FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers

Till Schlüter
till.schlueter@cispa.de
CISPA Helmholtz Center for Information Security

Amit Choudhari
amit.choudhari@cispa.de
CISPA Helmholtz Center for Information Security

Lorenz Hetterich
lorenz.hetterich@cispa.de
CISPA Helmholtz Center for Information Security

Leon Trampert
leon.trampert@cispa.de
CISPA Helmholtz Center for Information Security

Hamed Nemati
hamed.nemati@cispa.de
CISPA Helmholtz Center for Information Security

Ahmad Ibrahim
ahmad-ibrahim@hotmail.de
Unaffiliated

Michael Schwarz
michael.schwarz@cispa.de
CISPA Helmholtz Center for Information Security

Christian Rossow
rossow@cispa.de
CISPA Helmholtz Center for Information Security

Nils Ole Tippenhauer
tippenhauer@cispa.de
CISPA Helmholtz Center for Information Security

ABSTRACT

Prefetchers speculatively fetch memory using predictions on future memory use by applications. Different CPUs may use different prefetcher types, and two implementations of the same prefetcher can differ in details of their characteristics, leading to distinct runtime behavior. For a few implementations, security researchers showed through manual analysis how to exploit specific prefetchers to leak data. Identifying such vulnerabilities required tedious reverse-engineering, as prefetcher implementations are proprietary and undocumented. So far, no systematic study of prefetchers in common CPUs is available, preventing further security assessment.

In this work, we address the following question: *How can we systematically identify and characterize under-specified prefetchers in proprietary processors?* To answer this question, we systematically analyze approaches to prefetching, design cross-platform tests to identify and characterize prefetchers on a given CPU, and demonstrate that our implementation FetchBench can characterize prefetchers on 19 different ARM and x86-64 CPUs. For example, FetchBench uncovers and characterizes a previously unknown replay-based prefetcher on the ARM Cortex-A72 CPU. Based on these findings, we demonstrate two novel attacks that exploit this undocumented prefetcher as a side channel to leak secret information, even from the secure TrustZone into the normal world.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623124

KEYWORDS

Microarchitecture, Side Channel, Prefetching

ACM Reference Format:
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 *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623124

1 INTRODUCTION

Fast and efficient CPUs are the backbone of our digital infrastructure. To provide the high level of performance we are used to today, modern CPUs are equipped with a myriad of elaborate, proprietary, and often undocumented optimization mechanisms. Unfortunately, those mechanisms often come at the cost of security vulnerabilities that break fundamental guarantees by the platform, such as memory isolation. For example, transient execution attacks exploit code that the CPU executes only transiently, e.g., speculatively after control or data-flow mispredictions (Spectre-type attacks), or out-of-order after a faulting instruction (Meltdown-type attacks) [6, 21, 24].

While security issues related to transient execution received a lot of attention, less scrutiny has been given to predictive fetching of memory by *hardware prefetchers*, or *prefetchers* for short. Prefetchers are proprietary components of a CPU that try to predict future memory accesses and bring blocks of memory into the cache before they are actually requested. As the problem of detecting and accelerating memory access patterns can be approached in many different ways, a multitude of prefetcher design options is available to the chip designers [11]. Because prefetch operations are transparent to software applications, vendors usually do not disclose any detailed information on the types of prefetchers they use, characteristics of these prefetchers, and their implementation. For example, Intel has confirmed the existence of certain types of prefetchers by disclosing documentation on how to disable them [37] but limits public documentation of their characteristics to superficial recommendations

for software developers [19]. ARM is even more restrictive: For instance, the technical reference manual for the ARM Cortex-A72 confirms the existence of a prefetcher for data but does not even specify what kind of memory access patterns it detects [1].

This secrecy is especially problematic since security researchers identified several vulnerabilities that arise out of specific prefetcher implementations. For example, prior work characterized the stride prefetcher in certain Intel CPUs and exploited the reverse-engineered characteristics to build covert channels [7, 8] or to leak private keys from ECDH [32] or RSA [7] computations. Researchers also identified and characterized a prefetcher that dereferences pointers on the Apple M1 [29]. They exploited this prefetcher as an oracle for the validity of virtual addresses and thus to circumvent Address Space Layout Randomization (ASLR). These examples show that prefetchers are a security-relevant component of a CPU that requires systematic investigation. However, all prior work focused on a narrow set of prefetcher designs and CPUs; so far, no systematic analysis of a wide range of prefetcher architectures and their security implications has been provided.

Research Questions and Challenges. In this work, we close this gap by answering the following three research questions:

RQ1: How can we systematically identify and characterize prefetching mechanisms on unspecified architectures?
RQ2: What are the main types and characteristics of prefetchers in modern CPUs?
RQ3: What are the security implications of so-far unexplored prefetcher types?

We identified the following three main research challenges when addressing these questions: First, we need to collect and systematize possible prefetcher designs that CPU vendors are likely to use, despite having only limited documentation available. In addition, we have to identify the prefetcher's relevant implementation variables (*characteristics*). Second, as a CPU does not directly expose the prefetcher's internals, the identification and characterization of the prefetcher have to be performed based on its behavior during operation. While prior work mainly focused on characterizing a specific type of prefetcher [7, 8, 29, 32], we need solutions that allow identifying multiple prefetcher designs and characterizing each of them appropriately. Third, we want to verify that security-relevant inner workings of prefetcher designs enable new vulnerabilities. Such demonstrations will also require means of synchronization between victim and attacker code.

Proposed Approach. We first classify different prefetcher designs and create a taxonomy (see Figure 1), allowing us to relate prefetcher designs to each other and to identify common properties as well as fundamental differences. We then design and implement FetchBench, a modular framework that uses our taxonomy to identify a prefetcher design on a given CPU. In addition, our framework characterizes the prefetcher w.r.t. the most relevant characteristics (such as the trigger method or prefetch depth) for this particular design. We apply our framework to 19 processors of seven different vendors and two architectures (ARMv8 and x86-64).

With FetchBench, we are able to identify a variant of the replay-based Spatial-Memory-Streaming (SMS) prefetcher [35] in three ARM Cortex-A-series processors. We investigate the security properties of this prefetcher, which has received little attention so far.

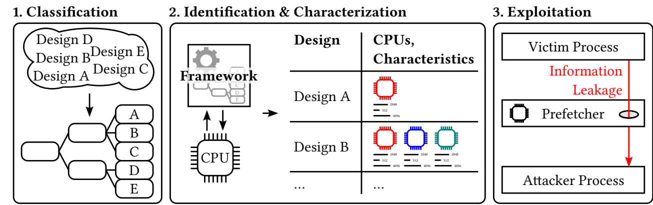

Figure 1: Proposed Approach

We demonstrate use cases for selected vulnerabilities. In particular, we exploit the SMS prefetcher to leak half of an AES key across process boundaries and show that the prefetcher's state is exempted from the usual separation between secure and non-secure world on TrustZone-enabled ARM CPUs. We build a covert channel that has a data rate of 1245 Bytes/s at 5.02 % error rate.

Contributions. Our main contributions are as follows:

- We compile and classify seven hardware data prefetcher designs based on a novel taxonomy.
- We design and implement FetchBench, a modular framework to identify and characterize prefetchers on real CPUs, and apply those tests to 19 CPUs of seven different vendors and two different architectures. FetchBench uncovers and characterizes a previously unknown, replay-based prefetcher on a series of ARM Cortex-A processors.
- We present a novel attack on the replay-based prefetcher of the ARM Cortex-A72, leaking 64 bits of an AES-128 key through the prefetcher. As part of the attack, we address the non-trivial problem of synchronizing the attacker process with the victim process to make the attack practical on Linux in presence of a scheduler.
- We show that the prefetcher's internal state is not separated between normal and secure world, leaking metadata across the TrustZone privilege boundary.

Disclosure. ARM acknowledged our findings on shared prefetcher state across hardware contexts and assigned CVE-2023-33936.

2 BACKGROUND

CPU Memory Hierarchy and Prefetching. Contemporary processors employ a hierarchy of memory denoted by caches to reduce the gap between processing time and memory access time. Caches decrease the time spent on memory operations as perceived by the processor by storing recently or frequently used chunks of data in fast memory. A cache is divided into fixed-size cache lines (CLs), which describe the basic unit for data transfers between the memory (in terms of "memory lines") and the processor.

Traditionally, caches contain memory that was requested by the CPU in the past, improving the memory latency for subsequent accesses to the same data. In contrast, prefetching aims to improve the latency on the initial load of a memory line. For this, cache lines that are likely needed in the near future are identified and fetched into the cache in advance. There are two fundamentally different approaches to prefetching: software and hardware prefetching. Software prefetching relies on prefetch instructions. These instructions indicate which memory addresses will soon be accessed, so that the CPU can bring them into the cache before they are used.

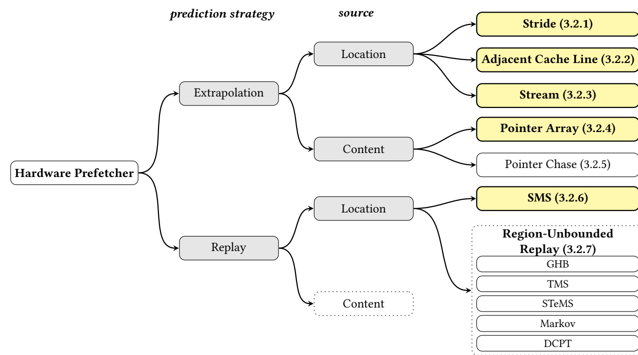

Figure 2: Prefetcher taxonomy. Yellow leaf nodes identify prefetcher designs that we experimentally observed in CPUs.

Hardware prefetching, which is our focus, comprises all mechanisms that analyze memory accesses performed at runtime and dynamically predict which addresses are likely to be loaded next. While various CPU buffers may support prefetching, such as instruction caches, data caches [33], and Translation Lookaside Buffers (TLBs) [31], we focus on prefetching into data caches. The two core aspects of a prefetcher implementation are the design of the *training* and the *prediction* mechanism. The (sometimes optional) training mechanism determines how the prefetcher learns patterns. The prediction mechanism uses the previously trained patterns to decide which addresses to prefetch next. CPUs often implement multiple hardware prefetchers that complement each other.

