shifting) are considered, as they are widely used in the calculation, and their calculation history can be tracked by using only two values for each register.

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 buffers. The fva_r is needed to help track the scale, and it records the calculation result if all the calculations of register r only depend on constant values (immediate numbers). If the value of r depends on some variables such as the loaded memory values, fva_r is not applicable (NA).

The cache access pattern predicted by the scale tracker mainly depends on the scale sc_r. Usually, array access address in a loop is calculated as base+scale$\times$i (e.g., base+128$\times$i), where base is the base address and i is an integer variable. The above calculation will be propagated through some registers, and the final calculation result is stored in a register and used as the target address of a load to access

TABLE III
THE RULES TO CALCULATE fva_{rd} AND sc_{rd}. (rd IS THE DESTINATION REGISTER; "-" IS NOT APPLICABLE. †THE RULE IS ALSO FOR SUBTRACTION WHEN + IS REPLACED BY −. ‡THE RULE IS ALSO FOR SHIFTING WHEN × IS REPLACED BY >> OR <<)

Instruction	Conditions				Results	
	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

the array. One task of the scale tracker is to track the scale by propagating scales and fixed values from registers to registers during the calculations. Assuming the target address $addr$ is stored in register r, we can obtain the scale sc_r related to r. When one load is executed even for a single time, the scale tracker can predict that the nearby cachelines ($addr \pm sc_r$) may also be accessed by the same load. This is the access pattern tracked by the scale tracker.

The scale tracker can also support more complicated access patterns, such as 128$\times$i+32$\times$j+imm, where i and j are variables as the indices and imm is an immediate number. In this example, 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 an access pattern that involves multiplications of several variables (such as ($128i_0 i_1 i_2+32j_0 \times 16j_1) \times (48k_0+imm$)) can also be handled by propagating the scales and the fixed values during the calculations.

The proposed rules for calculating sc_r (and fva_r, which can help calculate sc_r) are illustrated in Table III. When a program is started, the fixed and scale values are initialized to NA and 1, respectively. During the execution of the program, the fixed value and scale of the destination register rd are calculated according to the operand and the propagated values of the source registers.

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, fva_{rd} and sc_{rd} are reinitialized since we conservatively regard the loaded value as an unknown variable.

For addition, when fva_{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
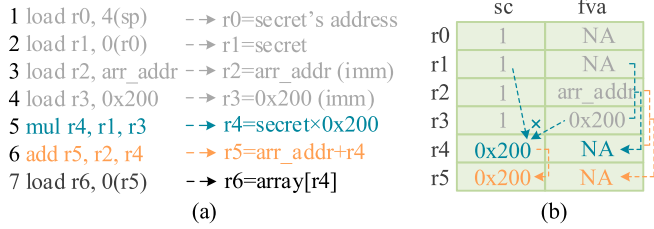
Fig. 5. (a) A pseudo code example for accessing `array[secret×0x200]`, where `arr_addr` is an immediate number that represents the address of the first element in `array`. (b) The scales (sc) and the fixed values (fva) in the access buffer, where each value is set according to the instruction with the same color and the values indicated by the arrows.

Fig. 6. An example of the access buffer.

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. 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.

When an instruction `load rd imm(rs)` or the equivalent instruction is executed, assuming the target address for this time is $addr'$, then $addr' \pm sc_{rs}$ are the candidate prefetching addresses. Once sc_{rs} is larger than the cacheline size and smaller than the page size, the candidate addresses that are not currently in the L1Dcache are prefetched. We conservatively assume that all the `load` instructions might be the victim's instructions that are vulnerable. Therefore, the scale tracker is applied to all the `load` instructions. Although all `load`s are considered, the defense is performed when the target addresses are calculated by addition and multiplication and the scales are larger than the cacheline size. This implies that the prefetching is performed when the `load`s are likely from phase 2 of the attacks instead of arbitrary `load`s, and this can mitigate the potential cache pollution. For implementation, since the scale tracker prefetches data in the same page, the bitwidth for storing and calculating fva_r and sc_r can be small (Section V-E).

An example is shown in Fig. 5. The pseudo code in Fig. 5(a) accesses `array[secret×0x200]` at line 7. For Lines 1-2, the instructions load the secret's address and the secret from the memory to r0 and r1, respectively. Therefore, the values of r0 and r1 are regarded as variables and fva of them are NA. Lines 3-4 load the immediate numbers to r2 and r3, which makes the fva of r2 and r3 be `arr_addr` and `0x200`, respectively. Next, line 5 multiplies r1 (`secret`) and r3 (`0x200`) and stores the result to r4. According to Table III, since r1's fva is NA and r3's fva is `0x200`, the sc of r4 is `0x200×1`, and fva of r4 is NA. For line 6, the r2 and r4 are added to r5, which makes sc of r4 directly propagated to r5 since r2 has a valid fva. Finally, when the `load` of line 7 is executed, the scale tracker will prefetch the data at (target address)$\pm sc_{r5}$, which are `arr_addr+secret×0x200±0x200`. In this case, assume `secret` is `12` at this time, there are at least 2 more eviction
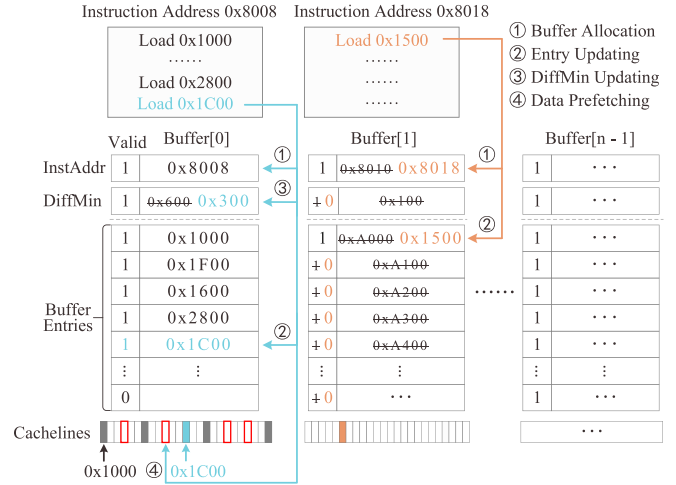
cachelines in the cache, which can mislead the attacker to get the wrong `secret` value 11 or 13.

