ShadowLoad: Injecting State into Hardware Prefetchers

Lorenz Hetterich
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
lorenz.hetterich@cispa.de

Fabian Thomas
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
fabian.thomas@cispa.de

Lukas Gerlach
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
lukas.gerlach@cispa.de

Ruiyi Zhang
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
ruiyi.zhang@cispa.de

Nils Bernsdorf
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
nibe00018@stud.uni-saarland.de

Eduard Ebert
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
edeb00001@stud.uni-saarland.de

Michael Schwarz
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
michael.schwarz@cispa.de

Abstract

Hardware prefetchers are an optimization in modern CPUs that predict memory accesses and preemptively load the corresponding value into the cache. Previous work showed that the internal state of hardware prefetchers can act as a side channel, leaking information across security boundaries such as processes, user and kernel space, and even trusted execution environments.

In this paper, we present ShadowLoad, a new attack primitive to bring inaccessible victim data into the cache by injecting state into the hardware prefetcher. ShadowLoad relies on the inner workings of the hardware stride prefetchers, which we automatically reverse-engineer using our tool StrideRE. We illustrate how ShadowLoad extends the attack surface of existing microarchitectural attacks such as Meltdown and software-based power analysis attacks like Collide+Power and how it can partially bypass L1TF mitigations on clouds, such as AWS. We further demonstrate FetchProbe, a stride prefetcher side-channel attack leaking offsets of memory accesses with sub-cache-line granularity, extending previous work on control-flow leakage. We demonstrate FetchProbe on the side-channel hardened Base64 implementation of WolfSSL, showing that even real-world side-channel-hardened implementations can be attacked with our new attack.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; **Hardware reverse engineering**; Cryptanalysis and other attacks; *Virtualization and security*.

Keywords: microarchitecture, prefetcher, side channel

ACM Reference Format:
Lorenz Hetterich, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, Nils Bernsdorf, Eduard Ebert, and Michael Schwarz. 2025. **ShadowLoad: Injecting State into Hardware Prefetchers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716020**

1 Introduction

Microarchitectural attacks often exploit undocumented CPU implementation details. A wide range of microarchitectural elements including caches [28, 53], predictors [20], prefetchers [10, 23], and dynamic voltage and frequency scaling (DVFS) [49], have been exploited for microarchitectural attacks. While prediction-based attacks have been used as side channels [8] and for transient-execution attacks [20], they primarily focus on branch prediction. Only recently have researchers investigated the prediction capability of hardware prefetchers [6, 33, 45, 56]. However, the focus has been on control flow [6] and data injection [45] (cf. Table 1). Hence,

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716020

several aspects of hardware prefetching are still unexplored in the context of microarchitectural attacks.

In this paper, we analyze the internal state used by stride prefetchers to answer the following questions: *What are the security implications of "poisoning" the internal state, i.e., injecting state into stride prefetchers? What additional information can be gained from the internal state of such prefetchers, also across security boundaries?*

We demonstrate ShadowLoad, a technique to bring an attacker-chosen, unaccessed memory location of the victim into the cache. In contrast to Spectre, ShadowLoad does not require complex gadgets but only a single load in the victim. ShadowLoad exploits the missing isolation of hardware prefetcher state between applications or privilege levels on Intel and AMD CPUs up to (and including) Zen 2. Our analysis uncovers previously unknown aliasing, enabling the creation of prefetcher state entries, which other contexts reuse. Hence, prefetcher state can be injected from userspace and then applied in different targets, such as the kernel, bringing architecturally inaccessible data into the L1 cache.

We additionally introduce FetchProbe, a technique to monitor offsets of memory accesses with sub-cache-line granularity across privilege boundaries without relying on shared memory. FetchProbe works on Intel and AMD CPUs up to Zen 2, and extends previous works that monitor the control flow with sub-cache-line granularity [6, 56]. The idea of FetchProbe is to trick the hardware stride prefetcher into completing a stride based on a prior access from the victim and trigger accesses from the attacker. By varying the address of the trigger accesses and observing if a stride completion happens, an attacker learns the partial address of victim data loads. Since FetchProbe relies on undocumented hardware stride prefetcher behavior, we introduce the StrideRE framework to reverse-engineer its implementation automatically.

We use ShadowLoad and FetchProbe in 5 case studies. We demonstrate that ShadowLoad can partially re-enable L1TF by returning data to the L1 cache after it is flushed by breaking KASLR of the hypervisor despite state-of-the-art mitigations. Such attacks are realistic since L1TF-affected machines are widely used in cloud environments like AWS. We combine ShadowLoad with Meltdown to leak uncached data. Our proof-of-concept implementation leaks up to 203.2 kB/s on an Intel Xeon E5-1505M v5 with 86.3 % of bytes recovered correctly. On CPUs unaffected by Meltdown or MDS, we combine ShadowLoad with Collide+Power and confirm leakage comparable to Spectre-based Collide+Power. We show how FetchProbe can re-enable cache-based attacks on the side-channel resilient Base64 implementation of WolfSSL via the data flow. Finally, we use FetchProbe with Spectre gadgets that are insufficient for cache-based Spectre attacks, leaking 18.6 kB/s on an Intel Core i9-13900K with 100 % correctly recovered bytes.

Our attacks show that hardware prefetchers are a powerful primitive for improving or re-enabling microarchitectural attacks. Currently, only turning off hardware prefetchers mitigates our attacks. Alternatively, we discuss a software mitigation similar to the ConTExT [35] Spectre mitigation.

Contributions. The main contributions of this work are:

1. We propose ShadowLoad, a novel attack primitive that allows prefetching data at rest into CPU caches by relying on previously unknown aliasing in the data structure of hardware stride prefetchers.
2. We build StrideRE, a framework to reverse engineer parameters required for hardware stride-prefetcher attacks automatically.
3. We present FetchProbe, a side channel on stride prefetchers that leaks offsets of memory accesses across security boundaries with sub-cache-line granularity, extending previous works that focus on control flow.
4. We demonstrate that ShadowLoad can partially bypass mitigations for L1TF, and FetchProbe can be used for Spectre attacks.
5. We evaluate ShadowLoad and FetchProbe by leaking data at rest and enabling attacks on the side-channel-hardened WolfSSL Base64 implementation.

Outline. The remainder of this paper is organized as follows. Section 2 covers the background required for the remainder of the paper. Section 3 introduces ShadowLoad and FetchProbe. Section 4 explains how StrideRE reverse engineers parameters of hardware stride prefetchers. Section 5 evaluates StrideRE, ShadowLoad, and FetchProbe. Section 6 demonstrates ShadowLoad and FetchProbe in different case studies. Section 7 lays out related work. Section 8 proposes possible countermeasures. Section 9 discusses our findings. Section 10 concludes.

Responsible Disclosure. We notified AMD and Intel about our findings. AMD will publish a security bulletin. Intel upstreamed patches into the Linux kernel, improving MDS mitigations against our attack.

Availability. Our code is available at https://github.com/cispa/ShadowLoad.

2 Background

2.1 Caches and Cache Attacks

Modern CPUs employ small, high-speed memory buffers known as caches. These caches bridge the latency between the slow memory and the faster CPU by keeping frequently accessed memory close to the CPU. A cache line, the smallest unit in caches, is typically 64 bytes. Modern CPUs use set-associative caches with n-way cache sets. Caches are hierarchically organized into two or three levels: a fast private L1 and L2 and a slower shared last-level cache (LLC).

Cache attacks exploit differences in access times between cache hits and misses to reveal a victim's memory activity. Flush+Reload [53] uses the `clflush` instruction to flush a cache line in shared memory and, after one execution of the victim, measures reload time to detect if the victim accessed

	CPU Vendor				Spatial Resolution			Target			
	Intel	**AMD**	**Apple**	**ARM**	**Control Flow**	**Data Flow**	**Data**	**Load Target**	**Load Instr.**	**Injection**	**Prefetcher**
FetchBench [33]	×	×	×	✓	–	64 Bytes	×	✓	✓	×	SMS
AfterImage [6]	✓	×	×	×	Byte	–	×	×	✓	×	Stride
BunnyHop [56]	✓	×	×	×	Byte	–	×	×	✓	✓	Instruction
Augury [45]	×	×	✓	×	–	–	✓	×	×	✓	Data-dependent
Our work	✓	✓	×	×	Byte	1-4 Bytes	×	✓	✓	✓	Stride

Table 1. In contrast to previous work, our attacks are the only ones that target hardware prefetchers on AMD CPUs and can infer data flow (the target of loads) with sub-cache-line accuracy.

the cache line. Prime+Probe [32] fills a cache set with known addresses (prime), runs the victim, and refills the set while measuring the time it takes (probe). If the victim accessed the cache set, one of the cached values from the prime step is evicted, leading to a slower time measurement in the probe step. These attacks have been used to leak cryptographic keys [3, 32] and monitor victim activity [12, 48].