Timing Attacks. Due to limited resources available on contemporary computing devices, resource sharing among processes is inevitable. However, if not done carefully, resource sharing can introduce unintended information flow channels, known as side channels, which are potentially exploitable by a malicious process to exfiltrate secret information on a system. Prominent representatives of such attack techniques are Flush+Reload [41] and Prime+Probe [26]. In Flush+Reload, an attacker process first flushes the shared cache lines from the cache. Second, the victim process executes and, depending on a secret value, either loads the flushed entries or not. Finally, the attacker measures the time needed to reload these entries. Depending on the reload latency, the attacker decides whether the cache entry was accessed by the victim and is thereby capable of extracting the secret. In Prime+Probe, the attacker first primes the selected cache entries. When the victim returns, the attacker measures the victim's secret-dependent memory operations on the primed entries to infer the secret.

3 SYSTEMATIC CLASSIFICATION OF HARDWARE PREFETCHERS

In this section, we describe and identify prefetcher designs commonly found in CPUs and in academic literature. In Section 3.1, we propose a two-level taxonomy based on the strategy used to decide what to prefetch and on the information source the prefetcher is trained on. Based on our taxonomy, we present a framework to automatically detect and characterize prefetchers (Section 3.2).

3.1 Hardware Prefetcher Taxonomy

Prefetcher Design Selection. A plethora of prefetcher designs have been discussed in the past, either in form of academic proposals, reverse-engineering results, or hardware documentation. With our goal of identifying prefetchers *in real-world hardware* in mind, we focus on designs that we consider likely to be implemented. We use the *Primer on Hardware Prefetching* by Falsafi and Wenisch [11] as a starting point. From their selection, we focus mainly on designs that (i) are known to be present in real-world hardware or (ii) do not require large (off-chip) data structures to keep their state. We follow the conjecture of Ayers et al. [3] that prefetchers with high memory requirements have a higher "implementation burden" and are less likely to be implemented. We further include designs that are present in real-world hardware but not covered by [11].

Figure 2 summarizes our prefetcher classification with respect to two dimensions that we describe in the following: (i) the prediction strategy and (ii) the source of prediction knowledge.

Prediction Strategy. We distinguish two types of strategies used for the prediction, *extrapolation* and *replay*:

- *Extrapolation*-based prefetchers detect sequences in recent memory accesses and extend these by prefetching likely future accesses—notably irrespective of (and agnostic to) past accesses to these prefetched locations.
- *Replay*-based prefetchers repeat known access patterns and thereby prefetch previously-observed accesses (only)—without trying to extend access sequences.

Extrapolation-based prefetchers aim to identify *and extend* memory access patterns. Once identified, the prefetcher uses these patterns to compute ("extrapolate") which memory locations are *likely* to be accessed next. This way, those prefetchers can boost performance even if a memory region is not accessed repeatedly. Usually, extrapolation-based prefetchers do not need to maintain an extensive access history, and—depending on the prefetcher—can even be fully stateless. Consequently, such prefetchers typically have a more lightweight internal state. A disadvantage of these prefetchers is that they cannot predict more complex access patterns.

Replay-based prefetchers *repeat* past memory accesses patterns. Unlike extrapolating prefetchers, they are not bound to simple patterns (e.g., monotonic access sequences). Instead, they rely on more involved data structures to capture even complex access patterns. The prefetched access patterns are thereby identical to (or a subset of) previously-observed accesses. Replay prefetchers do *not* extend access patterns but may replay accesses in other contexts (e.g., at different locations). Designs in this domain differ in the way how the history is stored and filtered and how the prediction is triggered. Some prefetchers operate on fixed-size memory regions [35], others are region-unbounded. Moreover, there are different trade-offs between memory consumption and accuracy, e.g., by deciding to only store longer and more frequent patterns [11, 38].

Prediction Source. For the second level of our classification, we use the source of the prediction, i.e., what property the hardware prefetcher uses to decide on the address to prefetch. We distinguish between *location*-based and *content*-based sources.

- *Location*: Only the location, i.e., address, of the memory load is used to decide what to prefetch.

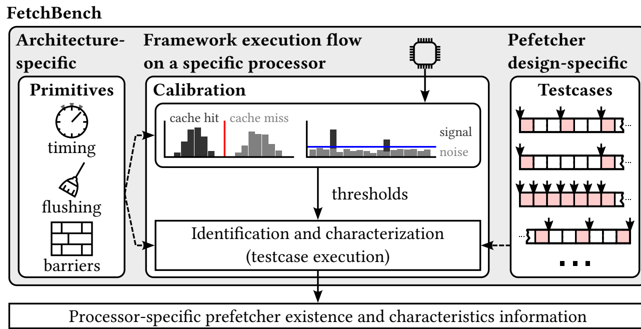

Figure 3: FetchBench Framework Overview

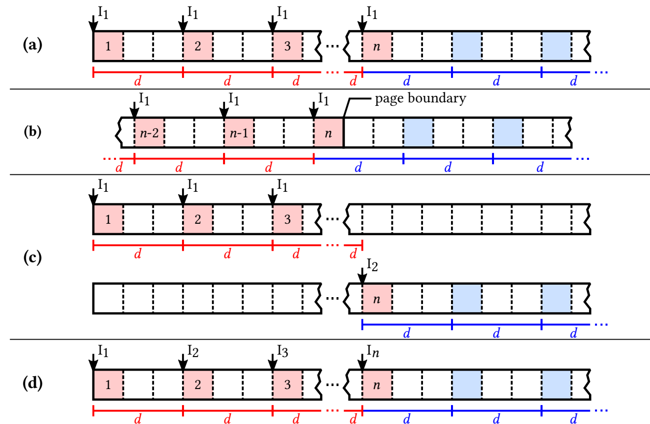

Figure 4: Stride prefetcher tests. Dashed boxes represent cache lines within a memory page. Arrows indicate loads and are labeled with the load instruction (I_x). Red CLs are cached due to architectural loads, blue CLs are potential prefetch locations. Boxes are numbered by the order of access.

- *Content*: The content of the memory load is used to decide what to prefetch.

The location of memory loads is a common source for many hardware prefetchers, both for extrapolating as well as for replaying prefetchers. Typically, these prefetchers detect common patterns found in many applications [3] or previously-observed access patterns. Simple patterns include strides, i.e., the linear access to memory with a constant offset between the accessed memory addresses. Based on the strategy of the prefetcher, a detected pattern either leads to an extrapolated completion or a history-based completion.

Prefetchers using the content of memory loads as a prediction source can prefetch indirect memory loads. The prefetcher predicts future accesses based on the observed value, not on the address where a value is stored. Such prefetchers can (heuristically) detect pointers and start to prefetch the target memory location, e.g., to improve the performance of linked lists or trees. These data structures are usually not contiguous, as the next item that is accessed during a list traversal is only referenced via a pointer.

3.2 FetchBench: Automatically Identifying and Characterizing Prefetchers

In this section, we propose FetchBench, an automated modular software framework to identify and characterize hardware prefetchers. Whereas knowledge about the existence and characteristics of prefetchers can, in principle, be acquired manually, such tedious efforts are typically bound to single CPU models. Time-consuming manual attempts thereby cannot generalize the findings to the by-now large landscape of CPU models.

Deep knowledge of prefetchers and their implementations is critical for security. First, the sole existence of a certain prefetcher determines if a given system is vulnerable to prefetcher-based side-channel attacks [7, 32] *in principle*. Second, only the precise knowledge of the exact characteristics of a given prefetcher implementation—which is normally not or only sparsely documented by CPU vendors—helps to understand if a particular defense is effective. Third, new insights into prefetcher implementations may reveal novel prefetcher-based side-channel vulnerabilities.

Overall Design. Figure 3 provides a high-level overview of FetchBench. The framework consists of one module for each potential prefetcher design. Each module implements test cases that first detect whether a prefetcher is implemented and, if so, what its characteristics are. We focus especially on characteristics that we consider security-relevant, i.e., those that may enable a side-channel or impact the quality or throughput of a potential side-channel. Modules essentially provide memory access patterns and queries for the cache state of memory locations. Based on a sequence of tests, a module infers the inner workings of the prefetcher. Tests are architecture-agnostic, as only access sequences and expected cache states are defined. This modular design allows adding new tests for different prefetchers. The functionality to access memory and query the cache state of specific cache lines is common for all modules but specific to the architecture that the framework runs on. On a given CPU, the framework first calibrates the primitives used to measure cache states. It then runs the test cases specified by the modules for the different prefetcher designs. In the end, the experimental results indicate the existence and characteristics of the tested prefetcher designs on the CPU. For our FetchBench prototype, we implement modules for seven prefetchers. In the following, we describe the design of these modules and how they detect and characterize hardware prefetchers.

3.2.1 Stride. Stride prefetching [4, 11] aims to detect sequences of accesses that are spaced a constant offset (*stride*) apart from each other. Once the prefetcher has detected such a pattern, a trigger causes it to extrapolate: The prefetcher predicts data locations by adding multiples of the detected stride to the last accessed address. Figure 4 (a) shows a sequence of n memory accesses (in red) that form a pattern with a stride of $d = 3$ cache lines, along with potential prefetch locations (in blue).

Characteristics. We distinguish between the following stride prefetcher characteristics: (i) Does the prefetcher also learn *backward* strides? (ii) Which are the *smallest/largest strides* that are supported? (iii) What is the *minimal training size*, how large is the *prefetched set*? (iv) Does the prefetcher *cross the page boundary*? (v) On which *granularity*, i.e., minimal data unit (e.g., cache line), does the prefetcher operate? (vi) What *triggers* the prefetching phase, e.g.,

a previously-trained instruction or an accessed memory address, and can there be trigger collisions?