C. Access Tracker

For phase 3, the attacker needs to time all the eviction cachelines to get the access latencies. Therefore, Access Tacker (AT) is proposed to interfere with phase 3 to further mislead the attacker. The goal of the access tracker is to learn the attacker's access pattern in phase 3, and prefetch the eviction cachelines before the attacker times them.

However, according to challenge C2, attackers may time the eviction cachelines in a random order to bypass prefetchers such as Stride prefetcher. This increases the difficulty of learning the access patterns. It is found that in common cases, the attacker's memory accesses in phase 3 are only associated with a few `load` instructions. This can help the learning of the access patterns by recording the access history of each `load` instruction separately.

For the access tracker, there is a set of access buffers, each of which is associated with a `load` instruction and records the target block addresses accessed by the associated `load`. The access buffers can help the access tracker learn the access patterns of the associated `load` instructions. For each `load`, the access pattern is estimated as a stride access—an arithmetic sequence with a constant difference, which is estimated as the minimum difference between block addresses in the associated buffer.

The microarchitecture of the access buffer is shown in Fig. 6. Each buffer maintains a register for storing the instruction address `InstAddr` of the associated `load`. For each entry of a buffer, the block address `BlkAddr` accessed by the associated `load` is recorded. There is also a register in each buffer, which stores the minimum difference `DiffMin` between two block addresses among all the entries. Each register or entry of an access buffer has a valid bit for indicating if the data is valid or not. All valid bits are set to 0 upon the reset of the buffer. 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).

Four stages are involved in the flow of the access tracker:

① **Buffer Allocation:** When a `load` accesses the cache each time, its instruction address (the value in the program counter) is compared with the `InstAddrs` to find the associated access buffer, which is then activated. If there is no associated buffer, an empty buffer is allocated to this `load`. If there is no empty nor associated buffer, one buffer is selected by the least recently used (LRU) replacement policy for allocation. For example, in Fig. 6, when the `load` with `InstAddr 0x8008` accesses the cache, associated Buffer[0] is activated. In contrast, when the `load` with `InstAddr 0x8018` accesses the cache, no buffer is associated, so Buffer[1] is selected to allocate this `load` by LRU policy.

② **Entry Updating:** In the activated buffer, if the `BlkAddr` of the accessed data is not recorded, a new entry is selected to store this `BlkAddr`. If all the entries are occupied, LRU is applied to find the entry for this `BlkAddr`.

③ **`DiffMin` Updating:** The access tracker calculates `DiffMin` of a buffer when the buffer is activated and the number of valid entries of this buffer surpasses a threshold (such as 4). The number of entries of each buffer is set to be small (such as 8) to reduce the hardware complexity. `DiffMin` can be used to estimate the difference between each two addresses to be accessed by the attacker in phase 3.

④ **Data Prefetching:** After the number of valid entries in a buffer surpasses a threshold (such as 4), each time this buffer is activated, candidate prefetching addresses are calculated. If `BlkAddr'` is the block address of the current `load`, the candidate prefetching addresses are `BlkAddr'± DiffMin`. Then, the access tracker checks if these addresses exist in the activated buffer, and prefetches one of them that is not in the activated buffer nor in the L1DCache. For example, assuming the cacheline size is 256 bytes, in Fig. 6, the colored cachelines' block addresses are recorded in the buffer entries, where the cachelines and their corresponding block addresses have the same color. When `load` with `InstAddr 0x8008` accesses the cache, Buffer[0] is activated. The latest block address `0x1C00` is stored in the buffer, and `DiffMin` is updated to `0x300` as it is calculated by $|0x1F00-0x1C00|$. At this moment, the access tracker predicts that the eviction cachelines are `0x1C00 + 0x300×k`, where k is an integer. The red margins in Fig. 6 indicate the eviction cachelines that are not currently accessed. In this case, the candidate addresses are `0x1C00±0x300`. As `0x1C00+0x300` is already in the buffer, `0x1C00-0x300` is finally prefetched by the access tracker (indicated by the arrow near ④).

In this way, the access tracker can learn the access patterns of the actively executed `load`, and prefetch the data accordingly to mislead the attacker. We conservatively assume that all the `loads` might be leveraged by the attacker, so the access tracker is applied to all of them. The possibility of associating the buffer with the attacker's `load` can be increased by increasing the number of the buffers. Note that the access tracker (or the scale tracker) only prefetch one cacheline for each `load` execution in order to reduce the risk of incurring performance overhead and

avoid additional hardware complexity. Although all `loads` are considered, the access tracker finally prefetches when a `load` is frequently executed in a time interval, which is the access pattern of the attack's phase 3. Therefore, prefetching happens when the `loads` are likely from the attacks instead of arbitrary `loads`, and the potential cache pollution is mitigated.

D. Record Protector

The access tracker can defeat the side channel attacks by prefetching the data that are predicted to be accessed by the attackers in phase 3. However, in practice, two scenarios (challenges C3 and C4) might bypass the access tracker.

- Challenge C3: In phase 3, between two eviction cacheline accesses of the attacker's `load`, there might be other benign memory access instructions executed, which are noise for the access tracker. The access buffer associated with the attacker's `load` can be occupied by a noisy instruction. According to the access tracker's policy, this noisy instruction will initialize the buffer and evict the attacker's information. In this case, the access tracker may fail to prefetch the eviction cachelines to defeat the attack.

- Challenge C4: In phase 3, if the attacker accesses the non-eviction cachelines, the access tracker will calculate wrong `DiffMin`. These accessed cachelines are also noise for the access tracker. For example, the `BlkAddrs` stored in the access buffers are `0x8000`, `0x8200`, `0x8400`, and `0x8600`, which are all the eviction cachelines. The `DiffMin` is `0x200` in this case. However, once a non-eviction cacheline with `BlkAddr=0x8100` is accessed by the same `load`, `DiffMin` will be changed to `0x100`. This can mislead the access tracker to prefetch the cachelines that are not the eviction cachelines, and the attacks can bypass the access tracker's defense.