2.2 Collide+Power

Collide+Power [21] is a software-based power-analysis attack that exploits data collisions in shared CPU components to leak data. It leverages the combined power consumption of attacker and victim data in components like the memory subsystem. Data-dependent power consumption can be exploited by an attacker who can interleave attacker-controlled data with the victim's secret data, for example, by successfully moving attacker and victim data over the memory bus. Collide+Power provides a new measurement strategy that optimizes the attacker-controlled data to maximize leakage. However, to leak arbitrary data from the kernel, Collide+Power depends on gadgets that allow an attacker to force the desired data collisions, such as speculative prefetch gadgets [19]. The complexity of mistraining and exploiting such speculative prefetch gadgets significantly reduces Collide+Power's leakage rate [21].

2.3 Prefetcher

Modern CPUs contain software and hardware prefetchers that prefetch data into the cache, reducing memory access latency. Software prefetchers operate through prefetch instructions as hints to the CPU to bring data into different cache levels [17]. Hardware prefetchers automatically preload data based on access patterns and conditions, such as predictable memory access patterns [33, 39] in the case of the stride prefetcher. Often, e.g., when executing a loop, addresses are accessed by a fixed increment or decrement called a stride. Once a stride pattern is detected, the CPU can predict this pattern and preemptively fetch memory into the cache.

2.4 Virtualization and Virtual Machines

Virtualization plays a fundamental role in cloud computing. A hypervisor partitions a physical machine into multiple virtual machines (VMs), each operating independently with its own operating system (OS) and applications. Modern hardware also incorporates virtual memory and address translation within the VM, known as nested paging [2, 17]. The VM manages its own page table to translate guest virtual addresses into guest physical addresses. Additionally, the hypervisor manages the translation from guest physical addresses to host physical addresses, ensuring that each VM's memory operations remain isolated. AMD SEV [1] adds transparent memory encryption to the VMs, guaranteeing confidentiality and integrity even in a malicious or compromised hypervisor.

2.5 Transient-Execution Attacks

Out-of-order execution runs instructions when their dependencies are met, minimizing pipeline stalls. Speculative execution predicts branch outcomes and runs instructions along the guessed path. Although these instructions are executed early and reordered, they architecturally retire in order. If a fault or misprediction occurs, the effects of the instructions (transient instructions) are discarded.

Transient-execution attacks exploit that transient instructions leave side effects in the microarchitectural state to leak confidential data via side channels [11, 31, 53, 55]. Numerous transient-execution attacks have been demonstrated, commonly categorized into Meltdown-type and Spectre-type attacks [5]. Meltdown exploits lazy error handling in the CPU, leaking inaccessible data in the L1 cache. MDS attacks, e.g., ZombieLoad [36], are Meltdown variants that leak in-flight data from internal CPU buffers. Foreshadow or L1 Terminal Fault (L1TF) [42] is a Meltdown variant targeting non-present mappings. While the original Foreshadow attack targeted Intel SGX, the underlying vulnerability is also exploitable against the OS and from within virtual machines, also called Foreshadow-NG [50]. Spectre-type attacks exploit control or data flow misspeculation to leak data.

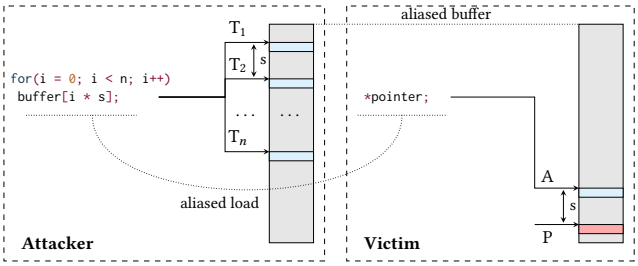


Figure 1. Overview of ShadowLoad. The stride prefetcher is mistrained in the attacker context by memory loads (T_1 to T_n) following a stride (s). Then, a single memory load in the victim context (A) leads to the prefetching of data at rest (P) following the learned stride.

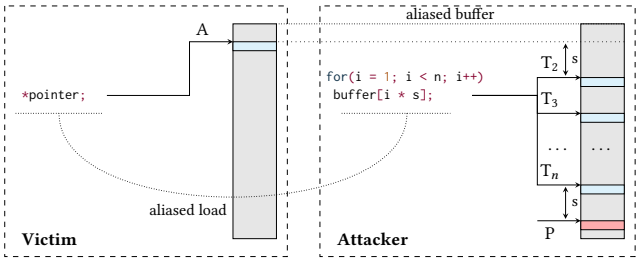


Figure 2. Overview of FetchProbe. The prefetcher is trained by a single access (A) in the victim context followed by multiple accesses (T_2 to T_n) in the attacker context following a stride (s). Only if the victim makes the access (A) that matches the stride is the prefetcher fully mistrained and prefetches the next location in the stride (P) triggered by the last access in the attacker context (T_n). By measuring the cache state of P, an attacker learns whether A was accessed.

3 Hardware-Prefetcher Attack Primitives

In this section, we introduce two new primitives, ShadowLoad and FetchProbe. ShadowLoad uses hardware-prefetcher state injection to bring inaccessible data at rest into the data cache, re-enabling attacks where the mitigation relies on cache flushing on context switch. FetchProbe in Section 3.2 turns ShadowLoad around, exploiting it as a side channel to detect whether a victim accessed an attacker-chosen address with sub-cache-line granularity. While we focus on the hardware *stride* prefetcher, we only refer to it as hardware prefetcher in the remainder of this paper for the sake of readability.

3.1 ShadowLoad

ShadowLoad exploits that internal hardware prefetcher state is usually not isolated between security contexts. Hence, the state learned in one process can influence the prefetching behavior of a different process. Thus, an attacker can trick the hardware prefetcher into continuing a sequence of memory loads inside the victim domain, bringing otherwise inaccessible data into the cache. Additionally, aliasing in the prefetcher data structures allows mistraining on different addresses; shared code or memory is not required.

ShadowLoad is illustrated in Figure 1. First, the (stride) prefetcher is mistrained by executing memory loads following a stride pattern in the attacker context, loading attacker-controlled memory (T_1 to T_n). The load instruction address, number of accesses, and base address of the memory buffer depend on the prefetcher implementation. They are chosen to alias with the victim load, so the prefetcher incorrectly assumes that they are in the same stride as the victim load. Section 4 shows how to reverse-engineer these values for a given microarchitecture automatically.

Once the prefetcher is mistrained, a memory load in the victim context is triggered (A). How this load is triggered depends on the victim context. For a kernel victim, a syscall can trigger the load. For a hypervisor victim, an instruction trapped by the hypervisor may be executed by the VM. This load additionally prefetches data following the trained stride (s) into the cache even though it is never architecturally accessed (P). The attacker chooses the trained stride (s) to specify the prefetched address. Since the victim triggers the prefetcher, the prefetching is done within the context of the victim. The state learned in userspace can influence kernel prefetching behavior on most tested devices. Thus, prefetching kernel memory is possible even if the prefetcher is mistrained in userspace if the single access triggering the prefetching originates in the kernel. We demonstrate how to use ShadowLoad for attacks in Section 6.

3.2 FetchProbe

FetchProbe is the inverse of ShadowLoad, leaking partial addresses of victim loads by observing prefetching in the attacker domain (cf. Figure 2). An attacker wants to learn whether the victim accesses address A. Assuming A is accessed, the attacker completes a stride using the memory accesses T_2 to T_n, where n is the number of accesses needed to train the prefetcher. If the victim accesses A aliasing with T_1, the prefetcher continues to load T_{n+1} in the attacker domain. Otherwise, the prefetcher is only partially trained and does not bring T_{n+1} into the cache. Thus, an attacker learns whether the victim accessed A by checking the cache state of T_{n+1}, e.g., using Flush+Reload. As the prefetching occurs on the attacker-mapped memory buffer and not the victim buffer, it can be observed even without shared memory.

Assuming we can trigger the victim access repeatedly, we can test all possible combinations of load address offset and load instruction address offset and use this oracle to gain information about the accessed memory address and load instruction address, i.e., the *data flow*. Further, many CPUs learn strides with sub-cache-line accuracy (as has been

shown for *control flow* in previous work [6, 56]), and prefetching only occurs if all training accesses follow the stride accurately. This allows leaking memory-access offsets with sub-cache-line granularity, which is especially interesting as traditional cache side-channel attacks are limited to cache-line granularity. We demonstrate how to use FetchProbe for attacks in Section 6.