Identification and Characterization. Figure 4 (a) illustrates how we test for the presence of a stride prefetcher. In our base test, we train a cache-line-aligned stride pattern repetitively (in red) and measure if the CPU caches memory past our accesses (in blue). We then expand our base test to measure each of the characteristics individually. To reveal the direction, supported strides, and training size, we change the direction, offsets, and number of data accesses, respectively, and test whether prefetching occurs. The number of prefetched cache lines may depend on the selected stride and the number of preceding training accesses, so we analyze the number of prefetches for different strides and after different numbers of training steps. To test whether the prefetcher crosses the page boundary, we align the sequence of training accesses to the end of a page, as illustrated in Figure 4 (b). If we see any prefetching on the following page, the prefetcher can cross the page boundary. To test the granularity, we use known-good parameters for the stride and the number of training steps but add a small random offset between 0 and (*cache line size* − 1) to each loaded address. If the prefetcher detects this pattern and completes it, we conclude that the prefetcher operates on addresses at cache-line granularity.

To identify the trigger, we perform three different tests. The most likely triggers are the instruction address of a load instruction, the data address accessed, or fractions or combinations of these. To verify this assumption, we start with a negative test. We train the prefetcher on a first memory page using a first load instruction (I_1) and try to trigger it on a second memory page using a second load instruction (I_2), as illustrated in Figure 4 (c). If the prefetcher is triggered by a combination of instruction and memory addresses, we expect no prefetching, since we changed significant portions of both. Next, we test whether the prefetcher can successfully be trained by different load instructions loading from the same memory page (Figure 4 (d)). Lastly, we test whether the prefetcher can be trained by a single load instruction that is executed with different operands multiple times (for example in a loop) across different pages. This is similar to Figure 4 (c), but using I_1 throughout all accesses. If we observe prefetching for the instruction address test but not for the memory page test, we conclude that the instruction address is the only trigger. In the opposite case, the memory address is the trigger. If both tests show prefetching, the prefetcher uses either the instruction address or the memory address as a trigger. For prefetchers that use an instruction address as a trigger, we further test whether only a fraction of that address is considered. We re-use the experimental setup of Figure 4 (c) but align the addresses of I_1 and I_2 such that their c least-significant bits (LSBs) match. We increase c and stop as soon as we start seeing prefetching, indicating a collision. If a collision occurs, we conclude that the prefetcher only stores the c LSBs of the instruction address as a trigger.

3.2.2 *Adjacent Cache Line.* When accessing a cache line, the adjacent-cache-line prefetcher [37] automatically prefetches one of the neighboring lines.

Characteristics. The prefetcher can either be forward-fetching or block-fetching. A forward-fetching prefetcher always loads the line after the accessed one. In contrast, a block-fetching prefetcher divides memory into blocks of 2 cache lines, starting at an address that is a multiple of 2 cache line sizes. When a memory address is accessed, this prefetcher always loads the sibling cache line from the same block, i.e., preceding *or* subsequent cache line.

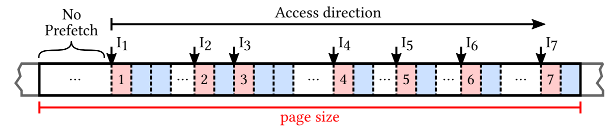

Figure 5: Stream prefetcher experiment. Load instructions (I_x) are performed in one direction.

Identification and Characterization. First, we access a cache line at an address that is a multiple of 2 cache line sizes. If the following line is prefetched, we identified an adjacent-cache-line prefetcher. We further access a cache line that would be the second line in a 2-cache-line block. If this access prefetches one cache line in backward direction, the prefetcher is block-fetching. Otherwise, we characterize it as forward-fetching.

3.2.3 *Stream.* A stream prefetcher [19, 28] is designed to prefetch subsequent locations for an irregular access pattern that runs in a constant direction, using extrapolation. Although its behavior may appear similar to that of an adjacent-cache-line prefetcher, there are three key differences. First, the stream prefetcher starts prefetching only when the direction is known, which requires at least two accesses. Second, it can prefetch multiple subsequent cache lines. Third, it prefetches cache lines in the same direction as the access, whereas an adjacent-cache-line prefetcher exhibits block or constant prefetching. Compared to the stride prefetcher, which is also directional and extrapolation based, the stream prefetcher does not need a regular pattern to initiate prefetching.

Identification. We develop an identification test to detect stream prefetchers by generalizing previously reverse-engineered properties of the Intel Kaby Lake CPU [28]. As illustrated in Figure 5, the test involves initiating a load instruction I_1 at a specific cache line (in this example, the 10th line). Thereafter the test continues to access cache lines in an irregular pattern using load instructions I_2, I_3, and I_4, on cache lines 16, 18, and 32, respectively, in that order. These irregular loads stop at the end of a page, and we measure the prefetched lines. Our test ensures that no two consecutive pairs of accesses have the same stride to prevent false positives from other types of prefetchers, such as the stride. Additionally, we use distinct load instructions for every load to prevent false prefetches from replay-based prefetchers that trigger based on the Program Counter (PC). In order to not restrict the test to just Intel's implementation [28], we perform more than four directional loads before observing the prefetches. We consider the test successful and identify a stream prefetcher when all prefetches run in the same direction, there is at least one instance of consecutive prefetches adjacent to load (e.g., cache lines 11 and 12), and no prefetching is observed before the first access (I_1).

3.2.4 *Pointer Array.* Arrays of pointers are common data structures that linearly store addresses of relevant data locations. The target addresses do not necessarily point to linear memory locations. Pointer array prefetchers [29, 42] analyze memory content to identify arrays of pointers and then prefetch their targets.

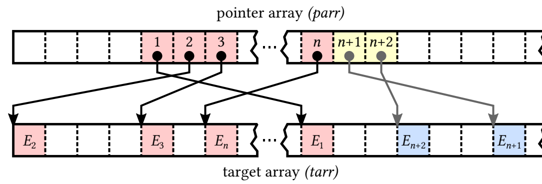

Figure 6: Pointer array experiment. Dashed boxes represent cache lines within a memory page. Red CLs are cached due to architectural loads during training, yellow and blue CLs are potential prefetch locations. Boxes are numbered by the order of access.

Characteristics. We distinguish the following prefetcher characteristics: (i) Does the prefetcher also detect *backward* iterations? (ii) How many *training accesses* are required? (iii) What is the maximum amount of data that is prefetched? (iv) How many *pointers* are prefetched? (v) Is the trigger instruction pointer dependent?

Identification and Characterization. Figure 6 shows how we test for the presence and features of a pointer array prefetcher. In our test, we set up an array of pointers (*parr*), each referring to random entries within a sparse target array (*tarr*). First, we evict all entries of *parr* and *tarr* from the cache. Second, we ensure the address translation is cached for each page of *tarr* by accessing one cache line on each page. After that, we iterate over *parr* and dereference the first n training pointers. Then, we measure the time it takes to dereference pointer $n + 1$ and access a predetermined random entry of *tarr* that was not accessed during training. If a pointer array prefetcher is present, we observe a cache hit for pointer $n + 1$ but a cache miss for the random entry.

To unveil parameters of the prefetcher, we vary the number of training pointers n in *parr*, the offset of the measurement pointer from the last accessed *parr* pointer, and the number of bytes we measure past the start of a possibly prefetched *tarr* entry. To identify whether the prefetcher also activates on backward iteration, we start accessing and dereferencing the last entry of *parr* and decrement the index with each iteration. To determine whether the prefetching behavior depends on the memory location of the instruction accessing *parr* or dereferencing the pointer, we unroll the loop accessing and dereferencing pointers in *parr*. In an additional test, we spread these instructions across different memory pages and align them to different page offsets.

3.2.5 *Pointer Chasing.*

In a chain of pointers, each pointer depends on its predecessor in the access sequence. The next address is stored where the previous pointer points to, as for example in a linked list. A pointer-chasing prefetcher [3, 11] loads subsequent pointers, reducing the stalling times when traversing such data structures. Note that related work [29] uses the term "pointer chase" to describe any behavior where data is directly used as an address. Our work, however, only uses the term in scenarios where the address of the next element of the data structure is unknown until it can be read from memory (e.g., a linked list). This effectively serializes memory accesses and prefetching, preventing parallelization.

Identification. As, to our knowledge, this type of prefetcher cannot be found in commodity hardware, we do not distinguish any characteristics for this type of prefetcher and only present an existence benchmark. In our test, we set up a linked list where each node of the list resides on a different cache line. The order of the list is randomized to prevent other extrapolation-based or replay-based prefetchers from interfering. First, we evict all entries of the array from the cache. Next, we iterate over the pointer chase and dereference the first n training pointers. Then, we measure the access time to pointer *n+1* and a predetermined random entry that was not previously accessed. If the prefetcher is present, we detect a cache hit for pointer *n+1* but a cache miss for the random entry.

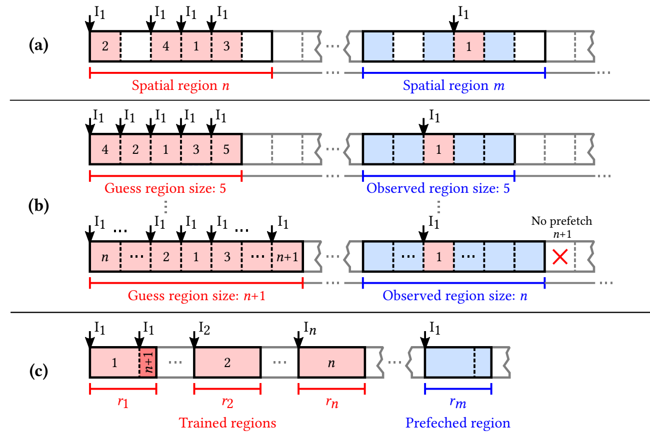

Figure 7: SMS prefetcher experiments. Dashed boxes represent cache lines, bold boxes represent the spatial region. The red cross represents a miss for an expected prefetch.