To tackle the above two challenges, Record Protector (RP) is proposed, which can link the scale tracker and the access tracker to increase the robustness of PREFENDER, as shown in Fig 2. When a victim `load` accesses the cache, assuming register r stores the target address, the scale tracker will prefetch the data according to sc_r. Meanwhile, the record protector will store sc_r and the block address $BlkAddr_r$ of this access's target address to the scale buffer. Each time when the attacker's `load` accesses the cachelines for the timing measurement in phase 3, the block address $BlkAddr'$ of this access is checked with all the sc_i and $BlkAddr_i$ pairs in the scale buffer, where i is the index of the entry. If $(BlkAddr'-BlkAddr_i)\%sc_i=0$, it is estimated that this access is the access to the eviction cachelines. Therefore, the associated access buffer is protected so that it cannot be directly replaced by LRU, and this tackles challenge C3. Meanwhile, upon protection, the prefetching is guided by sc_i but not `DiffMin`, which can protect PREFENDER from being affected by the non-eviction cacheline records in the access buffer, and challenge C4 is tackled.

An example of the flow of the record protector is shown in Fig. 7, and the detailed policy of the record protector is elaborated as follows, where 3 stages are involved.
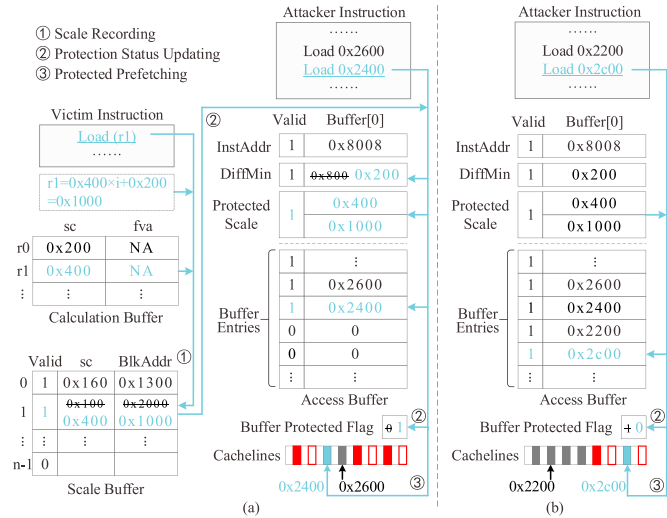
Fig. 7. An example of the flow of the record protector. (The underlined instructions access the eviction cachelines; each red margin is at the candidate address by the access tracker's policy; each red block is at the candidate address by the record protector's policy.)

① **Scale Recording:** When the victim accesses the eviction cacheline, the scale tracker uses the scale sc' in the calculation buffer for prefetching (Section IV-B). At the same time, the record protector records sc' and the block address BlkAddr' of the target address to the scale buffer, as shown in step ① of Fig. 7(a). The records in the scale buffer represent the *pattern* of the possible eviction cachelines. They can guide the access tracker to avoid being affected by noisy accesses, which is discussed in the later stages.

However, one pattern might be a subset of another pattern. If so, to reduce the redundancy, only the pattern with the larger scale is recorded. For example, in the step ① of Fig. 7(a), r1 is calculated by 0x400×i+0x200, and the target address is 0x1000 for this time. In this case, sc' =0x400 and BlkAddr' =0x1000, so the pattern is S' ={... 0x0c00, 0x1000, 0x1400, ...}. For Entry 1 of the scale buffer, sc_1 =0x100 and $BlkAddr_1$=0x2000, so the pattern is S_1 ={... 0x1f00, 0x2000, 0x2100, ...}. Since $S' \subset S_1$ (which means $sc' > sc_1$), all the possible eviction cachelines in S' are also in S_1. In this case, only S' needs to be kept for reducing redundancy, and Entry 1 is replaced by sc' and BlkAddr'. In detail, assuming the scale and the block address related to the current load are sc' and BlkAddr', when $(BlkAddr' - BlkAddr_i)\%min(sc', sc_i) = 0$ for Entry i of the scale buffer, only if $sc' > sc_i$, Entry i will be updated by sc' and BlkAddr'.

② **Protection Status Updating:** In phase 3, each time when the attacker's load accesses the cache, BlkAddr' of this load's target address is checked with all the records (sc_i and $BlkAddr_i$) in the scale buffer. If BlkAddr' matches one of the recorded patterns, which means (BlkAddr'- $BlkAddr_i$)%sc_i=0, we say BlkAddr' hits the scale buffer. When a cache access hits the scale buffer, it is estimated that the load of this access is the attacker's load in phase 3. Therefore, upon the hit, the hit sc_i and $BlkAddr_i$ are copied

to the protected scale registers in the associated access buffer, and this associated access buffer is marked as protected. With the record protector, the LRU policy in the access tracker for access buffer replacement is only applied to the unprotected access buffers. By using the scale buffer to predict which load is from the attacker, and protect its associated access buffer, the access buffer will not be replaced by noisy loads. In this way, challenge C3 is tackled.

For example, in the step ② of Fig. 7(a), load accesses address 0x2400, which corresponds to an eviction cacheline. Since it hits the scale buffer, the associated access buffer is marked as protected by setting the "Buffer Protected Flag" as 1. Scale 0x400 and block address 0x1000 are also copied to the protected scale registers.

③ **Protected Prefetching:** Besides tackling challenge C3 by protecting the access buffers, challenge C4 can be tackled by prefetching data according to the scales in the scale buffer. Each time a load's target block address BlkAddr' is stored into the access tracker, if it hits the scale buffer or the protected scale, the access tracker will use the hit scale sc_{hit} to prefetch data, i.e., the access tracker's candidate prefetching addresses are BlkAddr'$\pm sc_{hit}$. Otherwise, the candidate prefetching addresses are calculated by the access tracker's policy in Section IV-C. So, the noisy accesses of the attacker's load have much lower effects on the defense.

An example is the step ③ of Fig. 7(a). The load accesses address 0x2400. Although DiffMin in the associated buffer is 0x200, the prefetching is performed based on the hit scale 0x400 in the scale buffer. As a result, one of the candidate addresses 0x2400±0x400 not in the access buffer is prefetched. If the hit scale buffer entry is replaced later so that the BlkAddr' no longer hits the scale buffer, the associated access buffer's protected scale will be checked instead. If there is a hit like the case in Fig. 7(b), the prefetching is still performed according to the hit scale 0x400. For a protected access buffer, once the number of the prefetching using the hit scale exceeds a threshold or the buffer stays untouched for a time threshold, the access buffer is set back to unprotected status, as shown in Fig. 7(b).