4 Reverse-engineering Prefetchers

This section introduces StrideRE, our tool to reverse-engineer features of stride prefetchers automatically. The reverse-engineered parameters are vital for mounting ShadowLoad and FetchProbe. We open-source StrideRE with the final version of the paper.

4.1 Overview

The prefetcher data structures are indexed using an undocumented combination of partial address bits of the load *instruction* and load *target address*. While multiple load addresses and memory buffers map to the same index in the prefetcher, it is undocumented which address bits are used and how they are combined to form the index. It is further unknown whether some form of address-space identifier is used to prevent cross-address-space usage and whether prefetchers are shared across hyperthreads and security domains. StrideRE determines the aliasing of load-instruction and load-target addresses, and additional parameters impacting ShadowLoad and FetchProbe. The parameters StrideRE can infer and implementation details are outlined in the following sections.

4.2 Aliasing

Mounting ShadowLoad and FetchProbe requires the capability to choose aliasing addresses such that attacker and victim use the same prefetcher data structures. Indexing in the prefetcher data structure is based on partial addresses of the load instruction and the load target. By flipping a single bit in the load instruction address or accessed buffer address and observing whether the prefetcher is still triggered, StrideRE finds whether this bit is used for indexing the data structure. As this cannot find all forms of aliasing (i.e., if a combination of all bits is used for indexing), a second test that checks random addresses is executed in case no aliasing is found. StrideRE combines both aliasing tests to evaluate whether mistraining on different addresses for instruction and target is possible. Tests are also executed with the attacker and victim residing in different domains, i.e., the same-process, userspace and kernel, and different hyperthreads. With this, StrideRE determines whether the prefetcher data structures are shared across security domains.

4.3 Prefetcher Characteristics

In addition to collisions in the prefetcher data structures, ShadowLoad and FetchProbe require exact knowledge when the prefetcher is triggered. These characteristics include the maximum stride that can be learned, whether prefetching can cross page boundaries (i.e., the target of the triggering load and the prefetched memory reside on different pages), and the stride granularity. Additionally, StrideRE varies other parameters, like whether fence instructions are used between loads and different workloads, to determine the optimal conditions to trigger prefetching.

4.4 Implementation

StrideRE is built modularly, allowing the addition of tests (e.g., whether prefetching crosses page boundaries) and targets (e.g., kernel, userspace). As the tests require tight control over memory and executed instructions, they are written in C, while the analysis scripts are implemented in Python. An analysis script executes a test binary with different runtime and compile-time parameters and analyzes the results. Depending on the analysis script, a test might be repeated with different timers or targets.

Measurement. To detect if a memory location has been cached, StrideRE relies on Flush+Reload [53]. For this, StrideRE requires a high-resolution timer. StrideRE currently supports native timers and a generic counting-thread implementation [14, 24].

Targets. StrideRE implements a target for same-process, hyperthread, and kernel. The same-process target allows additional configuration with compile-time parameters. For instance, the virtual address of the buffer and memory-load-gadget instruction can be controlled.

Modularity. Tests can rely on an interface to interact with timing primitives and victims. This allows the implementation of tests that are agnostic to the victim. For example, the test for full aliased mistraining targets a victim in the same process, hyperthread, or kernel without modifying the test itself.

5 Evaluation

This section evaluates StrideRE by reverse-engineering the prefetcher details on 16 CPUs (Section 5.1). The results are used to evaluate ShadowLoad on 13 CPUs that StrideRE finds exploitable (Section 5.2). Additionally, we manually analyze the prefetchers of the Apple M1 and M2 CPUs in Section 5.3. We implement two variants of FetchProbe: One that infers whether a kernel memory access is executed (Section 5.4), and one that leaks the offset of a kernel memory load (Section 5.5). For both variants of FetchProbe, we provide versions for Intel and AMD CPUs.

5.1 StrideRE Evaluation

To evaluate StrideRE, we run it on 16 different CPUs. While all tested CPUs implement a hardware stride prefetcher, the details regarding aliasing and characteristics vary, as summarized in Table 2.

Aliasing. On Intel Coffee Lake, Skylake, Broadwell, and Sandy Bridge CPUs, the prefetcher data structure uses only the least significant 8 bits of the instruction address for indexing. This is increased to 10 bits on Raptor Lake (P-core), Alder Lake (P-core), and Ice Lake, while the E-cores of Raptor Lake and Alder Lake require at least 12 matching bits. Thus, finding aliasing loads on those CPUs only requires the knowledge of the least significant 8, 10, or 12 address bits. AMD Zen 1 and Zen 2 CPUs show similar aliasing if the least significant 12 bits of the load instruction match. For AMD Zen 3 CPUs, we do not observe aliasing.

$$
\begin{aligned}
A_0 &= b_0 \oplus b_8 & A_4 &= b_4 \oplus b_{11} & A_7 &= b_7 \oplus b_{11} \\
A_1 &= b_1 \oplus b_8 \oplus b_{11} & A_5 &= b_5 \oplus b_8 & A_8 &= b_{10} \\
A_2 &= b_2 & A_6 &= b_6 & A_9 &= b_9 \oplus b_{11} \\
A_3 &= b_3 \oplus b_{11}
\end{aligned}
$$

Figure 3. PC Indexing Function for Zen and Zen 2. b_i denotes the i^{th} address bit. Two addresses alias if A_0 - A_9 match.

We further manually analyzed the exact indexing functions on a subset of CPUs. On Intel Coffee Lake, Skylake, Broadwell, and Sandy Bridge CPUs, an identity function is used to map 8 input bits to 256 equivalence classes. On Zen 1 and Zen 2 CPUs, the 12 input bits are mapped to only 1024 classes using the function in Figure 3. The prefetcher cannot distinguish two load instructions with addresses mapping to the same class. The actual number of strides the prefetcher can simultaneously track might be smaller than the number of aliasing classes.

On Intel CPUs, aliasing between load targets is nuanced: On the third access, the prefetcher is still learning a stride. Aliasing and prefetching only occur if enough bits of the load targets match. For Raptor Lake (P-core), Alder Lake (P-core), and Ice Lake CPUs, 19 bits of the load targets (bits 0-19 excluding bit 16) are used for indexing. On Coffee Lake, Skylake, Broadwell, and Sandy Bridge CPUs, 14 bits are used (bits 0-14 excluding bit 12). With four or more accesses, a stride is already learned. Prefetching on Intel CPUs is triggered if bits 13 and 14 of the accessed address match (Comet Lake, Coffee Lake, Skylake, Broadwell, Sandy Bridge) or without any restrictions on the load target (Raptor Lake P-Core, Alder Lake P-Core, Ice Lake). Two notable exceptions are Raptor Lake and Alder Lake E-cores, where we do not observe any aliasing regardless of the number of accesses and amount of matching bits. Prefetchers on AMD Zen 1 and Zen 2 CPUs always use 19 address bits (bits 0-18) to decide whether prefetching should occur. AMD Zen 3 CPUs do not show any aliasing between load targets.

Characteristics. The number of accesses required for prefetching is 4 on all AMD CPUs. However, for strides larger than 4096 B, 3 accesses suffice to trigger the hardware prefetcher on Zen 1 and Zen 2 CPUs. Even if the same load instruction makes more accesses, the prefetcher is only triggered if the final access follows the stride. On Intel CPUs, 3 accesses suffice if the last access follows the stride. Starting with 4 accesses, prefetching takes place even if the last access does not follow the training stride.

The maximum stride the prefetcher can learn varies vastly across CPUs. The maximum stride on Zen CPUs is 8192, while on an Intel Xeon E5-1505M v5, strides are limited to 1984 bytes. More recent CPUs like an Intel Core i9-13900K and AMD Ryzen 9 5900HX can learn strides greater than 16 384 bytes. We find that strides on all tested AMD CPUs and Intel CPUs since Ice Lake and Alder Lake can cross pages. Only prefetchers on Intel Coffee Lake and before are limited to the same page. The stride granularity of all tested Intel CPUs is single bytes and 4 bytes for AMD CPUs.

5.2 ShadowLoad Evaluation

From the results of StrideRE, we can infer CPUs that are vulnerable to ShadowLoad. To reasonably utilize ShadowLoad for attacks, full out-of-place mistraining must be possible across privilege boundaries, i.e., StrideRE detects aliasing for loads, memory, and kernel. We inject prefetches from a userspace attacker into a kernel victim on all CPUs fulfilling those requirements.

Victim. As the victim, we use a kernel module exposing a memory load to a fixed location relative to a buffer via `ioctl`. Additionally, the victim exposes functionality to flush its buffer or only a specific cache line in the buffer from the cache. Finally, the victim allows measuring whether a specified location in the buffer is cached. We assume that the memory load instruction and buffer address are known to the attacker.