3.2.6 *Spatial Memory Streaming (SMS).*

Load instructions often access structured elements and exhibit repetitive access patterns in spatially-nearby memory relative to initial memory access. An SMS prefetcher [35] targets such use cases, which are common for a repetitive data structure such as socket buffers (`sk_buff`) in Linux. The prefetcher operates in two stages: training and prediction, which can be triggered by a particular load instruction (PC-triggered) or memory location (Mem-triggered). For instance, in the training stage, a PC-triggered prefetcher learns all the memory accesses over a given load instruction in spatial memory. Eventually, when a different spatial region is accessed with the same load instruction, it initiates the prediction stage to prefetch learned memory access patterns—speculating that the load instruction will show similar access patterns in other regions.

Characteristics. We identify the following characteristics of an SMS prefetcher. (i) What triggers the prefetcher to start learning memory access patterns? (ii) What spatial boundary does the prefetcher use for learning memory accesses? (iii) Is there a difference in the prefetching behavior in forward or backward direction from the first load from a region? (iv) Can the training continue when another load instruction's address partially collides with the trigger instruction? (v) How many spatial regions can the prefetcher learn simultaneously in the training stage?

Identification and Characterization. We leverage the two stages of the SMS prefetchers to design our tests. As depicted in Figure 7, we demonstrate the PC-triggered SMS prefetcher tests. First, we induce a pattern in the prefetcher's training stage. Second,

we trigger the prefetching stage to measure the prefetched cache lines in a different spatial region. Figure 7 (a) shows the base test case where we induce a pattern by accessing spatially-nearby cache lines in one region, followed by trigger access in a new region (in red). We then compare the cache lines prefetched in the new spatial region (in blue) with the memory accesses in the trained region. If they are identical, the test confirms the presence of the SMS prefetcher. Extending the base test, we develop tests to identify the relevant factors for successful training and prediction stages. These factors comprise the address of the load instruction and the address offset of the first memory access. Figure 7 (b) illustrates the technique used to determine the spatial boundary for the prefetcher. We gradually increase the spatial size and stop when the prefetched cache lines do not match the trained pattern. Knowing the spatial boundary, we also investigate the maximum size of the induced pattern for training when the consecutive memory accesses are in the same direction (forward or backward).

We further designed tests to identify the number of independent spatial patterns that can be learned simultaneously in the training stage. As illustrated in Figure 7 (c), we train the subject region r_1 with a few loads (I_1). We continue training $n-1$ additional regions r_2, r_3, until r_n. The training phase completes with one additional $(n+1)$ load I_1 in region r_1. For the load (I_1), we measure the prefetch corresponding to the last access $(n+1)$ in region r_m. If cache line $(n+1)$ is prefetched, the spatial region r_1 was not evicted from the training stage. The probability of eviction of the region r_1 increases with the number of simultaneously trained regions. However, due to unknown replacement policies, this test only provides an estimate.

3.2.7 Generalized Region-Unbounded Replay Prefetchers. As explained in Section 3.1, replay-based prefetchers aim to prefetch irregular but repetitive memory access patterns. The SMS prefetcher (Section 3.2.6) covers accesses that occur in spatially-nearby locations, but cannot cope with spatially-distant accesses. This is where region-unbounded replay prefetchers come into play. For example, when scanning a large unclustered database, although the database is logically sequential, the memory allocation is fragmented, making region-unbounded replay prefetchers more effective.

These prefetchers typically require more space to store the metadata necessary to detect unbounded and irregular patterns. To balance the accuracy and space trade-off, researchers have proposed innovative designs such as circular FIFO buffers, as implemented in the Global History buffer (GHB) [25], table-based as in Markov [20], storing a history in off-chip memory as in Temporal Memory Streaming (TMS) [39] and Spatio-Temporal Memory Streaming (STeMS) [34], and storing relative distance instead of complete addresses as in Delta Correlating Prediction Tables (DCPT) [13]. Despite these differences in design, the fundamental property to learn a sparse access pattern and replay it when detected remains common to all region-unbounded replay prefetchers.

Identification. As illustrated in Figure 8, we train the prefetcher by repeatedly accessing a sequence of a few memory locations at spatially-distant locations, where every memory access results in a cache miss. Subsequently, we flush the page from the cache and access only the initial few locations of the sequence to test the prefetching behavior. If the prefetcher correctly prefetches the subsequent memory locations, a region-unbounded prefetcher is present.

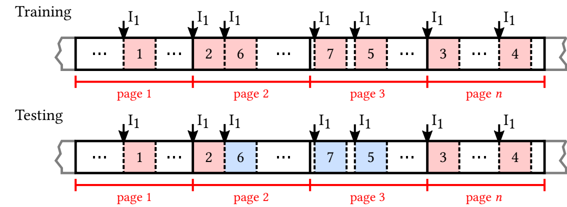

Figure 8: Region-unbounded prefetcher identification test

Classification. There are many academic variations of region-unbounded replay prefetchers but no known instances on real CPUs. Existing tests in FetchBench, such as tests for the trigger or the number of prefetches, can serve as a reference for more detailed identification tests. For instance, a variant of the GHB prefetcher (GHB PC/AC) utilizes the PC to localize a cache miss and subsequently employs address correlation (AC) to detect the access pattern. Using the memory correlation primitive in FetchBench is sufficient to identify such a prefetcher, as it will prefetch for the same memory address but not for a different address.

4 CHARACTERIZING ARMV8 AND X86-64 PREFETCHERS

4.1 Implementation

Our framework is implemented in C/C++ and Assembly, consisting of approximately 8,100 lines of code. It currently supports the ARMv8 and x86-64 architectures. The source code of our framework is available at https://github.com/scy-phy/FetchBench.

To implement the tests outlined in Section 3.2, we require three primitives on each platform: a precise timing source, a method to flush cache lines from the cache, and memory barriers. The flushing primitive allows training or triggering the prefetcher from a clean cache state. The timing source and the barrier enable us to determine whether a memory line is cached or not through a Flush+Reload cache side channel [41]. This allows us to observe prefetching behavior. On x86-64 CPUs, we use the `clflush` instruction to flush cache lines. On ARM-based platforms, we flush using the `DC CIVAC` instruction. On the Apple M1, we enable this feature through the privileged model-specific register `S3_0_C15_C4_X` first. To collect precise timestamps, we either call `clock_gettime`, use a counter thread, or use architecture-specific means like the `rdtscp` instruction on x86-64 or the *Performance Monitors Cycle Count Register (PMCCNTR)* on ARM (if available). We observe different noise characteristics for these methods across different platforms, so we choose the option that produces the least noise for each of them.

To ensure that the memory activities of the Flush+Reload side channel do not impact the results, we always probe one cache line at a time. More precisely, we run an experiment, probe the cache state of one memory location, repeat the experiment, probe a different location, and so on. This ensures that our probe accesses do not form patterns that could impact the prefetcher's behavior.

Table 1: Prefetcher identification and characterization results. Bold tests are existence tests.

(a) Stride Prefetcher (3.2.1)

Processor → / ↓ Characteristics	A53	A55	A72	A73	A76	M1i	M1f	i7SB, i5HW, XeSL, XeCL, i7CL	i3IL, i7TL, XeIL	i9ALp	R5Z, R5Z+	R7Z3, R9Z3+
Pos./neg. direction	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●
Min./max. stride (B)	±64/ ±256	±64/ ±2048	±64/ ±4096	±64/ ±2048	±64/ ±8192	±128/ ±256	±128/ ±8192	±64/ ±1024	±64/ ±8192	±64/ ±16384	±64/ ±8192	±64/ > ±16384
Min./max. prefetches	3/5	1/28	1/16	1/31	1/18	8/20	8/16	1/2-5	1/2-6	1/8	1/5-7	5/15
Trigger	Mem	PC/Mem	PC/Mem	PC/Mem	PC/Mem	Mem	Mem	PC	PC	PC	PC	Mem
PC collision (bits)	N/A	—	12	—	15	N/A	N/A	8	10	10	12	N/A
Cross page boundary?	○	●	●	●	●	●	●	○	●	●	●	●
Strides < 1 CL?	○	○	○	○	○	●	●	●	○	●	○	○
Strides with random inner-CL offsets?	●	●	●	●	●	●	●	●	●	●	●	●

Not identified on i9ALe.

(b) Ptr. Array Prefetcher (3.2.4)

Processor → / ↓ Characteristics	M1f
Existence	●
Trigger	Mem
Pos./neg. direction	●/●
Max. prefetch size	256
Max. prefetch amount	16 ptrs.
No. training pointers	2

Not identified on all other processors.

(c) SMS Prefetcher (3.2.6)

Processor → / ↓ Characteristics	A72	A73	A76
Trigger	PC	PC	Mem
Region size (B)	1024	1024	1024
PC collision (bits)	12	—	—
Pos./neg. direction	12/9	16/11	12/9
No. of entries (est.)	5	9	10

Not identified on all other processors.

(d) Other Prefetchers

Processor → / ↓ Prefetcher	i7SB, i5HW, XeSL, XeCL, i7CL	i3IL, i7TL, XeIL	i9ALe
Adjacent CL (3.2.2)	●B	●F	●F
Stream (3.2.3)	●	●	○
Pointer chase (3.2.5)	○	○	○
Region-unbounded replay (3.2.7)	○	○	○

●B Block ●F Forward; None identified on all others.

4.2 Experimental Setup

We run FetchBench on 19 different processors in total, comprising six ARMv8 SoCs, nine Intel x86-64 CPUs, and four AMD Ryzen CPUs. We provide a list of all testing environments in Table 2 in the appendix, where we also assign them short IDs to refer to them throughout the paper. Our selection of ARM-based platforms comprises five Cortex-A-series designs, ranging from the Cortex-A53 to the Cortex-A76, as well as the low-energy and performance cores of the Apple M1 Max SoC (dubbed *Icestorm* and *Firestorm*). The selected Intel CPUs comprise *Core* and *Xeon* models from the Sandy Bridge to the Alder Lake generation. We further test four AMD Ryzen CPUs from the Zen to the Zen 3+ microarchitecture. We run all our tests on each processor. We assume the analyst has root access to the systems under evaluation. We use a Linux distribution that is provided or recommended by the respective vendor for each of the ARM boards, as listed in Table 2.