In conclusion, the record protector can help the access tracker tackle challenges C3 and C4 by protecting the access buffers and performing prefetching based on the scale tracker's information, respectively. We still conservatively assume that all the loads might be the victim's and the attacker's instructions, so the record protector is applied to all of them. For implementation, since the access buffer stores the block addresses, the bitwidth for the modulus calculation can be small enough to be practical (Section V-E).

V. EVALUATION

A. Experimental Setup

In our experiments, gem5 simulator [37] is used, where the baseline configuration contains a 2GHz x86 out-of-order CPU with a 32KB L1Icache, a 64KB L1Dcache, and a 2MB L2cache. There are 4 miss-status handling registers (MSHRs), each of which can merge at most 20 requests to the same
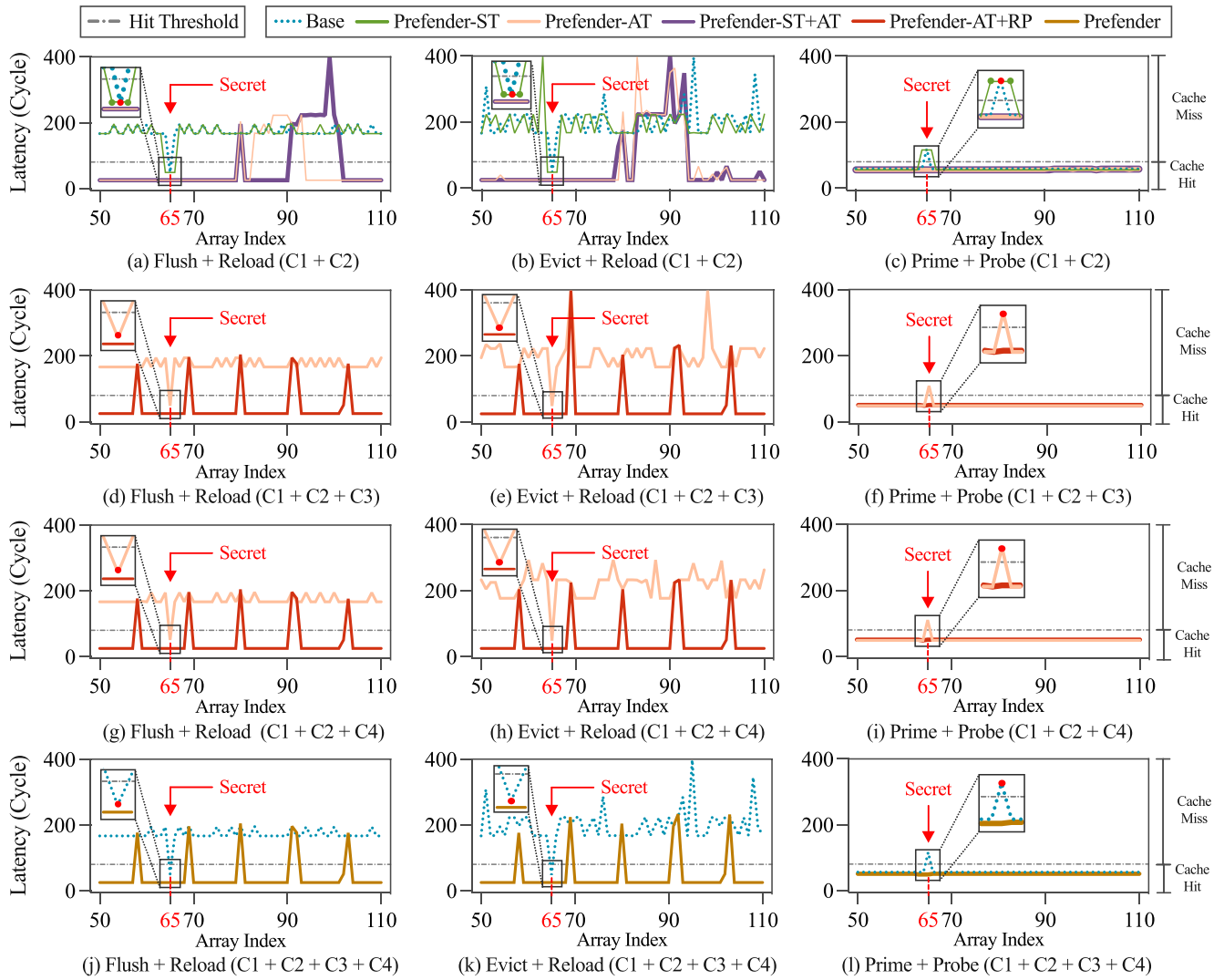
Fig. 8. The results of different attack methods with different challenges. ("PREFENDER" means that the scale tracker, the access tracker, and the record protector are all applied. Note that for PREFENDER-ST, the latency results of array indices 64-66 are the same in (a)-(c).)

line. For security analysis, we test different Spectre attacks using Flush+Reload, Evict+Reload and Prime+Probe. Challenges C1-C4 are involved based on these attacks. For performance analysis, SPEC CPU 2006 and 2017 benchmark suites [38], [39] are evaluated. Based upon the baseline, PREFENDER can include different basic prefetchers, including PREFENDER only, PREFENDER with a Tagged prefetcher [15], and with a Stride prefetcher [40]. Note that the priority of PREFENDER's prefetching is higher than basic prefetchers for timely defense.

B. Security Evaluation

Different side channel attacks are used to evaluate the security effectiveness of PREFENDER, and the results are shown in Fig. 8. We first evaluate without noisy memory instructions and noisy accesses (i.e., without challenges C3 and C4), and the results are shown in Fig. 8(a)–8(c). For Flush+Reload, without applying PREFENDER, the attacker can infer the secret value by obtaining the only cache hit when accessing the eviction