Attacker. The attacker is a userspace program. This program maps a load instruction and memory buffer such that the least significant 46 address bits match the kernel counterpart. We matched 46 bits of both, the load instruction addresses and the buffer addresses, as it leads to aliasing for all tested CPUs. The attacker uses 2 training accesses for Intel CPUs and 3 training accesses for AMD CPUs. In both cases, these loads follow a stride. Further, these loads are aligned such that the kernel load aliases the predicted next load in the stride. Next, the attacker triggers the kernel load using the `ioctl` interface. If ShadowLoad is successful, the next kernel buffer address following the trained stride is prefetched into the cache. For all affected Intel CPUs, StrideRE reports more lenient alignment requirements for the accessed buffer if 4 or more accesses are made. We perform an additional test on those CPUs using 3 training accesses but a non-aligned kernel load target.

Results. We run the attacker 100 000 times and record the number of successful injections and the runtime of the mistraining and execution of the victim gadget. The whole test is run once with the final victim accesses on memory not residing in the cache and once with cached memory.

Table 2. Results of StrideRE clustered per similar prefetcher. `Loads` and `Memory` denote bitmasks of address bits used in the prefetcher data structure.

	Aliasing			Characteristics			
Microarchitecture	*Loads*	*Memory*	*Kernel*	**Accesses**	**Stride**	**Cross Page**	**Granularity**
Zen 1	`0xFFF`	`0xFFFF0`	✓	3 (stride > 4096)/4	8192	✓	4
Zen 2	`0xFFF`	`0xFFFF0`	✓	3 (stride > 4096)/4	8192	✓	4
Zen 3	✗	✗	N/A	4	≥ 16384	✓	4
Ice Lake, Raptor Lake (P-Core), Alder Lake (P-Core)	`0x3FF`	`0xEFFFF/0x0`	✓	3/4	≥ 16384	✓	1
Raptor Lake (E-Core), Alder Lake (E-Core)	`0xFFF`	✗	N/A	3	≥ 16384	✓	?
Skylake, Coffee Lake, Comet Lake, Sandy Bridge, Broadwell	`0xFF`	`0x6FFF/0x6000`	✓	3/4	1984	✗	1

ShadowLoad can inject prefetches across all tested CPUs with varying success rates. AMD Zen 1 and Zen 2 CPUs (AMD Ryzen 5 2500U, AMD Ryzen 5 3550H, AMD Ryzen 7 5700U, and AMD EPYC 7252) have a high success rate of 67.5 % to 99.9 %. On most tested Intel CPUs (Intel Core i3-1005G1, Intel Core i9-12900K, Intel Xeon E5-1505M v5, Intel Core i5-6400T, Intel Xeon E3-2176M, Intel Core i7-10510U, Intel Core i5-2520M, and Intel Core i5-5010U), ShadowLoad succeeds with 46.8 % to 97.9 % of injections. One notable outlier is the Intel Core i9-13900K (P-Cores), with a success rate of only 11.0 % for unaligned injections using cached accesses while mistraining. Even aligned injections training on uncached memory are only successful in 23.0 % of cases on this CPU. The overhead of mistraining is just 64.7 ns to 3489.1 ns per cache line, and the overhead of executing the victim gadget is 85.5 ns to 2264.7 ns. The one-time setup overhead of ShadowLoad is small at 6.2 µs to 67.7 µs. We only observe prefetching on AMD CPUs if the kernel access targets uncached memory that aliases the next address in a trained stride. Intel CPUs show prefetching independent of the cache state and alignment of the final access.

5.3 Prefetchers of the Apple M1 and M2 CPUs

In our tests, the hardware stride prefetchers of the Apple M1 and M2 CPUs do not show aliasing for load targets. Still, we manually reverse-engineer the prefetcher of the Apple M1 and M2 P-Cores (called Firestorm and Avalanche, respectively). While the prefetching logic on both cores does not consider the load instruction address, the full load target address is relevant (i.e., we cannot detect any load-target aliasing). It might still be possible that some unknown non-linear function is used to compress the load-target address before it is used as an index.

We further analyze how many strides these prefetchers can track simultaneously. For this, we train `N` strides, continue one of the trained strides, and measure whether the prefetcher is triggered. By varying `N`, we determine the maximum amount of strides the prefetcher can simultaneously remember. Firestorm and Avalanche can keep track of 16 strides. If more than 16 strides are trained, continuing any of the previous strides no longer causes any prefetching. This indicates that the prefetcher data structure is fully-associative with 16 ways. Further experiments confirm that the eviction policy for new strides is least-recently used (LRU).

5.4 Monitoring Memory Accesses with FetchProbe

In this section, we evaluate FetchProbe as primitive to detect if a victim loads data from a monitored memory location.

Victim. Our victim contains a conditional branch based on a secret bit, a classical scenario for side-channel attacks. The taken branch contains a memory access to a fixed location, whereas the not-taken branch has no memory access.

Attacker. The attacker triggers the execution of the victim gadget, which, depending on the secret, accesses a known memory location. Following gadget execution, the attacker continues training the prefetcher in their context and observes whether prefetching occurs, as illustrated in Figure 2. The attacker can infer the secret value depending on whether prefetching takes place.

Evaluation. We run our proof of concept on 3 Intel (Intel Core i3-1005G1, Intel Core i9-12900K, Intel Core i9-13900K) and 4 AMD (AMD Ryzen 5 2500U, AMD Ryzen 5 3550H, AMD Ryzen 7 5700U, AMD EPYC 7252) CPUs. These CPUs are all affected by ShadowLoad and allow prefetching to cross page boundaries. We leak 32 768 bits (i.e., whether an access was made), run the code 1000 times for each CPU, and report the average leakage rate, false positives, and false negatives. Correctly leaking whether the memory access was executed works with a balanced F-score of 94.0 % to 99.5 % (precision: 88.6 % to 99.4 %, recall: 92.1 % to 100.0 %) at rates of 623.3 kbit/s to 2957.8 kbit/s.

5.5 Monitoring Memory Offset with FetchProbe

In this section, we evaluate the second aspect of FetchProbe, leaking intra-cache-line offsets of memory loads.

Victim. As a victim, we use a kernel module with secret-dependent access within a cache line. For the evaluation, this access can be triggered from userspace via `ioctl`. We

assume that both the buffer address and the address of the load instruction are known to the attacker.

Attacker. The userspace attacker leaks the secret offset from the victim kernel module. For all possible offsets, the attacker uses FetchProbe to infer whether an access is made to the buffer at this offset. For this, the victim gadget must be invoked once for each possible secret offset, i.e., 64 times for a byte-granular offset in a cache line.

Evaluation. For the evaluation, we use two possible offsets. Intel CPUs can learn strides with byte accuracy. Thus, we use offsets 0 and 1. AMD CPUs can only detect offsets in multiples of 4 bytes. Hence, we use offsets 0 and 4. Again, we run our proof of concept on all CPUs reported to be affected by ShadowLoad that allow strides crossing page boundaries. We leak 32 768 bits (i.e., offsets), run the proof of concept 1000 times for each CPU, and report the average leakage rate, false positives, and false negatives. Leaking which offset was accessed works with a balanced F-score of 96.3 % to 100.0 % (precision: 97.4 % to 100.0 %, recall: 93.6 % to 99.9 %) for each offset at rates of 315.3 kbit/s to 1511.1 kbit/s.

6 Case Studies

In this section, we evaluate ShadowLoad and FetchProbe in 5 case studies. Section 6.1 demonstrates how ShadowLoad can partially bypass L1 data cache flushing as L1TF mitigation. Section 6.2 evaluates ShadowLoad with MDS. Section 6.3 mounts Collide+Power using ShadowLoad to leak data at rest. Section 6.4 shows how FetchProbe can target code hardened against cache attacks. Section 6.5 evaluates FetchProbe as a Spectre covert channel.

6.1 Re-enabling L1TF using ShadowLoad

In this case study, we use ShadowLoad and L1TF to leak a kernel pointer from the hypervisor, breaking KASLR. To mitigate L1TF attacks from VMs targeting the hypervisor, flushing the L1 cache on VM-Entry was added to Linux as a mitigation [29, 50]. However, flushing the L1 cache is not the last operation before re-entering the VM. After the cache is flushed, memory accesses that load register contents of the guest are performed. These loads can serve as gadgets for ShadowLoad, bringing additional data into the L1 cache, which can be leaked using L1TF.