4.3 Experimental Results

According to our tests, all processors we examined implement at least one prefetcher design, most even two or more. We present the detailed results of our identification and characterization in Table 1 and summarize the most significant findings.

To the best of our knowledge, we are the first to identify and characterize a replay-based prefetcher in real-world hardware. As shown in Table 1 (c), our tests uncover previously unknown SMS prefetchers in the ARM Cortex-A72, -A73, and -A76 processors. All use a region size of 1 KiB. The prefetchers in A72 and A73 use the Program Counter (PC) as a trigger, i.e., they map the instruction address of a load instruction to a spatially-bounded memory access pattern. The A72's prefetcher cannot distinguish trigger instruction addresses with 12 or more identical least-significant bits. This enables address collisions, causing the prefetcher to apply spatial access patterns learned in one region to another. As we show in Section 4.3, such collisions pose a security risk, as they leak memory access patterns across privilege domains. The SMS prefetcher on A76 is memory-triggered, i.e., it stores an access pattern for a particular spatial region in memory. In addition, we discover that the SMS prefetchers on both A72 and A73 store the accessed memory locations within a region as distances relative to the first accessed memory line of a region. We suspect that the prefetcher stores the cache state of a limited number of memory lines around the initial access as a bit vector. Consequently, these prefetchers miss accesses at one end of the region when the initial access occurs close to a region boundary. Due to unknown replacement policies, we only estimate for the number of entries in the SMS prefetchers: 5 entries on the A72, 9 entries on the A73, and 10 entries on A76.

As revealed in Table 1 (a), the most commonly identified prefetcher in our test is the stride prefetcher, implemented in all of the tested processors except the Alder Lake efficiency core (i9ALe). The ARM-based CPUs in our test show considerably different stride-prefetch behavior. The prefetcher on A53 is the least aggressive: The maximum stride is low (256 B), and at most 5 cache lines

are fetched in advance. In contrast, the stride prefetcher on A76 detects strides up to 8 KiB. More aggressive prefetching can also be observed on A55 and A73, loading up to 28 and 31 cache lines upfront, respectively. While stride prefetching can be triggered by a matching load instruction address or an access to a nearby memory address for most Cortex-A-cores, the A53 and the Apple M1 (M1i, M1f) rely only on memory addresses as a trigger. The stride prefetchers on the Intel CPUs from the Sandy Bridge to the Comet Lake generations (i7SB, i5HW, XeSL, XeCL, i7CL) behave almost identically in our test, only differing in the maximum number of prefetched cache lines. We confirm the finding of Chen et al. that this prefetcher uses only the 8 least-significant bits of a load instruction to reference a stride pattern internally [7]. In line with Xiao et al., we see prefetching for strides smaller than the cache line size [40]. In the Ice Lake and Tiger Lake generations (i3IL, i7TL, XeIL), the stride prefetcher is enhanced to detect larger strides of up to 8 KiB. The number of relevant instruction address bits increases from 8 to 10. Our Alder Lake CPU features a hybrid design. On its performance cores (i9ALp), strides up to 16 KiB are detected. We did not identify a stride prefetcher on the efficient cores (i9ALe). On the AMD CPUs based on the Zen and Zen+ microarchitectures (R5Z, R5Z+), we find a PC-triggered prefetcher that detects strides up to 8 KiB. In contrast, the prefetcher in the Zen3 and Zen3+ generations (R7Z3, R9Z3+) is memory-triggered and detects strides up to a size that exceeds our testing capabilities. All stride prefetchers in our test can be triggered by a pattern that forms a regular stride at cache-line granularity but has random inner-cache-line offsets.

In our test, all Intel CPUs from Sandy Bridge to Comet Lake complement the stride prefetcher by a block-fetching adjacent-cache-line prefetcher and a stream prefetcher. Starting from Ice Lake, the adjacent-cache-line prefetcher is forward-fetching instead. The Alder-Lake CPU has no stream prefetcher and an adjacent-cache-line prefetcher only on the efficient core.

Matching Sanchez Vicarte et al.'s results [29], we identify a pointer array prefetcher on the Firestorm cores of the Apple M1 CPU (M1f). We confirm that the trigger does not depend on the location of the instruction and examine additional features such as the size of prefetched entries. We detect prefetching starting with 2 training pointers in contrast to 3 as reported in prior work [29].

None of our CPUs feature a pointer-chasing prefetcher or a region-unbounded prefetcher. We suspect the primary reason for this is storage limitations. Region-unbounded prefetchers tend to require significant amounts of memory to store an access history, making their implementation expensive in practice. To verify that our test for region-unbounded prefetchers works in general, we execute it on the implementation of the DCPT (Delta Correlating Prediction Table) prefetcher [13] in the gem5 microarchitecture simulator [5]. The DCPT prefetcher uses a PC-indexed table pointing to a circular buffer of deltas. Our test case successfully detects the region-unbounded prefetcher in this simulation.

The execution time of FetchBench mainly depends on the CPU performance and the identified prefetchers. In our tests, the runtime ranged from 27 minutes (R9Z3+) to 6.2 hours (A53). We illustrate the success rates of our experiments in Appendix B.

5 EXPLOITING PREFETCHERS

In this section, we investigate the security properties of a prefetcher that we identified in hardware for the first time: The SMS prefetcher. We know from FetchBench's results (see Table 1 (c)) that the implementation of this prefetcher on the ARM Cortex-A72 is PC-triggered. Further, trigger collisions may occur when the addresses of two load instructions share the 12 least-significant bits. This combination of characteristics is problematic, as it allows us to train a prefetcher in one context and observe its prefetching behavior in another. We present two case studies to illustrate this finding. First, we show that we can exploit the prefetcher's characteristics to leak secret-dependent memory accesses from one process to another. Second, we demonstrate that the prefetcher does not properly separate patterns from normal world and secure world, allowing for side-channel leakage from TrustZone.

5.1 Case Study: AES Attack

We exploit the SMS prefetcher on the ARM Cortex-A72 to leak parts of an AES key from one userspace process to another. Lookup-table-based AES implementations have been target of side-channel attacks in the past [15, 36]. Some attempts relied on tables being shared between attacker and victim process, enabling techniques such as Flush+Reload and Flush+Flush [15]. Approaches not based on shared memory faced the non-trivial challenge of profiling tables without knowing their addresses and cache-set mappings [36].

Our attack targets an implementation where lookup tables are *not* shared, making Flush+Flush and Flush+Reload infeasible. In contrast to prior approaches, we avoid the need to learn the table's memory addresses by exploiting the prefetcher. The prefetcher uses partial *instruction* addresses as a trigger, making it susceptible to address collisions—an insight gained from FetchBench. This allows us to leak memory access patterns in a targeted way based on the addresses of load instructions that access the tables. Prior attacks considered the prefetcher rather a source of noise [36], while we use it as a source of leakage.

We further address the problem of extracting a secret-dependent prefetcher state before it is overwritten or evicted by subsequent memory loads. While prior work modified the victim code to work around this problem [32], we synchronize the attacker process with the victim and interrupt it soon after the first AES key addition. We implement this synchronization primitive by combining Flush+Reload on code [41] with inter-processor interrupts [22].

System and Attacker Model. We perform the attack on our A72 system, a Raspberry Pi 4 with four ARM Cortex-A72 cores. We attack the default AES-128 implementation of Mbed TLS 3.3.0 (released December 2022), which is based on T-tables. We assume a chosen-plaintext scenario: The victim process encrypts an attacker-specified plaintext using a fixed 128-bit key targeted by the attacker.

The attacker can execute native code in userspace on the same CPU core as the victim and uses the PMCCNTR register as a timing source. We assume that the attacker can execute the victim program repeatedly, each time providing an arbitrary plaintext. Both the attacker and the victim process have access to the same (shared) Mbed TLS instance in memory. However, we emphasize that only the library code is shared in memory, not the T-tables. The tables are generated in victim memory during library initialization. The

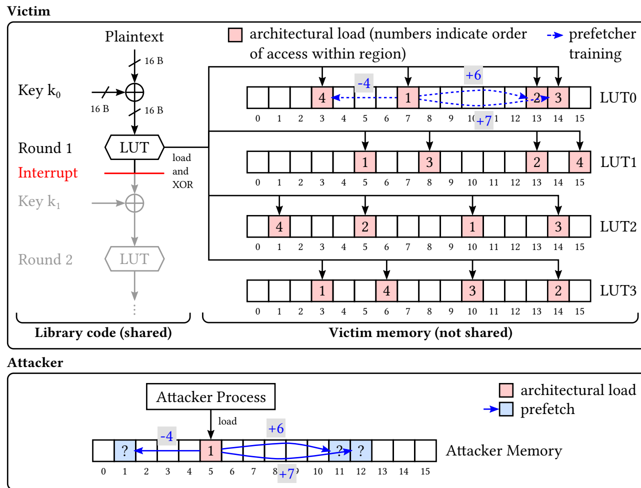

Figure 9: Loads and prefetches in victim/attacker process

attacker can thus apply Flush+Reload only to code (to synchronize with the victim process) but *not* to the T-tables (as prior attacks did to infer access patterns; we exploit the prefetcher instead).

Victim Program Procedure. The victim process receives a plaintext as a command line parameter and encrypts it using the Mbed-TLS library. During library initialization, four lookup tables of 256 4-byte integers each are generated. The generated tables are local to the victim process and thus inaccessible to the attacker. The default table size of 1 KiB (or 16 cache lines) matches the region size of the A72's prefetcher. As illustrated in Figure 9, the attacker-provided plaintext bytes are XORed with the AES key k_0 in the first encryption round. The 16 resulting bytes are secret-dependent and are used as indices to load elements from the lookup tables—the source for the prefetcher-based leak.