cachelines of the array in phase 3. When the Scale Tracker (ST) is applied, the scale tracker is able to introduce additional misleading cache hits on eviction cachelines, according to the calculation history. Besides, by learning the attacker's access pattern, the Access Tracker (AT) successfully predicts the accesses of the phase 3, and confuses the attacker by introducing the cache hits. When both the scale tracker and the access tracker are implemented, their effects on cachelines are overlapped. Similar results are also obtained when performing Evict+Reload attack. For Prime+Probe, the attacker infers the secret by the only cache miss. When the scale tracker is applied, more eviction cachelines are prefetched in phase 2, which incurs more cache misses. When the access tracker is applied, all eviction cachelines are prefetched so that the attacker can only obtain cache hits when accessing the array. This also misleads the attacker. When both the scale tracker and the access tracker are applied, only the effect of the access tracker remains since the access tracker prefetches (phase 3) after the scale tracker (phase 2).
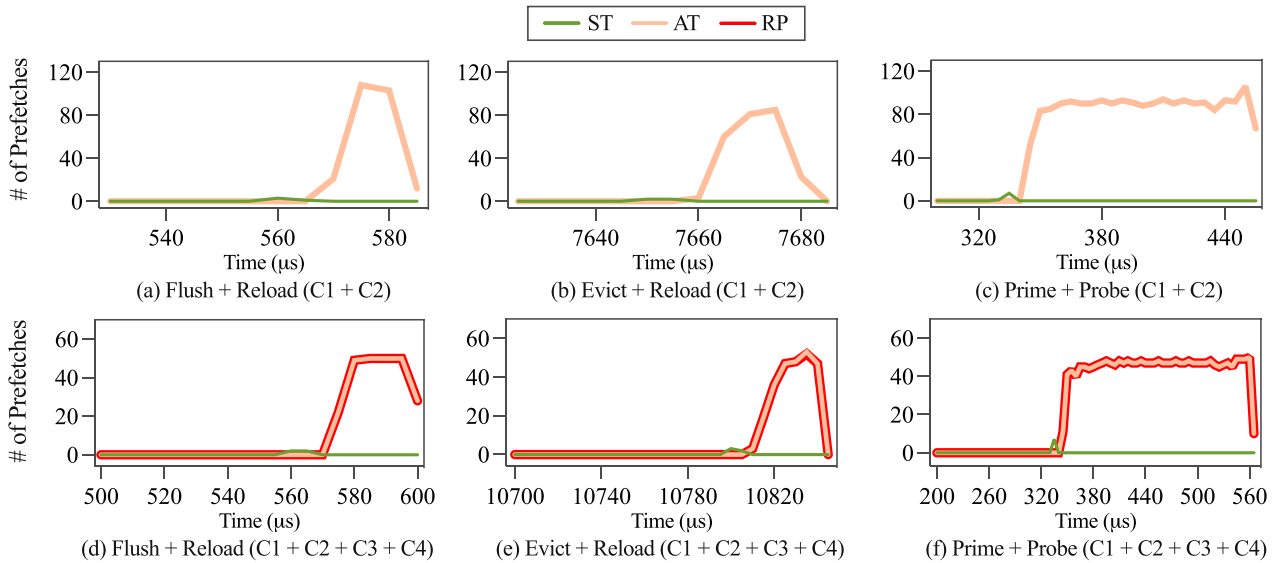
Fig. 9. The number of the prefetches performed under different attack methods with different challenges. (PREFENDER-ST+AT is applied in (a)-(c), and PREFENDER with the scale tracker, the access tracker, and the record protector is applied in (d)-(f). Note that the prefetches of the record protector refer to those of the access tracker guided by the record protector.)

When there are noisy memory instructions during phase 3 (challenge C3), the access buffers of the access tracker can be occupied by these accesses of the noisy instructions, and applying the access tracker only may not defeat the attack, as shown in Fig. 8(d)–8(f). However, when the Record Protector (RP) is implemented, the access buffer associated with the attacker's `load` is successfully identified and protected, so the access tracker is able to prefetch the eviction cachelines and mislead the attacker again. Similarly, without the record protector, when there are noisy accesses by the attacker's `load` in phase 3 (challenge C4), the value of `DiffMin` can be affected, and the access tracker may not be able to prefetch the eviction cachelines, as shown in Fig. 8(g)–8(i). In contrast, when the record protector is applied, the prefetching is guided by the scale buffer that contains the possible eviction cachelines from the victim, so the access tracker can again correctly prefetch the eviction cachelines to mislead the attacker.

Combining all the challenges and all the designs, the security can be illustrated in Fig. 8(j)–8(l). Without applying PREFENDER, the attacker can infer the secret with the only one cache hit (or miss). With PREFENDER, even though all the challenges are involved, multiple cache hits (or misses) are introduced, and the attack is defeated.

We further analyze the insights of the defense, which are shown in Fig. 9, where the x-axis represents the execution time. We only show the part where the attack is performed. For Fig. 9(a)–9(c), challenges C1 and C2 are involved, and PREFENDER-ST+AT is applied. One can notice that the scale tracker prefetches a small amount of data shown in Fig. 9(a)–9(c), which corresponds to the data at array indices 64 and 66 of the green curves in Fig. 8(a)–8(c). After this, the access tracker prefetches more data shown in Fig. 9(a)–9(c), which is also

shown by the orange curves in Fig. 8(a)–8(c). For Fig. 9(d)–9(f), all challenges are involved, and PREFENDER with all three designs is applied. It is indicated that the scale tracker still prefetches several data. After this, with the guidance of the record protector, the access tracker successfully prefetches the data even with the noisy instructions and accesses. The corresponding results are shown in Fig. 8(j)–8(l). This further shows the mechanism of the defense.

In summary, by successfully defeating the attacks in the threat model, PREFENDER can enforce the security as the same as the previous work [8], [9], [23], [24].

C. Performance Evaluation

While enforcing security, PREFENDER can also maintain or even improve performance. When the record protector is not implemented, the performance results of SPEC CPU 2006 benchmarks are shown in Table IV. The results show the improvement percentile compared with the baseline that has no prefetchers. The main results are Columns 2, 6 and 10, where 32 access buffers are implemented. When PREFENDER-ST+AT is implemented (Column 2), the performance improvement is about 2% on average, with the security enforcement. For Columns 6 and 10 where the conventional prefetchers are applied, PREFENDER-ST+AT can further improve the performance compared with Columns 4 and 8 where no PREFENDER is implemented, respectively. This shows PREFENDER's capability for maintaining or even improving the performance.

When the record protector is implemented, the performance results of SPEC CPU 2006 benchmarks are shown in Table V, where the performance distributions are similar as Table IV. With the record protector, PREFENDER also improves the performance on average, no matter if there are basic prefetchers or not. At the same time, not only is the security enforced, but

TABLE IV
PERFORMANCE IMPROVEMENT OF SPEC CPU 2006 BENCHMARKS WITHOUT THE RECORD PROTECTOR. (†THE BASIC PREFETCHER.)

Column ID	1	2	3	4	5	6	7	8	9	10	11
Prefetcher	PREFENDER-ST+AT			Tagged	PREFENDER-ST+AT (†Tagged)			Stride	PREFENDER-ST+AT (†Stride)		
# of Acc. Tra. Buf.	16	32	64	-	16	32	64	-	16	32	64
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%**

TABLE V
PERFORMANCE IMPROVEMENT OF SPEC CPU 2006 BENCHMARKS WITH THE RECORD PROTECTOR. (†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
400.perlbench	0.584%	0.562%	0.585%	0.241%	0.001%	0.524%	0.545%	0.389%	1.115%	1.118%	1.116%
401.bzip2	3.129%	3.192%	3.251%	4.428%	5.621%	5.646%	5.667%	1.777%	3.828%	3.916%	3.958%
429.mcf	4.347%	5.494%	5.497%	8.636%	9.335%	9.557%	9.540%	13.233%	12.114%	12.755%	12.755%
445.gobmk	-0.030%	-0.066%	-0.084%	1.318%	1.189%	1.171%	1.163%	0.363%	0.386%	0.347%	0.335%
456.hmmer	0.830%	0.861%	0.891%	10.115%	10.128%	10.149%	10.162%	7.119%	6.431%	6.467%	6.529%
458.sjeng	-0.411%	-0.428%	-0.422%	-0.437%	-0.649%	-0.660%	-0.687%	-0.016%	-0.324%	-0.337%	-0.373%
462.libquantum	6.516%	6.518%	6.521%	4.852%	6.502%	6.502%	6.502%	7.555%	9.781%	9.782%	9.782%
464.h264ref	0.346%	0.300%	0.346%	1.762%	1.739%	1.806%	1.800%	0.934%	0.899%	0.812%	0.843%
471.omnetpp	0.025%	0.047%	0.058%	0.112%	0.104%	0.112%	0.106%	0.229%	0.231%	0.225%	0.230%
473.astar	0.029%	0.308%	-0.139%	0.183%	0.208%	0.355%	-0.182%	0.032%	0.054%	0.385%	-0.027%
483.xalancbmk	0.860%	2.372%	3.822%	11.576%	11.533%	11.704%	10.624%	2.137%	3.123%	4.644%	5.628%
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.	**1.352%**	**1.597%**	**1.694%**	**3.566%**	**3.809%**	**3.905%**	**3.770%**	**2.813%**	**3.136%**	**3.343%**	**3.398%**

also the robustness of PREFENDER is greatly improved by the record protector.

While the performance is improved by PREFENDER on average, the impacts on different benchmarks vary. For example, *401.bzip2*, *429.mcf* and *462.libquantum* have the most speedup with PREFENDER. In contrast, there is almost no performance impact on *999.specrand*. For *445.gobmk*, *458.sjeng* and *471.omnetpp*, their performance has a slight drop with PREFENDER. The effect of the number of the access buffers is also evaluated and shown in Tables IV and V. The results indicate that more access buffers usually help the performance. Besides, if the buffers are more than 32, marginal improvements are obtained.

Besides, the results of the cases newly presented in SPEC CPU 2017 benchmarks are shown in Table VI. Similar to SPEC CPU 2006, PREFENDER also has performance improvement, both with and without the record protector. At the same time, PREFENDER can further increase the performance based on the basic prefetchers. Note that for some benchmarks such as *510.parest_r*, the performance improvement is relatively large. This is because the data prefetched by PREFENDER can greatly help reduce the cache miss rate. For example, the cache miss rate and the cache misses' access latency of *510.parest_r* in Column 2 of Table VI (in the revision letter) are 50.26% and 55.99% less than that without PREFENDER, respectively.

D. Analysis of Cache Miss and Defense

Prefetching can impact the cache miss rate and latency. We evaluated the total latency of all cache misses of each benchmark, which is shown in Fig. 10. Each result is normalized to the baseline. In Fig. 10, "PREFENDER-ST+AT" have the same configuration as Columns 2, 6, 10 in Table IV, where the record protector is not applied. "PREFENDER" has the same configuration as Columns 2, 6, 10 in Table V with the scale tracker, the access tracker, and the record protector. It is indicated that the total latencies of cache misses are reduced on average when PREFENDER is implemented. For a few cases, the latency becomes higher than the baseline, which leads to a slight performance drop, such as *458.sjeng*. Some cases have similar miss latencies before and after applying PREFENDER, but the performance is still improved, such as *400.perlbench* and *429.mcf*.

We further evaluated the number of the prefetches performed by the scale tracker, the access tracker, and the record protector of PREFENDER. The results are shown in Fig. 11. Note that the access tracker prefetches the most data, and the record protector guides the access tracker to prefetch more data than the scale tracker. The reason is that the scale tracker performs one prefetch when a target address of a `load` is calculated by addition and multiplication, and the scale is larger than the

TABLE VI
PERFORMANCE IMPROVEMENT OF SPEC CPU 2017 BENCHMARKS. (†THE BASIC PREFETCHER.)

| Column ID		1 (ST+AT)	2	3	4 (ST+AT)	5	6	7 (ST+AT)	8
Prefetcher		PREFENDER		Tagged	PREFENDER (†Tagged)		Stride	PREFENDER (†Stride)	
# of Acc. Tra. Buf.		32	32	-	32	32	-	32	32
Benchmark	507.cactuBSSN_r	0.917%	0.874%	12.256%	12.752%	12.711%	10.707%	11.672%	11.557%
	526.blender_r	0.015%	0.015%	0.356%	0.302%	0.302%	0.120%	0.133%	0.133%
	531.deepsjeng_r	-0.396%	-0.379%	-0.121%	-0.525%	-0.513%	0.000%	-0.380%	-0.369%
	538.imagick_r	5.664%	5.664%	4.240%	6.389%	6.389%	0.561%	6.292%	6.292%
	541.leela_r	-0.072%	-0.249%	0.164%	0.257%	0.120%	0.145%	0.187%	0.073%
	557.xz_r	0.243%	0.332%	4.015%	4.107%	4.104%	1.637%	1.873%	1.892%
	510.parest_r	39.738%	50.291%	44.043%	49.822%	54.617%	0.700%	35.586%	46.775%
	548.exchange2_r	0.000%	-0.006%	0.000%	0.000%	0.000%	0.011%	-0.004%	0.015%
	554.roms_r	0.000%	0.000%	30.898%	30.898%	30.898%	15.797%	15.797%	15.797%
	Avg.	**5.123%**	**6.282%**	**10.650%**	**11.556%**	**12.070%**	**3.298%**	**7.906%**	**9.129%**
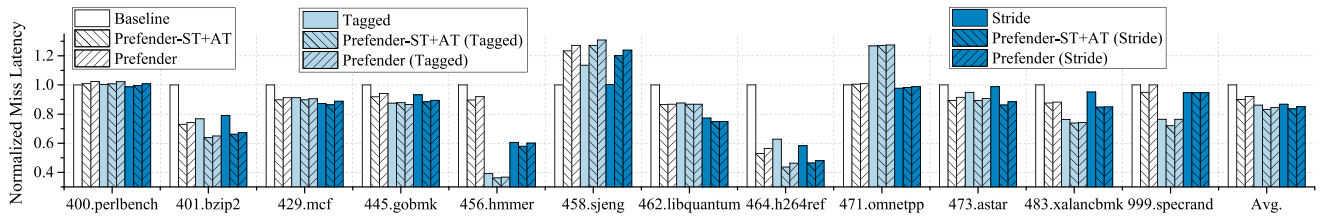
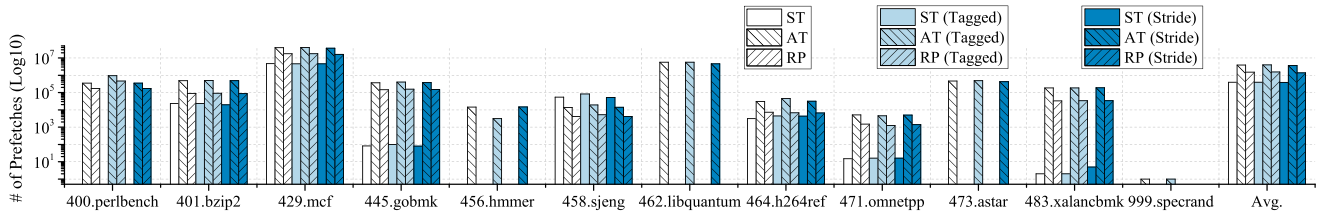Fig. 10.   The normalized total latency of all cache misses of L1Dcache.

Fig. 11.   The number of the prefetches. (The prefetches of the record protector refer to those of the access tracker guided by the record protector.)
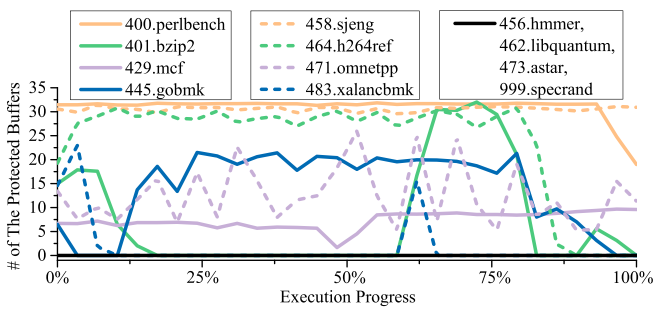

Fig. 12.   The number of the protected buffers during the execution. (The configurations are the same as that of Column 2 in Table V.)

different benchmarks have different patterns on the protected buffer numbers. For *400.perlbench*, *458.sjeng*, and *464.h264ref*, most of the buffers are protected during the execution. In contrast, *456.hmmer*, *462.libquantum*, *473.astar*, and *999.specrand* have no protected buffer. For other benchmarks, the number of the protected buffers varies. These results indicate that different functionality of the program can affect the behaviors of the record protector.

E. Hardware Resource Consumption Analysis

We briefly analyze the upper bound of the hardware resource consumption. For the SRAM size of the scale tracker, the prefetching is performed within one page, so 16 bits are enough for the values in the calculation buffers even with a page size of 64KB. For each register, there are two values associated, so the scale tracker needs hundreds of bytes in total for dozens of registers. For the datapath of the scale tracker, an adder, a multiplier and a comparator are used, which are also 16-bit.

For the SRAM size of the access tracker, there are 32 access buffers, each of which has 8 entries. Even if each value of the buffer is 64-bit, only <3KB SRAMs are required. For

cacheline size. This happens less frequently than triggering the record protector, which helps the access tracker prefetch each time a scale from the scale tracker is recorded and a `load`'s target address hits the scale history. For the access tracker, the requirement for prefetching is the easiest to be satisfied since it only needs a `load` to be frequently executed.

Finally, the number of the protected access buffers during the execution is tested in Fig. 12, which indicates that

the datapath of the access tracker, since the access tracker predicts and prefetches the eviction cachelines, 20 bits are enough for calculating the `DiffMin` even when L1Dcache is as large as 1MB. Several 20-bit comparators and 20-bit adders are used for each access buffer. The hardware consumption is also reasonable.

For the SRAM size of the record protector, the scale buffer has 8 entries in the experiments, with each entry $16(sc)+64(\text{BlkAddr})=80$ bits. For each access buffer, the record protector requires another 80-bit register for the scale history. Therefore, 400 bytes are needed. For the datapath of the record protector, a 2-way associative L1Dcache is 64KB, with each cacheline of 64 bytes, so 9 bits are used for the set index of the cache. Since the target of the prefetching is the cachelines, we only use the set index (the address of the cache entries) to calculate the modules, and several hardware modules of 9-bit modulus are needed. According to the synthesis results from Synopsys Design Compiler with ASAP 7nm library [41], the modulus only needs 2 cycles for calculation with 9-bit bandwidth, which is much quicker than memory access. Since the record protector only works upon the memory access of a `load`, the modulus calculation latency can be ignored through parallel calculation.

In summary, the hardware consumption is reasonable when PREFENDER is implemented in a modern 64-bit processor.

VI. CONCLUSION

In this work, a secure prefetcher named PREFENDER is proposed, which can defeat cache side channel attacks while maintaining or even improving performance. In PREFENDER, Scale Tracker (ST), Access Tracker (AT), and Record Protector (RP) are designed to predict the eviction cachelines according to the victim's memory access during phase 2, predict the attacker's access patterns during phase 3, and increase the robustness, respectively. The security is increased by prefetching the eviction cachelines that can confuse the attacker. Experiments on Flush+Reload, Evict+Reload, and Prime+Probe prove the effectiveness and robustness of our defense. Besides, the average performance is also increased by the accurate prediction, according to the evaluations on SPEC CPU 2006 and 2017 benchmarks.

REFERENCES

[1] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," IACR Cryptology ePrint Archive, Paper 2002/169, pp. 1–23, 2002.
[2] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proc. USENIX Secur. Symp.*, 2014, pp. 719–732.
[3] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1–19.
[4] M. Lipp et al., "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Secur. Symp.*, 2018, pp. 973–990.
[5] C. Canella et al., "A systematic evaluation of transient execution attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2019, pp. 249–266.
[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and counter-measures: The case of AEs," in *Proc. Cryptographers' Track RSA Conf.*, 2006, pp. 1–20.
[7] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution

[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," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 264–276.
[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 428–441.
[10] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovacki, and G. Venkataramani, "Defeating cache timing channels with hardware prefetchers," *IEEE Des. Test*, vol. 38, no. 3, pp. 7–14, Jun. 2021.
[11] H. Fang, M. Doroslovacki, and G. Venkataramani, "Reuse-trap: Re-purposing cache reuse distance to defend against side channel leakage," *ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
[12] A. Fuchs and R. B. Lee, "Disruptive prefetching: Impact on side-channel attacks and cache designs," *ACM Int. Syst. Storage Conf.*, 2015, pp. 1–12.
[13] B. Panda, "Fooling the sense of cross-core last-level cache eviction based attacker by prefetching common sense," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2019, pp. 138–150.
[14] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. USENIX Secur. Symp.*, 2015, pp. 897–912.
[15] A. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.
[16] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. ACM/IEEE Conf. Supercomput.*, 1991, pp. 176–186.
[17] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 63–74.
[18] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 1997, pp. 252–263.
[19] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2001, pp. 144–154.
[20] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 341–353.
[21] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "PrODACT: Prefetch-obfuscator to defend against cache timing channels," in *Proc. Int. J. Parallel Program.*, vol. 47, no. 4, pp. 571–594, 2019.
[22] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 572–586.
[23] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Spec-Shield: Shielding speculative data from microarchitectural covert channels," in *Proc. Int. Conf. Parallel Architec. Compilation Techn.*, 2019, pp. 151–164.
[24] 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," in *Proc. ACM/IEEE Des. Automat. Conf.*, 2019, pp. 1–6.
[25] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2020, pp. 132–144.
[26] T. Solanki and B. Panda, "SpecPref: High performing speculative attacks resilient hardware prefetchers," in *Proc. IEEE Int. Symp. Hardware Oriented Secur. Trust (HOST)*, 2022, pp. 57–60.
[27] F. Liu et al., "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 406–418.
[28] T. Kim, M. Peinado, and G. Mainar-Ruiz, "{STEALTHMEM}: {System-Level} protection against {cache-based} side channel attacks in the cloud," in *Proc. USENIX Secur. Symp.*, 2012, pp. 189–204.
[29] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 775–787.
[30] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2007, pp. 494–505.
[31] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based

side channel attacks," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2017, pp. 347–360.

[32] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. Int. Cryptol. Conf.*, 1996, pp. 104–113.

[33] J. Daemen and V. Rijmen, "AES Proposal: Rijndael", NIST AES Proposal, 1998, pp. 1–45.

[34] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *Proc. USENIX Workshop Offensive Technol.*, 2018, pp. 1–12.

[35] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," 2018, *arXiv:1806.07480*.

[36] J. V. Bulck et al., "Foreshadow: Extracting the keys to the Intel SGX Kingdom with transient out-of-order execution," in *Proc. USENIX Secur. Symp.*, 2018, pp. 991–1008.

[37] The gem5 simulator." gem5. Accessed: 2022. [Online]. Available: http://www.gem5.org/Main_Page

[38] SPEC CPU 2006 Benchmark." Standard Performance Evaluation Corporation (SPEC), Gainesville, VA, USA. Accessed: 2006. [Online]. Available: https://www.spec.org/cpu2006/

[39] SPEC CPU 2017 Benchmark." Standard Performance Evaluation Corporation (SPEC), Gainesville, VA, USA. Accessed: 2017. [Online]. Available: https://www.spec.org/cpu2017/

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

[41] ASAP 7nm predictive PDK." Arizona State Univ., Tempe, AZ, USA. Accessed: 2017. [Online]. Available: http://asap.asu.edu/asap/

Luyi Li received the B.S. degree in VLSI design and system integration from Nanjing University, China. He is currently working toward the Ph.D. degree with the Department of Computer Science and Engineering, University of California, San Diego. His research interests include computer architecture, hardware security, and domain-specific acceleration.

Jiayi Huang (Member, IEEE) received the B.Eng. degree in information and communication engineering from Zhejiang University, China, in 2014, and the Ph.D. degree in computer engineering from Texas A&M University, in 2020. He is currently an Assistant Professor with The Hong Kong University of Science and Technology, Guangzhou. His research interests include computer architecture, computer systems, and security. He is a member of the ACM and the IEEE Computer Society.
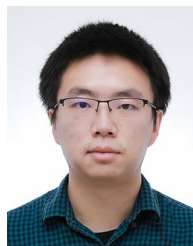
Lang Feng (Member, IEEE) received the B.E. degree in electronic science and technology (microelectronic technology) from the University of Electronic Science and Technology of China, Chengdu, China, in 2016, and the Ph.D. degree in computer engineering from Texas A&M University, College Station, in 2020. In November 2020, he joined the School of Electronic Science and Engineering of Nanjing University, where he is an Associate Research Fellow. He is working as an Assistant Professor at Sun Yat-sen University since June 2023. His research interests include computer architecture and security.

Zhongfeng Wang (Fellow, IEEE) received the B.E. and M.S. degrees from Tsinghua University, and the Ph.D. degree from the University of Minnesota, Minneapolis, MN, USA, in 2000. He has been working with Nanjing University, China, as a Distinguished Professor since 2016. Previously he worked with the Broadcom Corporation, CA, from 2007 to 2016 as a Leading VLSI Architect. Before that, he worked with Oregon State University and National Semiconductor Corporation. He is a world-recognized expert on low-power high-speed VLSI design for signal processing systems. He has published over 200 technical papers with multiple best paper awards received from the IEEE Technical Societies. In the current record, he had many papers ranking among top 25 most (annually) downloaded manuscripts in IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. In the past, he has served as an Associate Editor for IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS for many terms.