Setup. We use QEMU system to run a Linux 6.8 VM. Within this virtual machine, we run a modified version of a public L1TF proof of concept exploit [46]. We use ShadowLoad to bring a (secret) hypervisor kernel pointer into the cache. The load gadget required by ShadowLoad to cause prefetching in the hypervisor is triggered using a `cpuid` instruction after mistraining the hardware prefetcher. The hypervisor traps the `cpuid` instruction, forcing a VM re-entry. This flushes the L1 cache and afterwards triggers the load gadget, prefetching the target kernel pointer of the hypervisor into the cache. As the hypervisor kernel pointer is cached, it can be leaked using L1TF.

Evaluation. We execute this exploit on an *Intel Core i7-6700* (stepping 3, microcode `0xf0`) running *Linux 6.8.0* with *mitigations=on* kernel parameter. We use *QEMU 8.1.5* with KVM as hypervisor and VMX for virtualization. As a gadget to trigger prefetching of secret data, we target a load instruction that restores the guest *rcx* register. The structure containing the registers also contains pointers to other data. We leak one such pointer located 120 B before the saved *rcx* register in memory. Thus, we mistrain the prefetcher to learn a stride of −120 B. The exploit successfully leaks the kernel pointer in less than 1 s (excluding VM boot).

We also test ShadowLoad and L1TF on the AWS cloud to verify that such an attack is possible in the cloud, underscoring the practical relevance. AWS and Google Cloud both have many Sandy Bridge to Skylake CPUs affected by L1TF. In fact, the default instance type AWS recommends when creating a new EC2 instance (T2) is vulnerable to Meltdown and L1TF. Many instances AWS considers current generation (e.g., T3, C5, F1, G3, I3) are also affected. The default GPU instance on Google Cloud is also a vulnerable Skylake CPU, and the default general-purpose instance is randomly selected from CPUs, including vulnerable CPUs.

Conclusion. ShadowLoad can bypass mitigations relying on flushing secrets from the cache. On processors affected by L1TF, ShadowLoad is limited to prefetching on the same page, i.e., a 4096 B region containing the load target, which suffices to leak hypervisor addresses but not arbitrary memory. If attacks similar to L1TF are discovered on recent Intel processors with more powerful hardware prefetchers, the scope of data that can be leaked drastically increases. Thus, ShadowLoad should be considered in the threat model when mitigating microarchitectural attacks.

6.2 ShadowLoad with MDS

This case study shows how ShadowLoad can bring data into the cache, allowing an attacker to leak data at rest using Meltdown [26] on an affected CPU. As Meltdown is limited to leaking data in the L1 cache [38, 52], uncached data cannot be targeted directly. ShadowLoad can target such uncached data at rest, e.g., in the kernel, and let the hardware prefetcher bring it into the cache. Subsequently, we can use Meltdown to leak the cached data.

Setup. We use a kernel module as the victim and an unprivileged attacker for our proof of concept. The kernel module fills a page-aligned memory buffer with random data during initialization. Using `ioctl`, the kernel module exposes a memory load to buffer offset 2048, a gadget to flush the page from the cache, and a function bringing a chosen page offset into the cache. However, the latter is only used for a reference test on cached memory.

The attacker uses ShadowLoad to bring each cache line of the secret page into the cache and then leaks it using Meltdown. The attacker uses a cache-based Meltdown attack with return address mispredictions as a fault suppression mechanism [40] to leak the secret 4-bit at a time.

Evaluation. We run the Meltdown code with a secret of 64 B, i.e., one cache line, and measure setup and leakage time over a total of 10 000 executions on an Intel Xeon E5-1505M v5 (stepping 3, microcode `0xd6`) running Ubuntu 20.04.5 LTS (kernel 5.4.0-170-generic) with all mitigations except kernel page-table isolation (KPTI) enabled.

We run this test with three configurations: One using ShadowLoad to bring the secret into the cache ($\text{Meltdown}_{\text{ShadowLoad}}$), one attempting to leak uncached memory ($\text{Meltdown}_{\text{uncached}}$), and one with a hyperthread repeatedly accessing the secret cache line in the kernel ($\text{Meltdown}_{\text{cached}}$). $\text{Meltdown}_{\text{ShadowLoad}}$ correctly leaks 86.3 % of data at 203.2 kB/s, almost matching the 87.3 % at 203.8 kB/s of $\text{Meltdown}_{\text{cached}}$. $\text{Meltdown}_{\text{uncached}}$ leaks no data correctly. ShadowLoad's median one-time startup overhead is 11.9 µs, and the runtime overhead is measured at just 0.3 %. Our proof-of-concept also works with KPTI enabled and MDS mitigations disabled on an Intel Core i5-5010U, leaking data using RIDL [44].

When increasing the kernel buffer size beyond one page, we do not measure any leakage on the page not accessed by the gadget with $\text{Meltdown}_{\text{ShadowLoad}}$. This is in line with the results reported by StrideRE.

Targeting Kernel Code. To demonstrate that ShadowLoad also works on unmodified code in realistic scenarios, we target a load in the `getuid` system call of Linux kernel 6.8.0. We use ShadowLoad to cache the remaining data of the page, and subsequently like it with Meltdown. This increases the amount of data that can successfully be leaked by roughly factor ten compared to Meltdown without ShadowLoad.

Conclusion. ShadowLoad can enable Meltdown and MDS attacks on data that is never architecturally accessed. Thus, ShadowLoad expands the attack surface of such attacks to data at rest. While other techniques, such as Spectre gadgets, have been used to bring data into the cache [38], they rely on additional gadgets possibly not present within the victim. ShadowLoad only requires a memory load to data close to the secret [33].

6.3 Collide+Power with ShadowLoad

While previous case studies rely on CPU-specific vulnerabilities, Collide+Power operates across a wide range of CPUs [21]. For our Collide+Power proof-of-concept, we measure the power consumption of microarchitectural data collisions during prefetching. We use ShadowLoad to bring victim data into the L1 cache and evict the prefetched line with attacker-controlled data, combining their power of victim and attacker data. Hardware prefetching replaces the speculative prefetch gadgets [19] originally used [21]. We analyze and compare the derived leakage model of our new attack with the best case of the original Collide+Power attack when directly and speculatively accessing the *inaccessible* data using a load and a Spectre-RSB gadget.

Table 3. Results of the linear regression for the model coefficients and Pearson correlation between the measured power samples, the model for the complete model (ρ_{all}), and when only using the Hamming distance component (ρ_{hd}).

Gadget	ρ_{all} ·1	ρ_{hd} ·1	$\text{hd}(G,V)$ a in μW	$\text{hw}(G)$ b in μW	$\text{hw}(V)$ c in μW	Samples $\cdot 10^6$
Direct Access	0.3692	0.0750	109.47	528.16	−1.08	12
Spectre-RSB	0.0827	0.0071	11.12	129.13	−0.02	12
ShadowLoad	0.1098	0.0074	9.24	135.80	0.02	12

Setup. To mitigate the PLATYPUS attack [25], the Running Average Power Limit (RAPL) [17] interface is no longer unprivileged. Other unprivileged indirect power interfaces [27, 49] increase the overall noise, making attacks significantly slower. Thus, we rely on the RAPL interface for this proof-of-concept to enhance reproducibility. We exclude additional noise by placing the victim data directly in the attacker's virtual address space, removing the need for context switches. Finally, we implement the *nibble*-based amplification and differential measurement technique as described in Collide+Power to perform the comparison with fewer samples.

Evaluation. To record a single power sample, we measure the energy consumption for 10 ms using the RAPL interface over the following two steps: First, we prefetch the victim data into the L1 cache using ShadowLoad. Second, we evict the prefetched data using an L1 eviction set. We replace the first step with a memory access and a Spectre-RSB gadget for comparison, similar to the strongest gadgets presented in Collide+Power [21]. Finally, we post-process the samples using the same methodology and framework provided by Collide+Power [21]. Thus, we keep the attacker-controlled data G of the eviction set and the victim data V constant while recording a power sample. To compare the power leakage between the variants, we use the following power leakage model P and perform both a linear regression to compute the coefficients of the model and correlation analysis:

$$P(G,V) = a \cdot \text{hd}(G,V) + b \cdot \text{hw}(G) + c \cdot \text{hw}(V).$$

Table 3 shows the results of the linear regression when executing the proof-of-concept on an *Intel Core i9-9980HK* with the frequency fixed to 3.5 GHz. We record 12 million samples for the analysis per test. Our linear regression analysis shows that the exploitable Hamming distance leakage between the attacker and victim domain is reduced by a factor of 11.85 when using ShadowLoad compared to directly accessing the data. We assume this reduction in leakage performance is due to the memory accesses required to mistrain the hardware prefetcher. ShadowLoad shows comparable performance to

the Spectre-RSB gadget with only a reduction 1.20 times in the leakage coefficient. However, the Pearson correlation coefficient is 1.33 times stronger, indicating that less noise is present and that the model components represent the power consumption better when using ShadowLoad. Finally, the differential measurement technique [21] nullifies the influences of the Hamming weight of the victim data.