Cache-Line Granularity. Each cache line of the lookup tables contains 16 table elements and we expect the same prefetch behavior regardless of which of those elements is accessed. As a consequence, our attack cannot distinguish offsets within a cache line and can thus only infer the upper 4 bits of each access location and key byte. This limits us to leaking only half of the AES key. We emphasize however that knowing 64 key bits reduces the brute-force effort significantly, as an attacker knows which key bits are missing. To illustrate this, we refer to past approaches to brute-force block cipher keys. *Deep Crack*, a machine built in 1998 to brute-force a DES key, tried 92 billion keys per second [10]. At this pace, brute-forcing a full AES key would take $5.9 \cdot 10^{19}$ years on average, while 64 key bits could be found in only 3.2 years on average. As brute-forcing can be parallelized, for example in the cloud, a modern solution could be even faster.

Prefetcher Activation. When a lookup table is first accessed in the victim process, the prefetcher is activated, sets up a new region, and associates it with the address of the initial load instruction. For all subsequent loads from the same lookup table, the prefetcher stores the distances (in cache lines) between the initial memory address and the memory addresses of the three following accesses. For instance, in Figure 9, the initial load from LUT0 targets cache line 7, and the next three loads target lines 13, 14, and 3. The prefetcher stores the secret-dependent distances relative to the first access, i.e., $13 - 7 = 6$, $14 - 7 = 7$, and $3 - 7 = -4$.

Attack Procedure. Our attack has 3 steps. First, we determine the distances between the initial access and the three later accesses to a lookup table. Second, we find the offset of the initial access from the beginning of the lookup table. Finally, we decide which of the three leaking distances is influenced by which key byte. Since the procedure is the same for all tables, we describe it for only one.

Initial leakage. The attacker provides an arbitrary plaintext to the victim process, interrupts it after the first AES round, and extracts the prefetcher's state. We explain the interrupt mechanism in the next section. To extract the state, the attacker executes a load instruction that is located at a colliding address to the first load in the library accessing the table. This instruction address is public and can be extracted from the shared library binary. The address is also unaffected from ASLR, as the relevant lower address bits are usually not randomized. Executing the colliding load instruction activates the prefetcher in the attacker's memory space, as shown in Figure 9. The prefetcher erroneously associates the load with earlier memory activity in the victim process and prefetches three more addresses according to the previously recorded distances, starting from the initial load location in the attacker's memory. The attacker recovers the distances from the cache state of their own memory using a Flush+Flush cache side channel. However, the three distances are insufficient to recover the corresponding key bits. Additionally, (i) the offset of the initial load from the beginning of the table, and (ii) the order of the three later accesses are required.

Leaking the initial offset. To identify the offset of the initial access from the beginning of the table, the attacker needs to determine the 4 most-significant bits (MSBs) of the byte that influences the initial load from the table. To this end, the attacker flips one of the 4 MSBs of the corresponding plaintext byte and repeats the experiment. As a result, all three leaking distances move by a constant offset in either a positive or negative direction. The direction reveals the bit. The attacker repeats this process for each of the 4 MSBs.

Leaking the remaining offsets. The attacker already knows the three distances between the initial and the later loads but does not know which input bytes influence which of the distances. To find this mapping, the attacker flips a bit in one of the plaintext bytes that control the three later table accesses and repeats the experiment. As a result, only one of the three distances changes. The attacker maps this distance to the mutated plaintext byte and uses this knowledge to compute the four key bits for the corresponding key byte.

After this process is completed for all four lookup tables, the attacker successfully leaked 64 bits of the AES-128 key.

Synchronization. The attacker faces two challenges when synchronizing their process with the victim. First, they need to interrupt the victim process when the prefetcher's state is secret-dependent, as later memory accesses could alter that state. Second, they have to schedule attacker code that extracts the prefetcher's state as soon as possible after the interrupt on the victim's CPU core. We approach the challenge of synchronizing attacker and victim with a Flush+Reload attack on the library code [41] to identify the right moment for an interrupt and an inter-processor interrupt [22] to schedule attacker code.

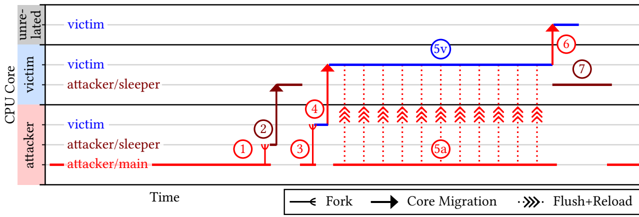

Figure 10: Synchronization of attacker and victim processes

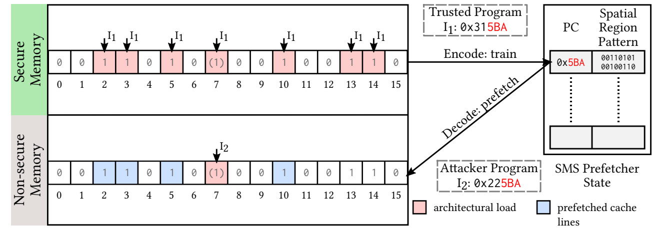

Figure 11: Covert channel based on the SMS prefetcher. Top: The sender encodes a bit vector in the SMS region (in red). Bottom: The receiver triggers a PC-colliding load to reveal this bit vector via the prefetcher regions (in blue).

The detailed procedure we use is illustrated in Figure 10. We are interested in the prefetcher's state as soon as possible after the first round of accesses to the AES lookup tables is completed (highlighted in red in Figure 9). After starting the main attacker process, we fork a child process (1), which we refer to as the *sleeper* process, and move it to the core where we later run the victim on. We block the sleeper process, for instance by making it wait for a semaphore controlled by the main attacker process. Next, the main attacker process starts the victim process (3) and moves it to the victim core (4). While the victim runs and performs the encryption (5v), the main attacker process performs a Flush+Reload attack on the shared library code (5a). More precisely, we select the address of a function that is called just before the encryption (`mbedtls_aes_setkey_enc`) as a trigger. We flush this address in a loop and reload it while measuring the memory latency. As soon as we notice a low latency, we know that the victim called the function. At this point, the main attacker process wakes the blocking sleeper process through the semaphore and triggers an inter-processor interrupt, e.g., by moving the victim process to another core (6) or calling the `membarrier` system call [22]. The scheduler interrupts the victim process on the victim core and schedules the sleeper process (7). The sleeper can then perform a memory load from an instruction at a colliding code address and inspect the cache state of its own memory to recover the prefetcher's state.

The synchronization may interrupt the victim process slightly too early or too late. However, we can easily filter the recovered prefetcher states based on the number of accesses: If we interrupt the victim process at the right moment, we expect to leak three distances. We ignore all other samples for our attack.

Evaluation. We repeated the attack with 5 random keys on a Raspberry Pi 4 and successfully recovered the 4 MSBs of 12.4 AES key bytes on average (maximum: 16, minimum: 8). The average runtime of our attack is 30 hours. We use 256 different plaintexts for each execution run and repeat each plaintext encryption between 10 000 and 15 000 times. Our synchronization mechanism provides us with the expected cache state in 0.19 % of these cases. While this hit rate may seem low, almost all unexpected samples are easy to detect and filter out based on the number of cache hits, as we primarily expect samples containing three distances.

Edge cases. The attack as described above assumes that the T-tables are 1 KiB-aligned (i.e., aligned to prefetch regions) in memory. The table alignment is determined at compile time and can change between compiler versions. If the tables are misaligned and span across two regions each, the attack can be adapted by tracking accesses in both regions or by mutating the plaintexts such that all accesses are moved into one of the regions. Similarly, if two memory accesses hit the same memory table element, less than three distances are leaked. This problem can be solved by mutating the plaintext until this is no longer the case.

5.2 Case Study: ARM TrustZone Covert Channel

Our second case study demonstrates a covert channel based on the SMS prefetcher characteristics uncovered by FetchBench. Our prefetcher-based covert channel uses the prefetcher's state containing the access pattern—as encoded by the sender and retrieved by the receiver—to implicitly send information between processes. We demonstrate that such covert channels are possible even if the two processes run in vastly different privilege domains. To this end, we leverage FetchBench's characterization of the SMS prefetcher implementation on ARM Cortex-A72. Using ARM TrustZone [2], we create a covert channel between a user space application in a non-secure world and a trusted app in a secure world. The covert channel thereby (a) demonstrates missing state separation between secure and non-secure world in a prefetcher, (b) approximates the upper bound of the data transmission rate.

The general idea of the covert channel is as follows. Given a spatial region comprising N cache lines, the sender uses one of these lines as a trigger and encodes the bits to send into the state of the remaining $N - 1$ cache lines. The receiver uses a colliding load instruction to trigger the prefetcher. Immediately thereafter, the receiver derives the transferred bits from the prefetcher's prediction.

Implementation. To run our attack on a fully TrustZone-enabled platform, we prototype our attack on a Rock 4 SE board. This board is based on a Rockchip RK3399-T SoC, which is TrustZone-supported [27] and contains four Cortex-A53 and two Cortex-A72 processor cores. We only use the A72 cores for the covert channel. We run Linux in a non-secure world and OP-TEE, an implementation of ARM TrustZone, in a secure world.

The concrete implementation of this covert channel requires a detailed understanding of the underlying prefetcher internals. We make use of the characterization obtained by FetchBench as reported in Section 3.2.6: The spatial region size is 16 cache lines. The prefetcher is PC-triggered and uses the 12 least-significant bits of the PC to map it to a region pattern. This implies that the sender and receiver applications only need to align 12 bits of their load instruction addresses to map them to the same pattern in the prefetcher's state. FetchBench also identifies a limitation on the A72 CPU: While a spatial region has a size of 16 cache lines, the

prefetcher is unable to capture the state of some of those lines in some situations. More precisely, the prefetcher can only record the state of up to 9 cache lines before and 12 cache lines after the initial (trigger) access to a region. Hence we choose the trigger location as the middle of the region, allowing us to make use of all cache lines of a region. Furthermore, FetchBench finds that at least five region patterns can be stored simultaneously in the SMS prefetcher. Thus, we optimize the data rate by training *multiple* region patterns in each iteration; we then empirically validated that training four region patterns in parallel hits the sweet spot between low error rate and high throughput.

Figure 11 illustrates the covert channel in detail. First, the attacker aligns the userspace code such that the least-significant 12 bits of the receiving load instruction collide with those of the sending load instruction in the secure world. Second, the sender encodes data into the prefetcher's state as discussed before—we use the cache state of each cache line within a region to encode one bit of information. Every accessed cache line (in red) corresponds to a set bit, while cache lines corresponding to unset bits are not accessed. Because one cache line (here: the 7th) serves as the trigger which must always be accessed, we cannot use this cache line to encode a data bit. Consequently, we encode 15 bits of data per region pattern into the SMS prefetcher's state. Finally, the receiver triggers prefetching (in blue) in userspace by accessing the 7th cache line in a region in the receiver's memory space using a colliding load instruction. By simultaneously training four region patterns in this way, we can transfer 60 bits per iteration.

Evaluation. To evaluate the performance of the covert channel, we transfer 1 MiB of test data from secure to non-secure memory. The accuracy is computed as the percentage of error bits while transferring the test data, and the data rate is computed as the total time required for transferring the test data. After decoding the data, we observed a bit error rate of 5.02 % and a data rate of 1245 Bytes/s.

Applicability to Other Prefetchers. Stride- and stream-based covert channels encode data in strides. Existing approaches [7] use two different stride widths to represent 0 or 1; i.e., 1-bit information per prefetcher entry. In comparison, the region-based prefetcher allowed us to encode more bits and achieve a significantly higher data rate, ten times higher than existing approaches. Furthermore, using FetchBench's insights into prefetcher characteristics allowed us to minimize the error rate and improve the data rate. We believe that our new insights into various prefetcher types can also improve related covert channels. For example, one could use FetchBench to characterize a stride prefetcher and encode more bits per pattern by also considering larger stride lengths.

6 RELATED WORK

6.1 Prefechter Reverse-Engineering

Most existing work focused on the x86 architecture. Rohan et al. [28] reverse-engineered the Intel Kaby Lake stream prefetcher and find the stream table size and the direction of its entries, the trigger condition of the prefetcher, and if the prefetcher is shared between different cores of the processor. Chen et al. [7] reverse-engineered and characterized the Intel IP-stride prefetcher in the Haswell and Coffee Lake microarchitectures, while Xiao et al. [40] investigated the behavior of the Intel Comet Lake IP-stride prefetcher for small strides. CacheObserver [9] characterized the L2 Stream prefetcher and the adjacent-cache-line prefetcher on the Intel Whiskey and Coffee Lake microarchitectures. While the aforementioned works focus on data prefetching, Zhang et al. [43] reverse-engineered the instruction prefetcher on a number of Intel CPUs and find that it follows branch predictions from the branch target buffer (BTB).

Only a few publications investigated prefetchers on ARM architectures. Sanchez Vicarte et al. [29] presented an approach to test the existence of the Array-of-Pointers prefetcher for the Apple M1 processors and characterize it by reverse-engineering its parameters, i.e., the activation condition and the access memory regions. Plumber [17] is a framework to facilitate reverse-engineering hardware features by leveraging instruction and operand fuzzing and statistical analysis. The authors used Plumber to study the prefetcher of ARM Cortex-A53 to, e.g., find out its trigger condition and whether prefetching respects page boundaries.

In contrast to prior works, our prefetcher characterization approach covers a multitude of data prefetcher designs. We are the first to identify and characterize a replay-based prefetcher design, in particular, the SMS prefetcher on the ARM Cortex-A72 architecture. Our approach enables us to better evaluate the security of a high number of CPUs across the x86 and ARM architectures.

6.2 Prefetcher Exploitation

Prefetchers have been exploited to leak data from program memory either through side channels [7, 17, 29, 30, 32] or by constructing covert channels [7, 8, 28]. Side-channel attacks exploiting prefetching target either hardware-based or software-based prefetching.

Hardware-based Approaches. The Intel IP-based stride prefetcher is a common target in literature. Cronin and Yang [8] exploited it on a Skylake CPU to construct a bi-directional inter-process *covert channel* between two adversary-controlled processes running on the same core. Using their approach, the two processes communicate by periodically forcing the prefetcher to forget its learned stride patterns, thus causing timing variations in memory accesses.

Shin et al. [32] exploit the same prefetcher as a *side channel* to attack a constant-time Elliptic Curve Diffie-Hellman (ECDH) implementation from OpenSSL to leak its private key. The Intel IP-based stride prefetcher has also been exploited in AfterImage [7]. The attack showed how an adversary could mistrain the IP-stride prefetcher in another context and use it to leak critical data from userspace applications and kernel routines. The authors also used AfterImage to attack RSA and successfully recovered the RSA key. Also, Xiao et al. [40] exploit the same prefetcher to attack a lookup-table-based AES implementation. BunnyHop [43] demonstrates that the instruction prefetcher on Intel CPUs can be exploited to leak cryptographic keys by causing partial eviction of lookup tables or control flow inference from the prefetcher's behavior.

Recent works [29, 30] examine data memory-dependent prefetchers [42], such as the pointer-array prefetcher in Apple M1 processors [29]. This prefetcher is exploited to perform out-of-bounds reads and retrieve leaked pointers. The attack technique assumes the same memory space for the attacker and victim and relies on Prime+Probe to retrieve data. Additionally, they present three prefetching-based primitives on the ARM architecture that may leak the control flow of programs and secret data from memory.

Compared to these works, we are the first to exploit a replay-based data prefetcher design. Similar to AfterImage [7], we cross the boundary between the operating system and the trusted execution environment and show that the prefetcher's internal data structure can be exploited to leak information across these domains.

Software-based Approaches. Gruss et al. [14] showed an attack that exploits software prefetch instructions on Intel CPUs to infer if an inaccessible page is physically backed. Lipp et al. [23] showed that the software prefetch instruction on AMD additionally leaks the TLB state of the prefetched page. Guo et al. [16] used the x86 prefetch instruction `PREFETCHW` to leak cryptographic keys through the data-dependent access pattern of an application.

7 DISCUSSION

Limitations. Our approach enables us to identify and characterize prefetcher designs we implemented test cases for, but not prefetchers that operate or are triggered in a different way. However, both our taxonomy and framework are extensible and can be adapted to support further contemporary or future designs.

In addition to data prefetchers, modern CPUs also utilize instruction prefetchers that allow predictive fetching of instructions before they are requested for execution. Data and instruction prefetchers share a common goal but are fundamentally different in the way they operate internally. For example, instruction prefetchers can be coupled with the branch predictor [43]. Characterizing and exploiting these prefetchers is an orthogonal research problem [12].

Countermeasures. A straightforward approach to mitigate prefetcher-based attacks is to permanently disable prefetching, which is possible on most recent processors through model-specific registers. However, this might impact performance. Several other countermeasures provide trade-offs between security, performance overhead, and implementation effort. For example, prefetching could be disabled selectively for security-relevant code. While causing less overhead than disabling prefetching completely, this countermeasure requires modifications to both the kernel and compilers, which need to expose and utilize this functionality, respectively. CPU vendors could adapt the following countermeasures: (i) partitioning the prefetcher state across processes and/or security domains; (ii) implementing a special load instruction that does not impact the prefetcher's state, which can be used in security-relevant code; (iii) implementing an instruction that flushes the prefetcher's state, which can be triggered upon context/domain switches; (iv) storing complete instruction addresses to correlate load instructions based on the program counter. Most of these countermeasures also induce non-negligible performance overhead, require kernel modifications, and may be incomplete. A software-based countermeasure is *constant-time programming*, i.e., writing code with secret-independent resource usage. More precisely, developers should ensure that runtime, code access patterns and data access patterns do not depend on secret values [18]. This approach could mitigate our attack on the AES implementation.

8 CONCLUSION

Due to their proprietary nature, the types and characteristics of specific prefetcher implementations are not known to the public for many common CPU models. This hinders systematic assessment of security threats posed by prefetchers. In this work, we addressed the following question: *How can we systematically identify and characterize under-specified prefetchers in proprietary processors?* We proposed a taxonomy for prefetcher designs that differentiates based on prediction strategy and information source, leading to seven different types of prefetchers that can be expected to be in use in modern CPUs. For each, we provided a detailed description and cross-platform test cases, which are used by our FetchBench framework to identify and characterize implemented prefetchers. We implemented and used this framework to characterize 19 different ARM and x86-64 CPUs, and we found that all differed in the specific behavior of their prefetchers. Our analysis and framework also allowed identifying and characterizing the previously unknown replay-based SMS prefetcher on the ARM Cortex-A72 CPU. To illustrate how accurate knowledge of prefetcher characteristics impacts prefetcher security, we demonstrated two side-channel attacks using the Cortex-A72 prefetcher to leak secret information, even from the secure TrustZone into normal world.

REFERENCES

[1] Arm Ltd. 2016. ARM® Cortex®-A72 MPCore Processor Technical Reference Manual.

[2] Arm Ltd. 2023. TrustZone for Cortex-A. https://developer.arm.com/Processors/TrustZone%20for%20Cortex-A

[3] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526.

[4] Jean-Loup Baer and Tien-Fu Chen. 1991. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 176–186.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7.

[6] 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 *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara (US)). USENIX Association, 19.

[7] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2023. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Vancouver BC Canada, 16–32.

[8] Patrick Cronin and Chengmo Yang. 2019. A Fetching Tale: Covert Communication with the Hardware Prefetcher. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[9] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. 2022. Characterizing Prefetchers using CacheObserver. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 170–179.

[10] Electronic Frontier Foundation. 1998. Frequently Asked Questions (FAQ) About the Electronic Frontier Foundation's "DES Cracker" Machine. https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html

[11] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching*. Number 28 in Synthesis Lectures Computer Architecture. Morgan & Claypool.

[12] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *IEEE Symposium on Security and Privacy (S&P) 2023*. IEEE Computer Society.

[13] Marius Grannaes, Magnus Jahre, and Lasse Natvig. 2010. Multi-Level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers* (Pisa, Italy) *(HiPEAC'10)*. 247–261.

[14] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR.

In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 368–379.
[15] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA 2016)*.
[16] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2023. Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks. In *IEEE Symposium on Security and Privacy (S&P) 2022*. IEEE Computer Society.
[17] Ahmad Ibrahim, Hamed Nemati, Till Schlüter, Nils Ole Tippenhauer, and Christian Rossow. 2022. Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*.
[18] Intel Corp. 2022. 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] Intel Corp. 2023. Intel® 64 and IA-32 Architectures Optimization Reference Manual.
[20] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 252–263.
[21] Paul Kocher, Jann Horn, Anders Fogh, and 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*.
[22] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2363–2380.
[23] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 643–660.
[24] 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 *27th USENIX Security Symposium (USENIX Security 18)*.
[25] Kyle J. Nesbit and James E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 96–96.
[26] Colin Percival. 2005. Cache Missing for Fun and Profit. In *In Proc. of BSDCan 2005*.
[27] Rockchip Electronics Co., Ltd. 2021. Rockchip RK3399-T Datasheet. Revision 1.0.
[28] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse Engineering the Stream Prefetcher for Profit. In *IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2020, Genoa, Italy, September 7-11, 2020*. 682–687.
[29] 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 *IEEE Symposium on Security and Privacy (S&P) 2022*. IEEE Computer Society.
[30] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. 2021. Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. 347–360.
[31] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenstrom. 2000. Recency-Based TLB Preloading. In *Proceedings of 27th International Symposium on Computer Architecture (ISCA 2000)*. 117–127.
[32] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA.
[33] Alan Jay Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (Dec. 1978), 7–21.
[34] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-Temporal Memory Streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 69–80.
[35] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA.
[36] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (Jan. 2010), 37–71.
[37] Krishnaswamy Viswanathan. 2014. Disclosure of Hardware Prefetcher Control on Some Intel® Processors. Intel. https://web.archive.org/web/20201112034737/https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html
[38] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical Off-Chip Meta-Data for Temporal Memory Streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 79–90.
[39] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2005. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA 2005)*. 12.
[40] Chong Xiao, Ming Tang, and Sylvain Guilley. 2023. Exploiting the Microarchitectural Leakage of Prefetching Activities for Side-Channel Attacks. *Journal of Systems Architecture* (April 2023), 102877.
[41] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. 719–732.
[42] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 178–190.
[43] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. BunnyHop: Exploiting the Instruction Prefetcher. In *32nd USENIX Security Symposium (USENIX Security 23)*.

A PROCESSORS UNDER EVALUATION

We provide a complete list of the processors under evaluation in Table 2.

B EXPERIMENT SUCCESS RATES

In this section, we illustrate the success rates of our identification and characterization experiments from Section 4. Since most of our CPUs implement a stride prefetcher, we use the results of the stride prefetcher existence test in positive direction as a representative example.

Generally, we repeat each test case 40,000 times per CPU. Figure 12 visualizes the cache state of the cache lines in our testing memory region after the experiment. The brighter the color, the more often did we observe the respective cache line in cache after completing the experiment. At the beginning of the memory area, we usually observe a strong signal for the loads that we performed to train and trigger the prefetcher (marked with blue dots). For many CPUs, we observed prefetching (red dots) almost as strong as the architectural loads without any false positives. For the A55, we see a long sequence of strong prefetch signals that finally fades out until the signal drops below the noise threshold (magenta dots). For the A72, most prefetches produce a strong signal. However, some prefetches have a significantly lower probability to occur. For the A73, we observe some very strong prefetches accompanied by some false positives at later cache lines. For the A76, we observe an overall lower signal for prefetching. For the Ryzen CPUs (R5Z, R5Z+, R7Z3, R9Z3+), we observe prefetching with a lower signal and some noise at later cache lines.

Table 2: List of hardware platforms under evaluation, prefetcher existence, and test runtime. For SoCs that combine multiple different cores in a single package we highlight the tested cores in boldface.

ID	System Information					Prefetcher Existence							Test Runtime (min)
	Vendor/Model	OS	Arch.	CPU/SoC	CPU/SoC Release	Stride	SMS	Adj. CL	Stream	R.-U. Replay	Ptr. Array	Ptr. Chase	
A53	Raspberry Pi 3	Raspberry Pi OS 11	ARMv8	Broadcom BCM2837 (Cortex-A53)	2016	●	○	○	○	○	○	○	376.0
A55	HardKernel Odroid C4	Ubuntu 20.04	ARMv8	Amlogic S905X3 (Cortex-A55)	2019	●	○	○	○	○	○	○	54.9
A72	Raspberry Pi 4	Raspberry Pi OS 11	ARMv8	Broadcom BCM2711 (Cortex-A72)	2019	●	●	○	○	○	○	○	78.4
A73	96Boards HiKey 960	Debian 9	ARMv8	HiSilicon Kirin 960 (Cortex-A53, -**A73**)	2016	●	●	○	○	○	○	○	79.7
A76	NanoPi R6S	Ubuntu 22.04	ARMv8	Rockchip RK3588S (Cortex-A55, -**A76**)	2021	●	●	○	○	○	○	○	56.6
M1i	Apple Mac Studio	Asahi Linux	ARMv8	Apple M1 Max *Icestorm core*	2021	●	○	○	○	○	○	○	269.8
M1f	— " —	— " —	— " —	*Firestorm core*	— " —	●	○	○	○	○	●	○	236.5
i7SB	HP EliteBook 2760p	Fedora 37	x86-64	Intel Core i7-2620M (Sandy Bridge)	2011	●	○	●	●	○	○	○	36.2
i5HW	Lenovo ThinkPad T440p	Debian 11	x86-64	Intel Core i5-4300M (Haswell)	2013	●	○	●	●	○	○	○	32.7
XeSL	Mini PC	Ubuntu 20.04	x86-64	Intel Xeon E3-1505Mv5 (Skylake)	2015	●	○	●	●	○	○	○	49.0
XeCL	Custom PC	Ubuntu 20.04	x86-64	Intel Xeon E-2176M (Coffee Lake)	2018	●	○	●	●	○	○	○	217.2
i7CL	Lenovo ThinkPad X1 Carbon Gen 8	Ubuntu 22.04	x86-64	Intel Core i7-10510U (Comet Lake)	2019	●	○	●	●	○	○	○	62.3
i3IL	Mini PC	Ubuntu 22.04	x86-64	Intel Core i3-1005G1 (Ice Lake)	2019	●	○	●	●	○	○	○	115.4
i7TL	Lenovo ThinkPad X1 Carbon Gen 9	Ubuntu 22.04	x86-64	Intel Core i7-1165G7 (Tiger Lake)	2020	●	○	●	●	○	○	○	47.7
XeIL	Custom PC	Ubuntu 22.04	x86-64	Intel Xeon Gold 6346 (Ice Lake)	2021	●	○	●	●	○	○	○	363.0
i9ALp	Custom PC	Ubuntu 22.04	x86-64	Intel Core i9-12900K (Alder Lake) *Perf. core*	2021	●	○	○	○	○	○	○	63.4
i9ALe	— " —	— " —	— " —	*Efficient core*	— " —	○	○	●	○	○	○	○	340.1
R5Z	Mini PC	Ubuntu 22.04	x86-64	AMD Ryzen 5 2500U (Zen)	2017	●	○	○	○	○	○	○	59.5
R5Z+	Mini PC	Ubuntu 22.04	x86-64	AMD Ryzen 5 3550H (Zen+)	2019	●	○	○	○	○	○	○	58.4
R7Z3	Mini PC	Ubuntu 22.04	x86-64	AMD Ryzen 7 5700G (Zen 3)	2021	●	○	○	○	○	○	○	40.4
R9Z3+	Mini PC	Ubuntu 22.04	x86-64	AMD Ryzen 9 6900HX (Zen 3+)	2022	●	○	○	○	○	○	○	27.4

● Prefetcher identified, ○ Prefetcher not identified

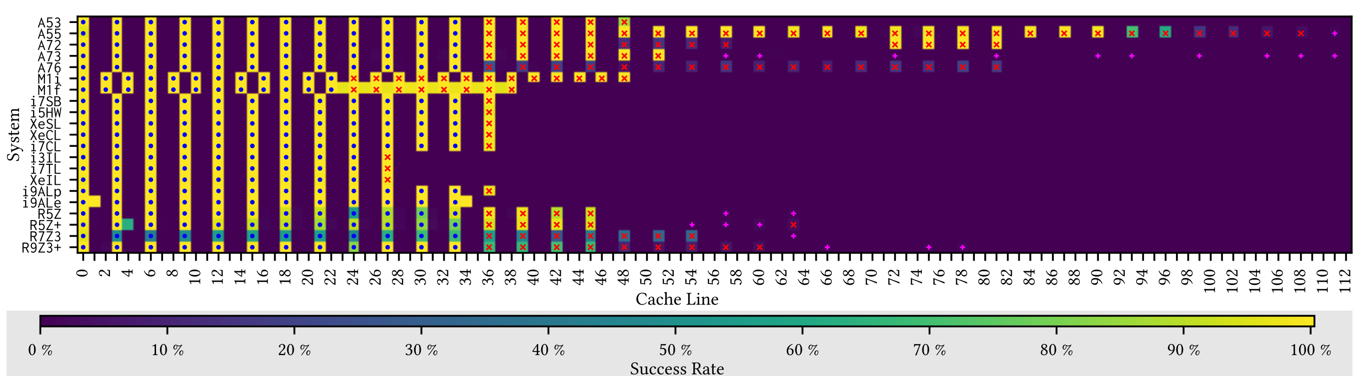

Figure 12: Visualization of experiment success rates. Blue dots indicate the locations of architectural loads that train and trigger the prefetcher. Red crosses (x) indicate potential prefetch locations that were frequently observed in cache. Magenta crosses (+) indicate potential prefetch locations that were observed only rarely (below the noise threshold).