Conclusion. ShadowLoad enhances the threat model of Collide+Power by replacing the required speculative prefetch gadgets with hardware-based prefetches. Due to the minimal requirements of ShadowLoad, an attacker can target data that is never accessed by the victim, even when a specific Spectre prefetch gadget cannot *reach* the data. Finally, we show that ShadowLoad is comparable to a Spectre-RSB gadget even though it does not require mistraining and is the best-performing gadget for the Meltdown-Power attack [21]. Nevertheless, we leave further optimizations to increase the leakage signal as future work, for example, by potentially combining the prefetcher mistraining and the cache eviction.

6.4 FetchProbe on Side-Channel-Hardened Base64

In this case study, we attack the side-channel-hardened Base64 implementation of WolfSSL 5.7.2. WolfSSL uses an ASCII lookup table to decode Base64 characters, which, due to the non-continuous character set, spans over two cache lines. To prevent side-channel leakage, the implementation accesses both cache lines and uses masking to select the correct value. While this method appears secure at cache-line granularity, FetchProbe can monitor loads at sub-cache-line granularity, circumventing the mitigation and recovering Base64-encoded secrets.

Setup. We generate a random 256-bit key and Base64 encode it, resulting in a 44-character secret. We call the vulnerable `Base64_Char2Val` function on each encoded character and mount FetchProbe to infer the secret. Given FetchProbe's byte-granularity the access indices yield two possible values for 24 of the 64 Base64 characters, leaving a candidate set of 2 (1 bit). For all other characters, only one accessed value is valid. We can uniquely recover these characters and know the candidate set for the uncertain ones, allowing us to enumerate all possible keys via brute force. If any character is recovered incorrectly, we report the attempt as failed.

Evaluation. We execute our attack 10 000 times on an Intel Core i9-13900K, successfully recovering the partial secret in 9208 runs (92.1 %). In these 9208 runs, the required brute-force effort is reduced from 256 bit to 16.5 bit on average. Previous work [4] showed that even modern desktop CPUs can brute-force around 400 million AES keys per second, resulting in a brute-force time in the micro- to millisecond range. On average, the attack's execution time is 2.4 ms (leaking 106.7 kbit/s) and requires 1425.1 executions of the `Base64_Char2Val` function or 32.4 per character.

Conclusion. This case study demonstrates that FetchProbe can monitor memory accesses with sub-cache-line granularity, allowing it to target side-channel-resistant code. Thus, a threat model that assumes only cache-line granular observation is inadequate on affected CPUs. Side-channel-hardened code based on this model should be updated to defend against byte-granular attacks like FetchProbe.

6.5 FetchProbe with Spectre

In this case study, we use FetchProbe as encoding for Spectre to leak kernel memory using a simple secret-dependent memory access without a spreading factor [20]. Traditional cache side channels detect memory accesses with cache-line granularity, so 6 bits of the secret cannot be recovered. In contrast, FetchProbe can recover memory accesses with a granularity of single bytes on an Intel Core i9-13900K or 4 bytes on an AMD Ryzen 7 5700U.

Setup. In line with other research [37], we introduce our own Spectre gadget. We use a kernel module and expose the gadget using `ioctl`. The Spectre gadget only has a secret-dependent memory load without any spreading factor.

Using traditional side channels with cache-line granularity (64 bytes), only the most significant 2 bits of the secret value can be recovered. With FetchProbe on an Intel Core i9-13900K, all 8 bits of the secret can be recovered. On an AMD Ryzen 7 5700U, 6 bits can be leaked.

Evaluation. We initialize a page of kernel memory with random secret bytes. Using the Spectre gadget, the attacker speculatively encodes a secret byte into the internal prefetcher state and relies on FetchProbe as a side channel to recover it. An attacker can leak all secret bytes by repeating these steps with different offsets. We execute this code on two machines, an Intel Core i9-13900K and an AMD Ryzen 7 5700U. We additionally scale the secret by a factor of 4 on an AMD Ryzen 7 5700U to fully recover it. This attack still recovers twice as many bits as a traditional cache side channel using the same gadget. On an Intel Core i9-13900K, the attack leaks memory at 18.6 kB/s while correctly recovering 100.0 % of all bytes. On an AMD Ryzen 7 5700U, the leakage is 9.5 kB/s with a correctness of 98.8 %.

Conclusion. FetchProbe can leak memory accesses with sub-cache-line granularity and works during speculative execution. This precision enables Spectre gadgets not exploitable for traditional cache covert channels. We further modified InspectreGadget [51] to find such gadgets, resulting in an 24.8 % increase of the reported number of exploitable gadgets for the Linux kernel v6.6-rc4 from 1566 to 1955.

7 Related Work

This section discusses related work on hardware prefetchers and compares ShadowLoad and FetchProbe to other attacks.

Reverse Engineering Hardware Prefetchers. FetchBench [33] categorizes hardware prefetchers into models and provides a tool that automatically scans for various hardware prefetchers and reverse engineers their parameters based on

the model. We focus on stride prefetchers and reverse further details such as instruction-pointer and previously unknown load-address collisions needed for out-of-place mistraining. Chen et al. [6] similarly do not reverse-engineer mistraining conditions but other details, such as replacement policy or history table size on the instruction-pointer-based stride prefetcher on Intel. Further, Vicarte et al. [45] also reverse-engineer a data-dependent prefetcher on recent Apple cores.

Hardware Prefetcher Side Channels. Schlüter et al. [33] use a replay-based prefetcher attack to leak partial AES keys from a T-table implementation and build an ARM TrustZone covert channel. They exploit the shared prefetcher state, similar to FetchProbe but targeting a different prefetcher, to leak metadata across the privilege boundary of TrustZone. Chen et al. [6] exploit changes in the internal state of the Intel IP-stride prefetcher based on victim execution to leak secret-dependent control flow of the MbedTLS RSA implementation. While their attack targets the same prefetcher as FetchProbe, it does not leverage aliasing in the accessed memory and instead uses a Prime+Probe-like approach. Thus, it does not directly translate to AMD and cannot inject prefetches nor leak the target of a memory access but only whether a load instruction is executed. BunnyHop [56] exploits the instruction prefetcher on Intel CPUs to leak cryptographic keys. Vicarte et al. [45] exploit the data-dependent prefetcher on recent Apple cores to leak out-of-bounds data at rest with a Spectre gadget with Speculative Load Hardening enabled. These related attacks are summarized in Table 1.

Microarchitectural Injection-Style Attacks. Spectre-type attacks mistrain microarchitectural components such as the Branch Target Buffer (BTB) such that the CPU speculatively continues execution at an attacker-controlled address [20]. An attacker can transiently encode inaccessible data into the microarchitectural state and later retrieve it via a side channel. In contrast, ShadowLoad does not change the control flow but brings victim data into the cache to later leak it via another primitive, e.g., MDS or Collide+Power. Load Value Injection [43] transiently injects attacker-controlled data into a victim. An attacker-controlled value is transiently passed to a faulting victim load. Thus, in contrast to ShadowLoad, no predictor is mistrained.

Software-based Prefetch Attacks. Previous work also investigated software prefetching as a side channel [10, 13, 23]. However, in contrast to hardware prefetchers, software prefetchers are limited to the same security domain. They are also significantly simpler, as the prefetch target is provided as an argument. Thus, no data structure is required for the prefetch mechanism. Consequently, these attacks are orthogonal to our attacks.

8 Countermeasures

This section discusses and evaluates different countermeasures for mitigating ShadowLoad and FetchProbe on the hardware, firmware, and software layer.

Hardware. CPU vendors could use an address-space identifier to tag the predictor state with a security domain. As such identifiers are already used in the TLB [22], we expect this to be a feasible design change. Chen et al. [6] propose a privileged `clear-ip-prefetcher` instruction to clear the state of Intel's IP-based prefetcher when transitioning between security contexts requiring OS support.

Firmware. CPU vendors can provide mitigations in microcode. Similar to Spectre mitigations [15, 30], it could be possible to introduce new restricted predictors. We imagine a new MSR bit that changes the predictor such that it does not use predictions across security domains, akin to IBRS for Spectre [15]. A new MSR that lets software clear the predictor state could also be introduced.

Software. A simple mitigation to prevent all leakage from hardware prefetchers is to turn off all prefetchers. This is possible on Intel and AMD CPUs via model-specific registers [23, 47] but comes with a performance penalty.

Instead of turning off hardware prefetchers, we propose marking specific memory regions as not prefetchable using memory types. Like speculative loads, the CPU does not issue hardware prefetches to uncachable memory regions, as this could have unexpected architectural effects on memory-mapped devices [35]. Hence, by marking memory containing secret values as uncachable, they are protected not only from being leaked with Spectre attacks [35] but also from being fetched into the cache. Our experiments confirm that marking a memory range as uncachable using either memory type range registers [17] or the page-table entry prevents the hardware prefetcher from prefetching any values from this page. As uncachable memory introduces a performance penalty, Intel also proposed using a new memory type that only prevents speculative accesses [41]. If implemented, this could also be extended to mitigate prefetcher access.

As hardware mechanisms for clearing the predictor state cannot be retrofitted to existing CPUs, Chen et al. [6] propose instruction sequences that evict the prefetcher state at each context switch. Such sequences could be extended to other hardware prefetchers, reducing leakage over security domains. Similar mitigations have been used for other microarchitectural attacks [16]. However, due to the number of different hardware prefetcher implementations and lack of documentation, implementing such clearing sequences is cumbersome and prone to errors.

Side-channel-resilient code [3, 18], also called constant-time or data-oblivious, does not trigger secret-dependent prefetching without data-dependent prefetchers, therefore mitigating FetchProbe.

9 Discussion

This section discusses potentially vulnerable devices, the applicability to other hardware prefetchers, and potential further attack scenarios.

Vulnerable Devices. While recent CPUs are unaffected by MDS and L1TF, we still deem these case studies relevant: These case studies show that ShadowLoad is a viable primitive to increase the attack surface of such attacks and needs to be considered when mitigating existing and possible future attacks. Furthermore, affected processors are still widely used in the cloud, with AWS and Google Cloud heavily relying on Broadwell and Skylake CPUs.

Prefetcher Types. While we focus on exploiting the hardware stride prefetcher, the concept applies to other prefetchers that predict memory loads from previous accesses [33]. In principle, prefetchers with internal state that is not properly isolated between security domains can be targeted by ShadowLoad and FetchProbe. For instance, for stream prefetchers, prefetches following an attacker-controlled stream direction could be injected using ShadowLoad. Further, FetchProbe could guess the offset of a victim memory load and continue a stream based on this guess. By repeating the victim access and varying the guess, FetchProbe could learn the target of the memory load if the prefetcher state is not sufficiently isolated and aliasing between victim and attackers can be exploited. Similarly, ShadowLoad could inject prefetches with replay-based prefetchers like SMS prefetchers. FetchProbe targeting the SMS prefetcher yields attacks similar to the attack demonstrated by Schlüter et al. [33]. However, to analyze the capabilities of ShadowLoad and FetchProbe for certain prefetcher implementations, a comprehensive analysis of the prefetcher implementation is required. Thus, we limit our analysis to the stride prefetcher and leave other hardware prefetchers to future work.

Virtualization and TEE. Using QEMU (7.2.0) with KVM, we perform FetchProbe to extract the load offset within a VM, with the VM as the victim and the hypervisor as the attacker. Synchronization is achieved through interrupting the VM, with partial stride training before the interrupt. We verified that the prefetcher state remains unaffected by context switching between hypervisor and VM. Our observations indicate that when the load address and the load instruction address offset align with the VM's load, completing the stride is possible post-context switch.

Furthermore, our investigation extends to AMD Secure Encrypted Virtualization (SEV). While side-channel attacks are known within the TEE threat model [34], FetchProbe narrows cache attacks to a granularity of 4 bytes. However, as shown in Table 2, FetchProbe is limited to pre-Zen 3 microarchitectures. Thus, AMD-SEV SNP is currently unaffected. On Intel CPUs, our L1TF case study shows that ShadowLoad targeting the hypervisor is possible. In summary, ShadowLoad and FetchProbe transcent kernel (Section 5.5), hypervisor, and Trusted Execution Environment (TEE) boundaries.

Hyperthreading. We do not observe leakage when mistraining on the other hyperthread or on another core. This aligns with previous works [6, 45] which do not report cross-hyperthread or even cross-core prefetching.

Further Attack Scenarios. The possibility of bringing data at rest into the cache is relevant for a wide range of attacks, making FetchProbe and ShadowLoad powerful attack primitives. ShadowLoad is a generic primitive that can be used as a replacement for different techniques as used in Meltdown [26], MDS attacks [36, 44], ÆPICLeak [4], or Collide+Power [21]. Attacks such as Cache+Time [9] can also be mounted using ShadowLoad instead of branch predictors. Moreover, ShadowLoad could improve architectural race conditions by caching selected data and instructions, making their execution faster and biasing race conditions.

FetchProbe can be used as a replacement for cross-security-domain cache attacks such as L1 Prime+Probe [32]. With the higher granularity, it re-enables attack scenarios of CacheBleed [54], which are not possible since Intel Haswell. FetchProbe can monitor memory accesses similar to MWAIT-based attacks [55] or TSX-based covert channels [7].

10 Conclusion

In this paper, we showed that we can apply injection-style techniques known from Spectre and Load Value Injection attacks to hardware prefetchers. Building on our analysis, we presented ShadowLoad, a technique for bringing inaccessible victim data into the cache across security domains. In our case studies, we demonstrated that ShadowLoad extends the attack surface of existing microarchitectural attacks such as Meltdown and Collide+Power and can partially bypass L1TF mitigations. With StrideRE, we designed a framework to characterize security-relevant properties of hardware stride prefetchers automatically. Based on our results, we demonstrated FetchProbe, a side-channel attack to leak offsets of memory accesses with sub-cache-line granularity on various Intel and AMD CPUs. We showed that FetchProbe can leak secrets from cache-side-channel resilient code and exploit Spectre gadgets not exploitable using cache attacks. Our attacks show that microarchitectural injection is a powerful technique not limited to transient-execution attacks.

Acknowledgement

We want to thank the anonymous reviewers and the shepard Mengjia Yan for their valuable feedback and suggestions. We also want to thank Andreas Kogler for helping with experiments. We further thank the Saarbrücken Graduate School of Computer Science for their funding and support.

References

[1] 2023. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[2] 2024. AMD64 Architecture Programmer's Manual. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

[3] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*. https://www.usenix.org/system/files/sec22-borrello.pdf

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*. https://www.usenix.org/system/files/sec19-canella.pdf Extended classification tree and PoCs at https://transient.fail/..

[6] Yun Chen, Lingfeng Pei, and Trevor E Carlson. 2023. AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *ASPLOS*. https://dl.acm.org/doi/10.1145/3575693.3575719

[7] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-disselkoen.pdf

[8] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*. https://dl.acm.org/doi/10.1145/3173162.3173204

[9] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *S&P*. https://ieeexplore.ieee.org/document/10179399

[10] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*. https://dl.acm.org/doi/10.1145/2976749.2978356

[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*. https://dl.acm.org/doi/10.1007/978-3-319-40667-1_14

[12] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf

[13] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial prefetch: New cross-core cache side channel attacks. In *S&P*. https://ieeexplore.ieee.org/document/9833692

[14] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*. https://dl.acm.org/doi/10.1007/978-3-031-09484-2_7

[15] Intel. 2018. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

[16] Intel. 2021. Microarchitectural Data Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html

[17] Intel. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. https://cdrdv2.intel.com/v1/dl/getContent/671447

[18] Intel Corporation. 2020. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[19] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2022. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2022-221-paper.pdf

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*. https://ieeexplore.ieee.org/document/8835233

[21] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2023. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity23-kogler.pdf

[22] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*. https://ieeexplore.ieee.org/document/9230388

[23] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security*. https://www.usenix.org/system/files/sec22-lipp.pdf

[24] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_lipp.pdf

[25] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2020. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*. https://ieeexplore.ieee.org/document/9519416

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf

[27] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency throttling side-channel attack. In *CCS*. https://dl.acm.org/doi/10.1145/3548606.3560682

[28] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*. https://ieeexplore.ieee.org/document/7163050

[29] LKML. 2018. Re: Linux 4.18.1. https://lkml.iu.edu/hypermail/linux/kernel/1808.2/00177.html

[30] LKML. 2018. x86/pti updates for 4.16. http://lkml.iu.edu/hypermail/linux/kernel/1801.3/03399.html

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*. https://dl.acm.org/doi/10.1007/11605805_1

[32] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*. https://papers.freebsd.org/2005/cperciva-cache_missing

[33] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. 2023. FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers. In *CCS*. https://dl.acm.org/doi/10.1145/3576915.3623124

[34] Michael Schwarz and Daniel Gruss. 2020. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security & Privacy* (2020). https://ieeexplore.ieee.org/document/9107096

[35] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for

Mitigating Spectre. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24271-paper.pdf

[36] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*. https://dl.acm.org/doi/10.1145/3319535.3354252

[37] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*. https://dl.acm.org/doi/10.1007/978-3-030-29959-0_14

[38] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2021. Speculative Dereferencing of Registers: Reviving Foreshadow. In *FC*. https://link.springer.com/chapter/10.1007/978-3-662-64322-8_15

[39] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS*. https://dl.acm.org/doi/10.1145/3243734.3243736

[40] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018). https://arxiv.org/abs/1806.07480

[41] Ke Sun, Rodrigo Branco, and Kekai Hu. 2019. A New Memory Type Against Speculative Side Channel Attacks. https://github.com/IntelSTORMteam/Papers/blob/main/2019-A_New_Memory_Type_Against_Speculative_Side_Channel_Attacks.pdf

[42] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-van_bulck.pdf

[43] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*. https://ieeexplore.ieee.org/document/9152763

[44] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*. https://ieeexplore.ieee.org/document/8835281

[45] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W Fletcher, and David Kohlbrenner. 2022. Augury: Using data memory-dependent prefetchers to leak data at rest. In *S&P*. https://ieeexplore.ieee.org/document/9833570

[46] Gregory Vish. 2018. l1tf-poc. https://github.com/gregvish/l1tf-poc

[47] Vish Viswanathan. 2014. Disclosure of Hardware Prefetcher Control on Some Intel Processors. https://web.archive.org/web/20151114175224/https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

[48] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. 2019. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05B-3_Wang_paper.pdf

[49] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *USENIX Security Symposium*. https://www.usenix.org/system/files/sec22-wang-yingchen.pdf

[50] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf

[51] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. 2024. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity24-wiebing.pdf

[52] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/23105-paper.pdf

[53] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf

[54] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *JCEN* (2017). https://link.springer.com/chapter/10.1007/978-3-662-53140-2_17

[55] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity23-zhang-ruiyi.pdf

[56] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security Symposium*. https://www.usenix.org/system/files/usenixsecurity23-zhang-zhiyuan-bunnyhop.pdf

A Artifact Appendix

A.1 Abstract

Our artifacts demonstrate how our hardware prefetcher attacks FetchProbe and ShadowLoad work. The artifacts include our tool StrideRE and proof-of-concepts from the paper and allow reproduction of their results. The artifacts require a 12th or 13th-generation Intel processor and an Intel processor affected by Meltdown (e.g., Skylake or Coffee Lake). The artifacts only work on x86_64 Linux and require root access.

A.2 Artifact check-list (meta-information)

- **Hardware:** Intel 12th Gen or 13th Gen processor + Intel processor affected by Meltdown (e.g., Skylake or Coffee Lake)
- **Run-time environment:** Linux, Root required
- **How much disk space required (approximately)?:** 1GB
- **How much time is needed to prepare workflow (approximately)?:** a few minutes
- **Experiments:** Python scripts for evaluation are available
- **How much time is needed to complete experiments (approximately)?:** about 12 hours to several days (if Collide + Power is evaluated)
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT / GPL
- **Archived (provide DOI)?:** 10.6084/m9.figshare.28381319

A.3 Description

A.3.1 How to access. The artifacts can be obtained in the following repository: github.com/cispa/ShadowLoad.

A.3.2 Hardware dependencies.

- 12th or 13th-generation Intel processor
- Intel processor affected by Meltdown (e.g., Skylake or Coffee Lake)

We tested the provided artifacts using an Intel Core i7-8700k and an Intel Core i9-12900k. Two processors are required as our provided attacks are fine-tuned for hardware prefetchers that can cross page boundaries, and other attacks require processors vulnerable to Meltdown. The former is only available on newer processors, while the latter is only available on old Intel processors. Some of the provided artifacts can be executed with different processors but might require additional tweaking.

A.3.3 Software dependencies. Linux operating system (no VM), git, kernel headers, GNU Make, gcc, python3, matplotlib. We used Ubuntu 22.04 as an operating system during testing. Matplotlib should also be set up work when python3 is executed as root.

A.4 Installation

After downloading the artifacts, run the *compile_and_check.py* as root (*sudo python3 compile_and_check.py*). This script compiles all examples and kernel modules and should terminate with exit code 0 or provide more information on failure.

A.5 Experiment workflow

Experiments are divided into different directories. Most directories contain an *eval.py* script that will automatically run the experiment and reports the results or stores them in a directory called *out*. The expected results and more details on how each experiment can be run are detailed in the next section.

A.6 Evaluation and expected results

00_fetch_probe. (5-10 minutes)
The experiment should be executed on a 12th or 13th-generation Intel processor. The *eval.py* script must be executed as root. An example output contains the following lines:

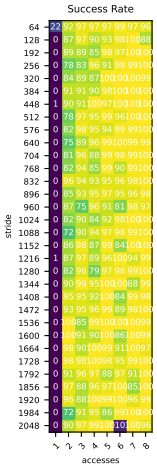
```
(...)
times: 0.022398622
correct: 32767.0
false_positives: 0.0
false_negatives: 1.0
positives: 16383.5
negatives: 16384.5
precision: 100.0%
recall: 100.0%
f-score: 100.0%

(...)
times: 0.043121079
correct: 32766.0
false_positives: 0.0
false_negatives: 2.0
positives: 16380.0
negatives: 16388.0
inv_correct: 32766.0
inv_false_positives: 0.0
inv_false_negatives: 2.0
inv_positives: 16388.0
inv_negatives: 16380.0
precision: 100.0%
recall: 100.0%
f-score: 100.0%
inv_ precision: 100.0%
inv_ recall: 100.0%
inv_ f-score: 100.0%
```

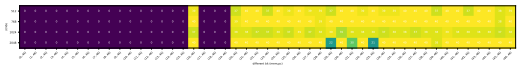
The experiment is successful if the amount of false positives and false negatives is low. The precision, recall, and f-score should be comparable to the numbers shown in the paper but at least 80%.

01_shadow_load. (about 5 minutes)
The experiment should be executed on a 12th or 13th-generation Intel processor. The *eval.py* script must be executed as root. After execution, the *out* directory should contain, amongst others, the files *result_shadowload_1.svg*

and *result_shadowload_kernel_1.svg*. These images should show cache hits starting with two accesses:



02_stride_re. (up to 12 hours)
We recommend using a 12th or 13th-generation Intel processor for this experiment. The *eval.py* script must be executed as root. This experiment may take up to 12 hours to complete. After execution, the *tests/out* directory should contain many files. For instance, the file *test_prefetch_both_collisions_mem_aligned_2_rdtsc,kernel,-DEVAL,-DACCESS_MEMORY.svg* which visualizes the relevant bits in the accessed address for userspace to kernel mistraining of the hardware stride prefetcher with two training accesses and an aligned trigger access in the kernel. This should look comparable to this example on 12th or 13th-generation Intel processors:



03_base64. (2 - 10 seconds)
The experiment should be executed on a 12th or 13th-generation Intel processor. The *eval.py* script can be executed as an unprivileged user. The experiment is successful if the incorrect rate is low (it should be below 30%). The incorrect rate describes the number of runs out of 1000, where at least one bit was incorrect.

Sample output:

```
(...)
leakage: 87104.25 bit/sec.
incorrect: 194 / 1000 (19.40%
unknown left: 16.56575682382134
invokations : 1238.1042183622828
```

04_meltdown. (30-90 minutes)
The experiment should be executed on an Intel processor vulnerable to Meltdown. Page-table-isolation should be disabled (*mitigations=off* or *nopti* kernel parameter).
On Ubuntu 22.04, the kernel parameters can be changed by editing the *GRUB_CMDLINE_LINUX_DEFAULT* configuration in */etc/default/grub* and executing *update-grub* afterward. The *analyze.py* script (run by the *eval.py* script) further enables huge pages using `sysctl -w vm.nr_hugepages=50`. The script must be modified if this command is does not apply to the system. The *eval.py* script must be executed as root. The amount of correct bytes should be at least 50% (50% is better than random guessing which would be $\frac{1}{255}$).

05_spectre. (approximately 3 minutes)
The experiment should be executed on a 12th or 13th-generation Intel processor. The *eval.py* script must be executed as root. The experiment is successful if the amount of correct bytes is high (should be at least 80%).

Sample output:

```
(...)
rate    : 18.2KB/s
correct: 99.4%
--------------------
(...)
```

06_collide_power. (several days)
We included the Collide+Power victim for completeness. It can be evaluated using the open source tool by Kogler et al.: github.com/0xhilbert/rda. Further instructions are included in the *README.md* in the repository at *06_collide_power/runner/user/README.md.*

A.7 Notes

If anything fails, ensure that no kernel modules from other experiments are still loaded.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae