A Scheduling-Aware Defense Against Prefetching-Based Side-Channel Attacks

Till Schlüter

CISPA Helmholtz Center for Information Security Saarbrücken, Germany till.schlueter@cispa.de

ABSTRACT

Modern computer processors use microarchitectural optimization mechanisms to improve performance. As a downside, such optimizations are prone to introducing side-channel vulnerabilities. Speculative loading of memory, called *prefetching*, is common in real-world CPUs and may cause such side-channel vulnerabilities: Prior work has shown that it can be exploited to bypass process isolation and leak secrets, such as keys used in RSA, AES, and ECDH implementations. However, to this date, no effective and efficient countermeasure has been presented that secures software on systems with affected prefetchers.

In this work, we answer the question: How can a process defend against prefetch-based side channels? We first systematize prefetching-based side-channel vulnerabilities presented in academic literature so far. Next, we design and implement PreFence, a scheduling-aware defense against these side channels that allows processes to disable the prefetcher temporarily during securitycritical operations. We implement our countermeasure for an x86 64 and an ARM processor; it can be adapted to any platform that allows to disable the prefetcher. We evaluate our defense and find that our solution reliably stops prefetch leakage. Our countermeasure causes negligible performance impact while no security-relevant code is executed, and its worst case performance is comparable to completely turning off the prefetcher. The expected average performance impact depends on the security-relevant code in the application and can be negligible as we demonstrate with a simple web server application.

We expect our countermeasure could widely be integrated in commodity OS, and even be extended to signal generally securityrelevant code to the kernel to allow coordinated application of countermeasures.

1 INTRODUCTION

Prefetching is an optimization mechanism of modern CPUs that aims to bring chunks of memory into the cache before they are actually loaded by application code. By bringing those chunks closer to the CPU in the memory hierarchy, applications benefit from lower memory latency. There are two kinds of prefetching: software prefetching and hardware prefetching. Software prefetching relies on explicit hints issued by application software to indicate which memory locations are likely to be accessed in the near future. In contrast, hardware prefetching is an automatic and fully transparent mechanism that analyzes memory accesses at runtime, tries to detect regular or recurring patterns and tries to predict memory locations that are likely to be accessed soon.

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

Most hardware prefetchers keep an internal state that controls the prefetching process. After the prefetcher has observed secretdependent memory accesses, its state may correlate with those secrets. Prior work has shown that the prefetcher's subsequent behavior can then be exploited as a side channel to compromise Diffie-Hellman keys [7, 37], RSA private keys [7, 9], or AES symmetric keys [35, 45]. In addition, covert channels based on hardware prefetching have been presented [9, 12, 31, 35], in some cases bypassing process isolation guarantees. A covert channel is a hidden communication channel between a sender and a receiver, both controlled by an attacker. In contrast, in a side-channel attack, only the receiving end is controlled by the attacker, while a victim process (involuntarily) acts as the sender. No defense has been presented to date that protects against prefetcher-based attacks effectively and and efficiently. For example, while most platforms allow to disable the prefetcher completely, this will have significant performance impact on non-security relevant parts of the applications, and all parallel applications that share the prefetcher.

In this paper, we propose Prefence, our novel countermeasure that allows an application to defend itself against the perils of hardware prefetching with minimal overhead. More precisely, our solution comes with negligible performance overhead on non-security-critical workloads. For security-critical workloads, the worst-case performance impact is comparable to disabling the prefetcher completely, but can also be negligible overall, depending on the protected workload and the way our countermeasure is applied.

We systematically analyze existing side-channel attacks that exploit hardware prefetching for their differences and similarities and we identify suitable entry points for defenses. Based on these insights, we design, implement and evaluate Prefence: a mechanism that allows applications to disable the prefetcher temporarily during security-critical operations. We also address the challenges arising from process scheduling and related to multi-core processing and Simultaneous Multithreading (SMT). As a software-based mitigation, Prefence leverages the widespread support of processors to disable the prefetcher and does not require further hardware adaptations. We focus on defending processes against falling victim to side-channel attacks based on hardware prefetching, as those attacks directly expose secrets from the victim's context; we exclude covert channels, as those can merely be used to transfer information that is already accessible to the attacker.

Contributions. We summarize our contributions as follows:

 We systematize existing prefetch-based side channel attacks and identify their similarities and differences. We identify 5 main stages and map each attack's flow to those stages, demonstrating that there are core components required by all attacks.

- We design, implement and evaluate PREFENCE, our approach
 to mitigate prefetch-based side channels in the scheduler.
 We demonstrate the performance impact is negligible, and
 that it prevents a prior work attack.
- We review software-based mitigations proposed in prior work and argue why they are either incomplete (e.g., do not consider SMT) or too costly (e.g., permanently disable the prefetcher, rewriting code as constant time). Prefence fills this gap.

We provide an open-source implementation of Prefence at https://github.com/scy-phy/Prefence/tree/preprint.

2 BACKGROUND

2.1 Caches and Prefetching

Caches. Modern processors aim to reduce the effective latency of memory accesses by maintaining *caches*. A cache is a fast and small temporary storage that stores frequently or recently used chunks of memory. These chunks are called cache lines and have a fixed size. When a program loads data from a memory address, the processor first checks whether the data is present in a cache (*cache hit*) or not (*cache miss*). In case of a hit, the load is significantly faster. Otherwise, the data needs to be fetched from DRAM, which takes more time.

Prefetching. Apart from chunks of memory that have been used in the past, modern processors may also bring chunks of memory into the cache that are likely to be accessed in the near future. To this end, a hardware unit of the processor, the prefetcher, observes memory accesses at runtime and predicts addresses that are likely to be accessed next. Those predictions are often generated by undocumented prediction mechanisms [35]. While most prefetchers only analyze addresses to generate predictions, more powerful data memory-dependent prefetchers (DMPs) also take the memory contents into account [7, 34]. Prefetching is a completely transparent mechanism from the application's point of view. To make useful predictions, most prefetchers keep an internal state that reflects recent memory activity. They are often implemented as a shared resource between processes running on the same physical processor core. These properties make prefetchers susceptible to side-channel vulnerabilities, as we discuss in detail in Section 4.

2.2 Simultaneous Multithreading (SMT)

Traditionally, every processor core executes exactly one program thread at a time. On a system that supports multithreading, multiple threads take turns in using the core. To switch from one thread to another, the state of the current thread needs to be stored in memory and the state of the next thread needs to be restored to the processor registers. This procedure is known as *context switching* and handled by the scheduler, a component of the operating system [40].

Simultaneous Multithreading (SMT) [41] is a concept that aims to better utilize the resources of a processor core. The idea behind SMT is to schedule multiple threads on a single processor core *at the same time*, based on the insight that a single thread is often not able to utilize all the processing units that a processor core has available.

SMT has been adopted by major processor vendors such as Intel (branded "HyperThreading") [23] and AMD [11]. These practical

implementations expose one physical processor core as multiple (often two) independent logical cores to the operating system. We refer to logical cores that are backed by the same physical core as *sibling cores*. The operating system schedules threads on logical processors in the same way as it would on a traditional system [23].

On a non-SMT system, a thread has exclusive access to the resources of a processor core while it is scheduled. In contrast, on an SMT-enabled system, the instructions issued by parallel threads scheduled on sibling cores share processor resources at the same time, potentially also the prefetcher. We emphasize that, as a result of SMT, instructions issued by multiple processes can be executed on the same physical core without requiring a context switch.

3 DEFENDING AGAINST PREFETCHING-BASED SIDE-CHANNEL ATTACKS

3.1 System and Attacker Model

We assume a system with a processor that performs prefetching. We further assume that the defender and attacker know the type of the deployed prefetcher, as well as its security-relevant characteristics (e.g. obtained by the attacker with a copy of the target hardware and a suitable testbench [35]). The defender is able to modify the software running on the CPU, including the operating system kernel. The hardware provides an interface to control (i.e., enable or disable) the prefetcher from the kernel. The attacker is able to execute arbitrary code in userspace. In Section 8.1, we extend the attacker model to attackers at higher privilege levels.

3.2 Research Questions and Challenges

In this work, we answer the following research questions:

- **RQ1:** What kind of side-channel vulnerabilities in prefetchers have been exploited in prior work? Is there a core set of vulnerabilities that are critical for all known attacks?
- **RQ2:** Is there a software-only countermeasure to mitigate all known prefetching-based side-channel vulnerabilities effectively and efficiently?
- **RQ3:** How can prior work on countermeasures be systematized, and which attacks can be expected to be prevented by those defenses?

Challenges. To answer these research questions, we need to overcome the following challenges:

- Prefetcher side channels have been exploited in different settings in prior work. We need to work out similarities and differences between those approaches to be able to identify common patterns.
- (2) Any countermeasure will cause a performance impact, which will need to be quantified and minimized.
- (3) Countermeasures require trustworthy arguments on why they can be expected to prevent current and future attacks. So far, such arguments have not been provided in prior work and across different attacks.

Proposed Approach. To overcome these challenges, we pursue the following approach. First, we systematize known prefetching-based side-channel attacks and their exploited vulnerabilities. We identify a minimal set of vulnerabilities that are required for any attack to work. Second, we design and implement a solution to prevent exploitation of this minimal set of vulnerabilities, leading

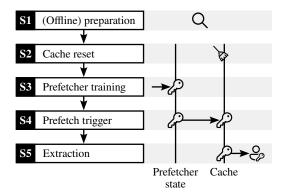


Figure 1: Stages of prefetching-based side channels

to a countermeasure effective against all prefetcher side-channel attacks from userspace. We then evaluate the implemented solution on real-world hardware. Finally, we review software-based mitigations from prior work and discuss their efficacy and efficiency.

4 SYSTEMATIZATION OF ATTACKS

To protect against prefetching-based side channels, we first need to understand the attack vectors in detail. To this end, we systematize all attacks exploiting hardware-based data prefetchers that we could find in academic literature (13 attacks across 7 papers). We list them in Table 1. Inspired by prior works on mitigating other microarchitectural side channels [5, 25, 36], we break down prefetcher-based attacks into stages. We further define the relevant scopes that those attacks operate in and report them in Table 1. Finally, we visualize our systematization by plotting the attack sequences in Figure 2, deduce similarities and differences, and expose where software-based mitigations can effectively be applied.

4.1 Stages of Prefetching Side Channels

We observe that prefetching side channels can be split into the following five stages, as illustrated in Figure 1:

- S1: (Offline) Preparation. For some attacks, an attacker has to take preliminary steps before the actual attack begins, such as reverse engineering or setting up data structures.
- **S2:** Cache Reset. To start from a clean cache state, some attackers perform actions that reset the initial cache state.
- S3: Prefetcher Training. Most prefetchers keep an internal state that determines their behavior. The prefetcher observes memory accesses at runtime and tries to identify patterns in the addresses or data being accessed. If a pattern is detected or a previously detected pattern is continued or interrupted, the prefetcher's internal state may be altered to change the prefetcher's future behavior. We refer to this state change as prefetcher training throughout this paper. From the attacker's perspective, this step can be seen as encoding information into the prefetcher's state. In the context of an attack, this information is secret-dependent.
- S4: Prefetch Trigger. Upon a trigger event, such as another memory access that matches certain criteria, a prefetcher may bring additional memory lines into the cache. Those memory lines are selected based on the prefetcher's internal

Table 1: Overview of prefetching-based attacks in prior work

Attack	Prefetcher	Scope	Target
Shin et al. [37]	Intel IP stride	CP	OpenSSL ECDH
Augury [34] OOB	Apple DMP	SP	Custom
Augury [34] SLH	Apple DMP	SP	Custom
Augury [34] Addr.	Apple DMP	SP	_
AfterImage [9] Var. 1	Intel IP stride	CT/CP	Custom
AfterImage [9] Var. 2	Intel IP stride	KU	Custom
AfterImage [9] SGX	Intel IP stride	TO	Custom
AfterImage [9] RSA	Intel IP stride	CT	MbedTLS RSA
AfterImage [9] Sync	Intel IP stride	CP	OpenSSL RSA
Xiao et al. [45]	Intel IP stride	SP	AES
FetchBench [35] AES	ARM SMS	CP	MbedTLS AES
PrefetchX [8]	Intel XPT	CP	MbedTLS RSA, GnuPG RSA
GoFetch [7]	Apple DMP	CP	Go RSA, OpenSSL DHKE,
			CRYSTALS

state. We refer to this process as *prefetch trigger* throughout this paper. From the attacker's perspective, this step can be seen as *extracting information from the prefetcher's state into the cache.*

S5: Extraction. The cache state is inspected to extract information about the prefetcher's internal state. This step can be seen as *extracting information from the cache into the attacker's context*.

Attacks may skip some of these stages (see Figure 2).

4.2 Scopes

Prefetch attacks often operate across privilege domains. In this respect, we classify attacks based on the following scopes:

- **SP: Same-process.** Leaking within the same process.
- **CT:** Cross-thread. Leaking from one thread of a userspace process to another.
- **CP:** Cross-process. Leaking from userspace process to another.
- KU: Kernel to user. Leaking from kernel to userspace.
- **TO: TEE to OS.** Leaking from a trusted execution environment (TEE), such as Intel SGX or ARM TrustZone, to the (untrusted) operating system.

If victim and attacker do not share the same context (e.g., they are different userspace processes), the attacker needs to make sure that the prefetcher keeps its state across the context switch. Especially when leaking between two userspace threads or processes, the attacker faces the problem that the scheduler manages the process runtime, making it non-trivial to interrupt the victim process at a specific point in time (when the prefetcher's state is secret-dependent) and schedule the attacker process (to extract the state). Some attacks assume shared memory, either data memory or shared libraries, between both processes to address this issue [9], others use additional side channels for synchronization [35].

4.3 Prefetching Attack Systematization

We now map 13 attacks from prior work to the five stages introduced above (details in Appendix A) and answer RQ1. We summarize the results in Figure 2, showing the sequence of activities per attack. If an attack does not perform any of the alternative activities at some point in the sequence, we record a transition through a *no-operation (NOP)* block.

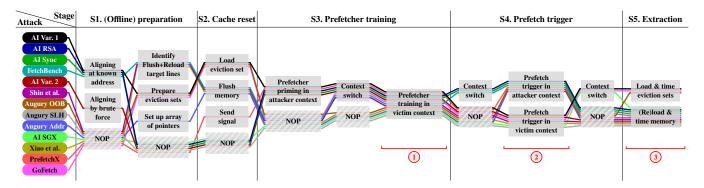


Figure 2: Overview of the sequence of activities in prefetching-based attacks. Core activities required by all highlighted in red.

Finding: There Are Mandatory Stages. Our systematization shows that prefetcher training ①, prefetcher triggering ②, and cache extraction ③ (highlighted in red in Figure 2) are activities that are common to all attacks, as there are no *NOP* alternatives.

Finding: Prefetch Attacks Are Cache Attacks. We further note that all prefetcher-based side channels are cache-based attacks: All attacks are built upon techniques to probe the cache state of certain cache lines ③, typically akin to Flush+Reload [46] and Prime+Probe [26].

Finding: Prefetching Is Triggered by Victim or Attacker. We identify that many attacks rely on the victim context to trigger the prefetcher to extract information from the prefetcher's state and transfer it into the cache (2). However, recent works have shown that this step can be moved into the attacker's context in some cases, even though this involves more complex synchronization steps [8, 9, 35].

Finding: Victim Trains The Prefetcher. Most importantly, we emphasize that all attacks rely on prefetcher training within the victim context ①. This is plausible, as the victim necessarily needs to work with the secret (e.g., perform secret-dependent memory accesses) to encode it into the prefetcher's state. Notably, from a defender's perspective, this means that the victim is able to protect itself using mitigations applied to its own code.

We conclude that a general mitigation approach against prefetch-based side-channel attacks is to *ensure that no training occurs in the victim context*. Fortunately, code running in victim contexts is easier to control than attacker code, which is (as the name suggests) constituted by the attacker.

5 PREFENCE: DESIGN AND IMPLEMENTATION

We now answer RQ2 by presenting PREFENCE, our software-only countermeasure against prefetching-based side-channel attacks. It exploits that the victim process trains the prefetcher in all attacks. We discuss alternative software-based defense approaches and why we consider them infeasible in Section 7.

 $\bf Design~Goals.$ The main design goals of PreFence are:

DG1: It mitigates all prior prefetch-based attacks conducted from userspace

DG2: It is simple to use for application developers and end users

DG3: It has minimal runtime overhead

DG4: It is still functional when the prefetcher is shared across physical or SMT sibling cores

We show that Prefence achieves these goals in the following sections.

Approach: Disabling the Prefetcher Temporarily. We take the approach of disabling the prefetcher temporarily, for example while security-critical code is executed. As we have shown in Section 4.3, stage S3 is the critical phase in all attacks. In this stage, secrets are encoded into the prefetcher's state. To achieve DG1, we exploit (and verify in Section 6.2) that the prefetcher does not update its state while it is disabled, i.e., it cannot be trained in this state. By disabling the prefetcher temporarily while secrets are processed, we ensure that the prefetcher is not trained with secrets. Afterward, the prefetcher can be enabled again. This limits the performance impact of the missing prefetching mechanism to the security-critical code and thus enables DG3.

Challenges. While this countermeasure sounds straightforward to implement at first, some challenges become apparent on closer inspection. First, a security-critical process that requests the prefetcher to be disabled may be interrupted by the scheduler and replaced by another process. In that case, the prefetcher must be re-enabled while the other process is running and disabled again when the security-critical process is re-scheduled. This ensures (i) that the interrupting process can benefit from the prefetcher's performance boost, and (ii) that the interrupting process cannot maliciously re-enable the prefetcher. Second, a security-critical process may be migrated from one processor core to another. If the prefetcher operates on a per-core basis, it needs to be re-enabled on the original core and disabled on the destination core. Third, if the prefetcher is shared across physical or SMT sibling cores, a malicious process that runs concurrently on another core could re-enable the prefetcher while the security-critical process is running. Consequently, the prefetcher must not be enabled if any of the processes sharing the same prefetcher requested it to be disabled. Lastly, access to MSRs that control the prefetcher is usually not allowed from userspace.

5.1 Design

We conclude from the above challenges that our countermeasure needs to be tightly integrated with the operating system kernel and the scheduler. In that way, relevant registers can be accessed and it can be ensured that the prefetcher is disabled as long as a program requests it, regardless of intermittent scheduling events.

Victim-Initiated Prefetch Control. Our countermeasure empowers a userspace process to protect itself from prefetching-based side channel attacks by requesting the prefetcher to be disabled temporarily. More precisely, a process signals to the operating system kernel when it enters or leaves a security-critical code section, such as an encryption function. While this requires developers to add small amounts of code, the required changes are minimal and the effort is low compared to complex re-writes required by other countermeasures (such as constant-time programming, see Section 7). In this way, Prefence achieves DG2. The kernel keeps track whether a process is currently in a security-critical code section and ensures that the prefetcher is disabled during this period—even if it is interrupted by the scheduler. After disabling the prefetcher, it remains disabled until the requesting process is finished with executing its security-relevant code, or a different process without securityrelevant code is scheduled into. If the next-to-be-scheduled process also requested to have the prefetcher disabled, the prefetcher remains disabled until non-critical code is reached. This is enforced by the scheduler, which changes the prefetcher's state based on the request of the next process to be executed.

The Simple Case: Per-Core Prefetcher, No SMT. We illustrate our countermeasure by example of Figure 3 (a), starting with the simple case in the upper half: a prefetcher that is exclusive to one core on a CPU without SMT. When a process is started, prefetching is enabled by default. As process P1 shows, the process can then request prefetching to be disabled, perform a security-critical operation, and request prefetching to be enabled again. Process P2 illustrates the case where a process is interrupted by the scheduler during a security-critical code section. The scheduler deschedules P2 and checks whether the next process requested prefetching to be disabled or not. In this example, the next processes (the new process P3) did not request prefetching to be disabled. Consequently, the scheduler re-enables prefetching while P3 is running. Once P3 is finished, the scheduler disables the prefetcher again and switches back to P2.

The Special Case: Shared Prefetcher or SMT. If the processor uses SMT and sibling cores share the same prefetcher, we need to be careful about implicit context switches in order to fulfill DG4. With SMT, multiple processes can be executed on the same physical processor core simultaneously without operating-system-controlled context switches between them. In the worst case, when an attacker and a victim process are scheduled on sibling cores, the victim could request the prefetcher to be disabled, while the attacker requests it to be enabled again. Similarly, if the prefetcher is shared across cores, an attack could be run simultaneously from a different core. For this reason, it is not enough to update the prefetcher's state on OS-controlled context switches in these cases. Instead, we keep prefetching disabled on all (logical) cores that share a prefetcher as soon as and as long as any of the scheduled processes requests it.

We provide an example in the lower half of Figure 3. First, process P1 requests prefetching to be disabled while the parallel process P4 does not request it. To protect process P1, prefetching is disabled until P1 leaves the security-critical code section. Next, process P5 enters a security-critical section, but is soon interrupted by the

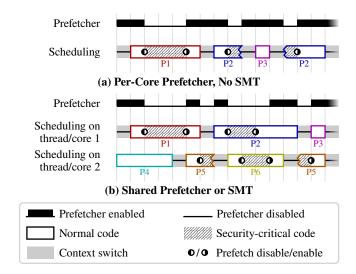


Figure 3: Prefence at work: The prefetcher is disabled temporarily while security-critical code is executed. On SMT-capable cores and for shared prefetchers, the scheduler considers requested states of the relevant parallel processes.

scheduler. Since no more processes request the prefetcher to be disabled after P5 is descheduled, the prefetcher is enabled again. Next, processes P2 and P6 both run security-critical code in parallel. The prefetcher is disabled when the first process (P2) enters the security-critical section and is re-enabled when the last process (P6) leaves it. Finally, P5 is re-scheduled. As it was interrupted in a security-critical section, the prefetcher is disabled at context switch and re-enabled once the security-critical section is completed.

Core Migrations. We note that our methodology also handles the case of a core migration. On a context switch, the scheduler adjusts the prefetcher's state based on the request of the next process. When a core migration occurs, the prefetcher's state on the original core is determined by the next process running on that core. On the target core, the migrating process is the next process, so it decides about the prefetcher's state.

5.2 Implementation

PREFENCE Kernel Patch. We implement our countermeasure as a Linux kernel patch. Our prototype is currently able to control the prefetchers of Intel x86_64 (tested on Comet Lake) and ARM Cortex-A72 CPUs. Excluding comments, our patch adds only 91 (Intel) / 62 (ARM) lines of code to the Linux kernel code base. We extend the task_struct, the place where the scheduler keeps all information related to a process, with a boolean prefetch_disable flag. The flag is initialized to false for new processes, i.e., prefetching is enabled by default and can be disabled on request.

To allow processes to control this flag from userspace, we add options to set, clear or query the flag to the prctl system call. When the flag is changed through the system call, the kernel updates the task_struct of the calling process accordingly. In addition, the kernel changes the prefetcher's state on the respective CPU immediately by writing to the corresponding MSRs [43] before returning to userspace. We further extend the scheduler's context_switch

function to update the state of the prefetcher on context switches based on the prefetch-disable flag of the next process.

To deal with prefetchers shared across physical or SMT sibling cores, we keep a global bit vector (of type cpumask_t) that indicates for each CPU whether it currently runs a process with the prefetch-disable flag set. We check this bit vector before enabling the prefetcher on any core. Only if none of the cores sharing the same prefetcher currently runs a process with the prefetch-disable flag set, the prefetcher can be enabled; otherwise, it remains disabled.

Using Prefence in Applications. To make use of Prefence, the prefetch_disable flag needs to be set before entering security-critical code sections and cleared afterward (through system calls). The countermeasure can be applied at various levels. For instance, library developers can protect security-critical code sections, for example those that process secrets. This ensures that the prefetcher is disabled only for a minimal period of time. Application developers who use unprotected legacy libraries can resort to setting the bit themselves before performing security-critical library calls and clearing it afterward. In this case, the prefetcher is disabled for a longer but still limited period of time.

PREFENCE can even be made available to knowledgeable end users who want to protect legacy software. A wrapper program similar to taskset can set the prefetch_disable flag in advance and then execute a target application. The target application inherits the flag and is thus executed with prefetching disabled. This means that the target application cannot benefit from prefetching at all. However, the impact on the overall system performance is still limited to one application, and other applications can still benefit from prefetching.

6 EVALUATION

In this section, we evaluate our Prefence implementation for efficacy and efficiency. First, as a prerequisite, we evaluate the behavior of a disabled prefetcher to verify that disabling the prefetcher has the expected effects (no further training), and thus Prefence is applicable. Next, we demonstrate the efficacy of our countermeasure. It prevents the prefetching-based side channel presented by Shin et al. [37] (reproduced by us). Finally, we show that Prefence is also efficient. We investigate this in three scenarios:

- Scenario 1: Stock kernel. For an unmodified ("stock") kernel, we measure the performance impact when the prefetcher is disabled for the whole execution time of an application (using SPEC benchmarks). These measurements serve as a baseline for the following experiments.
- Scenario 2: Patched kernel with non-critical workload. For a patched kernel, we measure the performance impact (on SPEC benchmarks) when no process makes use of the possibility to disable prefetching. These measurements show the fixed performance overhead on a process introduced by our countermeasure.
- Scenario 3: Patched kernel with critical workload. As an end-to-end example, we evaluate the performance of a web server application running on a patched kernel. Using our syscall, we disable the prefetcher during the TLS-related code execution. In addition to cryptographic code, each

HTTP request to our server also triggers application code which can still benefit from the prefetcher (e.g., a database lookup). We test different values for the complexity of the database lookup to find at which point the performance impact of disabling the prefetcher during TLS parts becomes negligible.

To investigate the overhead further, we specifically evaluate the introduced fixed overhead on every context switch and the one-off overhead of a system call whenever a prefetch_disable flag is modified in isolation. These additional results can be found in Appendix B.

6.1 Evaluation Environments

For all experiments in the main body of this paper, we use the following two platforms throughout the evaluation.

x86_64. Our x86_64 platform is an Intel Core i7-10510U (Comet Lake) CPU running Alpine Linux 3.19 with kernel 6.6.14-r0-lts. Depending on the experiment, we either use the original kernel from the Alpine repositories or our patched kernel derived from it. We use the rdtscp instruction to measure time.

ARM. Our ARM platform is a Raspberry Pi 4 using a Broadcom BCM2711 SoC with four Cortex-A72 cores. It runs Raspberry Pi OS 12 64-bit with Linux kernel 6.6.22-v8. We either use a kernel that we compiled from the official sources [30] without any changes or a kernel derived from it using our kernel patch. We use the cycle count register (PMCCNTR_EL0) to measure time.

6.2 Prerequisite: Disabled Prefetcher Behavior

Prefence requires that a disabled prefetcher cannot be trained, i.e., it does not update its state while it is disabled.

Experiment. To test the prefetcher for this behavior, we implement a corresponding testcase for stride prefetchers, the most common type of prefetchers, in the FetchBench framework [35]. We first disable the prefetcher, access a sequence of memory locations with constant distance between them, re-enable the prefetcher, and perform one more memory access matching the pattern. If the prefetcher keeps learning while disabled, we expect it to be triggered by that last access and bring more elements into the cache. Otherwise, no prefetching effects should appear in cache. As a baseline, we repeat the same experiment with the prefetcher being enabled. In that case, we expect prefetching effects in the cache.

Results. We run the testcase on the Intel Core i7-10510U and BCM2711 processors. As illustrated in Figure 4, we find that the prefetchers cannot be trained while disabled. We conclude that PREFENCE is applicable to these prefetcher implementations.

6.3 Efficacy: Protecting OpenSSL

In this experiment, we evaluate the efficacy of our countermeasure using the attack by Shin et al. [37] on the ECDH implementation in OpenSSL 1.1.0g as an example. Instead of re-implementing the end-to-end attack, we focus on reproducing the underlying prefetching side channel in both evaluation environments and show that PREFENCE prevents the leakage successfully.

Vulnerability. The leakage is caused by memory accesses to a lookup table when a point on an elliptic curve is squared. If those

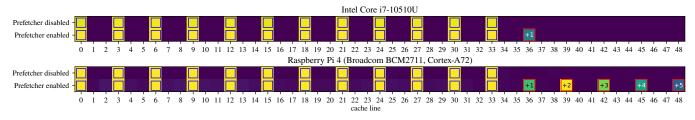
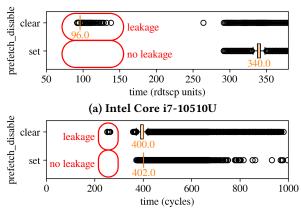


Figure 4: Prefetcher behavior when trained while disabled (compared to the behavior when trained while enabled). Blue boxes denote accesses, red boxes indicate prefetch locations. The tested prefetchers cannot be trained while it is disabled.



(b) Raspberry Pi 4 (Broadcom BCM2711, Cortex-A72)

Figure 5: Latency of accessing the prefetch location after calling the vulnerable OpenSSL function with the PREFENCE countermeasure not applied (prefetch_disable flag cleared) and applied (flag set). Short access latency indicates unwanted leakage, which is prevented by activating our countermeasure.

accesses (by chance) form a regular pattern, the prefetcher is activated and fetches memory lines before and/or after the lookup table. This leaves traces in the cache state of shared memory, leaking relations between different portions of the point on the curve. Depending on the context where this operation is used, the point may be secret information.

Experiment. We identify the OpenSSL library function BN_-GF2m_mod_sqr_arr as the function that operates on the lookup table. Our test program calls this function with a value that produces a regular access pattern and thus triggers the prefetcher (if enabled). It then accesses the potentially prefetched location, in our case the first cache line after the lookup table, and measures the memory latency to determine its cache state.

We repeat the experiment in two configurations. In the first configuration, we call the function without any countermeasure against prefetching-based side channels enabled. This experiment serves as a baseline and shows that the library function actually leaks information when called with specific inputs. In the second configuration, we set the prefetch-disable flag before calling the library function and clear it after returning from the library function. If PREFENCE is effective, we expect no more prefetching leakage.

Results. We run both configurations in both evaluation environments and present the results in Figure 5. We repeat each configuration 1,000,000 times on the Intel CPU and 10,000,000 times on the ARM CPU. When the prefetch_disable flag is cleared on the Intel CPU, we observe a significantly lower latency when loading from the memory line right after the lookup table (median: 96 units). This indicates that the prefetcher loaded this memory line into the cache (i.e. unwanted leakage). In contrast, when PreFence is activated on the Intel CPU by setting the prefetch_disable flag, the observed memory latency is above typical values for cache hits (median: 340), indicating a cache miss and absence of leakage. On the ARM CPU, we observe a weaker leakage signal (possibly indicating that the prior work attack would not perform as well here). Without a countermeasure, the prefetching leakage is visible in the form of outliers appearing at around 250 cycles. When we set the prefetch_disable flag before calling the target function, we observe that these outliers disappear reliably. We conclude that Preferce successfully prevents the prefetch-based side channel in both environments.

6.4 Efficiency: Non-critical Workloads (Scenarios 1 and 2)

We now investigate the efficiency of Prefence, starting with the performance impact on workloads that are not security-critical.

Experiment. We run SPEC CPU 2017 benchmarks [39] on three different system configurations. As a baseline, we measure the performance of the SPEC workloads on a stock kernel while prefetching is either enabled or disabled permanently (scenario 1). These measurements show us how much different workloads benefit from prefetching at all, and how expensive the radical-but-simple defense of disabling the prefetcher permanently would be. Afterward, we measure the performance of the same workloads on a patched kernel with prefetching enabled, and without setting the prefetch_disable flag (scenario 2). This allows us to rate the performance impact on non-security-critical workloads caused by the added code that is executed on every context switch.

Benchmark Parameters. We run the *SPEC CPU 2017 Integer Rate* set of benchmarks and report the execution time of the individual benchmarks as a metric for their performance. We run each benchmark three times (which is the maximum number of iterations in a "reportable run" [38]).

Results. Figure 6 shows the benchmark results in both evaluation environments. The bars represent the median runtime of the individual benchmarks across the three iterations, while the black error bars indicate the runtime of the other two iterations.

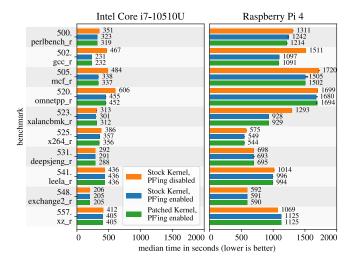


Figure 6: SPEC CPU 2017 benchmark results. Disabling the prefetcher permanently causes significant performance overhead in benchmarks 502 to 523. The performance overhead introduced by our patched kernel is negligible for non-security-critical workloads.

Comparing the two stock kernel configurations (orange and blue bars), we find that the prefetcher especially speeds up the benchmarks 502–523. At a maximum, the prefetcher improves performance by 43 % (benchmark 505 on the Intel CPU) and 37 % (benchmark 502 on the Raspberry Pi), respectively. In most other workloads, both configurations performed similarly. In one exceptional case, we see a slowdown by 5 % caused by the prefetcher (557 on the Raspberry Pi). We conclude that disabling the prefetcher permanently can lead to a significant performance drop on both tested systems.

When we compare the stock kernel and the patched kernel, both with prefetching enabled (blue and green bars), we observe only small differences in execution time. For most benchmarks, the absolute difference is around 1 %. We conclude that our kernel patch has negligible impact on non-critical workloads.

6.5 Efficiency: Security-critical Workloads (Scenario 3)

Next, we evaluate the performance impact of Prefence on a security-critical workload that uses our protection mechanism.

Experiment. To evaluate the efficiency of Prefence in a realistic end-to-end scenario, we now apply it to real-world software. We use the web server *lighttpd 1.4.75* [19] (released in March 2024) as an example. Lighttpd ships with plugins for various cryptographic libraries that can be used as backends to provide HTTPS support. In the following experiments, we use the OpenSSL plugin for this purpose. We compare two approaches of applying Prefence: on plugin level and on application level.

PREFENCE on Plugin Level. In this approach, we modify the OpenSSL plugin of lighttpd such that the prefetch_disable flag is set whenever the control flow enters any function in the plugin code (which then calls OpenSSL), and cleared before the control flow

returns from the plugin code. This approach allows the majority of the web server code base to benefit from prefetching, but causes frequent system calls to enable or disable the prefetcher.

PREFENCE on Application Level. In this approach, we set the prefetch_disable flag when lighttpd is started and clear it when it quits. In other words, the prefetcher is disabled for the whole lighttpd code base, including non-critical server code and any hosted web application that is executed in a child process. While this means that the server cannot benefit from prefetching at all, it also means that fewer system calls are required.

Web Application. In our example, we use lighttpd to host a web application that we expect to benefit from prefetching in a scalable way. Our example application is written in Python and attached to lighttpd via CGI. It accesses a table in an SQLite database filled with random numbers. We set up the table with 16 columns and a variable number of rows. Whenever the application is called, it randomly selects one of the columns and finds the minimum value in that column. Thus, the application needs to access every value in that column. We expect that this activity can benefit from prefetching, especially for tables with many rows.

Benchmarking Approach. We configure lighttpd to serve our web application via HTTPS over TLSv1.3. To measure the server's performance, we use the *httpit* HTTP(S) benchmark [13] over the local loopback interface. We configure httpit to perform one connection at a time and measure the average number of requests per second that the server is able to serve over a 5-minute time period as a performance metric.

To rate the performance of this setup when protected using PREFENCE, we measure it in scenario 3, i.e., with a patched kernel and prefetching temporarily disabled. For completeness and as a baseline to compare with, we also include performance measurements of the same setup in scenario 1 (stock kernel) and scenario 2 (patched kernel with prefetching permanently enabled).

Results: Plugin-level vs. Application-level Defense. We present the results of this experiment in Figure 7. The y-axis represents the performance of the web server relative to a baseline. This baseline is the performance of the web server in scenario 1 (stock kernel) with prefetching permanently disabled, indicated by the orange line in the plots.

We find that applying Prefence on plugin level (purple line) results in negligible performance impact of less than 1 % on average compared to those scenarios where prefetching is permanently enabled, either on the stock kernel or on the patched kernel (blue and green lines, respectively). Thus, the performance impact of Prefence is negligible in this case. In contrast, we find that applying Prefence to the whole web server (pink line) results in a performance roughly equal to the baseline, i.e., turning off prefetching permanently. Therefore, we conclude that Prefence is best used on a fine-grained level if possible, as this allows most parts of the application to still benefit from prefetching and keep a high performance level.

Results: Platform-specific Impact of SQL Query Complexity. On the x-axis of Figure 7, we show the number of table rows traversed by the web application. We find that enabling prefetching for the runtime of the web application always improves its performance, even if the table contains only a single row. Because the prefetcher is unlikely to speed up the actual database lookup in a

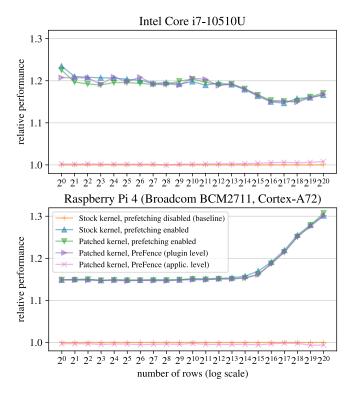


Figure 7: Lighttpd benchmark results. Applying PREFENCE on plugin level results in performance similar to permanently enabled prefetching. Applying PREFENCE on application level results in performance similar to permanently disabled prefetching.

table with only one row, we attribute the speedup in that case to beneficial prefetching activity in code that is not directly related to the database lookup (such as the Python interpreter or the CGI handling routines in the web server). We further note that the relative performance compared to the baseline increases for higher row counts on the Raspberry Pi. This indicates that also the actual database lookup benefits from prefetching on that platform, at least for higher row counts. However, this is not the case on the Intel CPU, which indicates that the prefetcher on that CPU generally does not speed up this workload. As pointed out by prior work, prefetching on the Raspberry Pi is more aggressive than on the Intel CPU [35], which likely explains this discrepancy between the two architectures.

6.6 Summary of Results

We have demonstrated that Prefence is applicable to our Intel and ARM processors, as their prefetchers cannot be trained while they are disabled (Section 6.2). We also verified that Prefence is effective, as it eliminates leakage caused by the prefetcher in a known-vulnerable library function on both architectures (Section 6.3). We further demonstrated that Prefence is efficient: For workloads that are not security-critical, such as SPEC benchmarks, we observed a performance overhead of less than 1 % for the majority of the benchmarks (Section 6.4). For our example of a security-critical

workload, a lighttpd web server hosting a web application that may benefit from prefetching, we found that the performance overhead is negligible (i.e., less than 1 % on average) if Prefence is applied on the level of the OpenSSL plugin. In this way, the prefetcher is only disabled when security-critical parts of the code are executed (Section 6.5).

7 PRIOR-WORK SOFTWARE-BASED COUNTERMEASURES

We now answer RQ3 by discussing prior work mitigations to prefetch-based attacks. We show that none of them provide an easy and efficient way to prevent attacks. We classify a countermeasure as *software-based* if it can be applied to at least one real-world CPU architecture via (modified) software.

7.1 Prior Work Countermeasures

Constant-Time Programming [9, 35, 37, 45]. One way to prevent most prefetching-based side channels-and also other cachetiming side channels—is the programming technique of constanttime programming. Despite the name, it refers not only to writing code that executes in the same time regardless of the (potentially secret) information processed, but also mandates that no secret-dependent control flow or memory access patterns occur [14]. Avoiding secret-dependent memory access patterns prevents address-based prefetchers from transferring secrets into their internal state during training. Moreover, avoiding secret-dependent branches prevents prefetcher-based attacks that infer the victim's control flow based on conditionally executed load instructions. However, a recent work has shown that constant-time programming is ineffective against DMP-based attacks that exploit secretdependent values instead of addresses or branches [7]. In addition, making code constant-time requires complex re-writes and results in significantly reduced performance [6, 28].

Clearing the Prefetcher's State on Context Switches [9, 12, 35]. Some prefetching-based attacks train the prefetcher in a victim context, then switch into the attacker's context and trigger it there. In practice, this context switch can be a switch between two userspace processes, from kernel to userspace, or a return from trusted execution. To mitigate such attacks, the prefetcher's state could be cleared on context switches. On Linux systems, switches between kernel and userspace and within userspace are handled by the scheduler. As processes cannot control when the scheduler interrupts (or resumes) them, clearing the prefetcher's state needs to be implemented in the scheduler. For context switches between trusted execution and operating system, the prefetcher could be cleared before returning control from the TEE back to the (untrusted) operating system. The process of clearing the prefetcher's state is straightforward to implement if the CPU provides a suitable instruction, such as CPP RCTX on some ARM CPUs [2, 3]. Otherwise, all patterns need to be evicted from the state and replaced with secret-independent ones. This process is non-trivial to implement, as it requires knowledge of implementation details such as the number of patterns stored and the replacement policy.

We note that clearing the prefetcher's state on context switches is an incomplete countermeasure in three cases. First, it is not applicable to attacks that trigger the prefetcher in the victim process.

Second, this countermeasure assumes that the prefetcher is not shared across physical or SMT sibling cores. Otherwise, the attack could be executed from a different core before the context switch resets the state. Third, in the case of trusted execution, prior work has shown that an attacker is able to interrupt trusted execution before it completes [18, 33, 42]. If this is possible, clearing the prefetcher's state only at the end of a trusted execution procedure is insufficient.

Mitigating Cache-Timing Side Channels [9, 37]. We found in Section 4.3 that all prefetching attacks rely on a cache-timing side channel. To mitigate those, access to timer interfaces can be restricted or their resolution can be reduced. This mitigation has especially been applied to browsers in the past [10, 44]. However, modern browsers provide many indirect ways to acquire timestamps [32] and also in other contexts, attackers may fall back to alternatives such as a counter thread as a timer replacement [20]. General countermeasures against cache-based side channels have been discussed in prior work extensively [21, 26, 46, 48], but none were implemented on a large scale. Consequently, we consider it next to impossible to reliably block an attacker from all possible ways to generate precise timestamps.

Anomaly Detection [9]. During a prefetcher-based side channel attack, the attacker may execute code that results in unusual amounts of cache flushes or evictions, leading to unusual amounts of cache hits and misses. In addition, if the attacker interacts a lot with the prefetcher in order to prime or trigger it, the amount of prefetch-related events may increase. If the CPU exposes performance counters for these events, those can be observed and evaluated to detect unusual activities [27, 47]. However, such a heuristic detection system will produce false-positive alerts, miss malicious events (false negatives), and introduce a constant runtime overhead affecting all workloads. In addition, intrusion detection systems generally do not prevent attacks, but merely detect them and react after the attack has started. Thus, we consider this mitigation strategy incomplete.

Security-Aware Core Assignment [7, 34]. If a prefetcher operates per-core, one possible mitigation is more advanced core assignment. For processors with a heterogeneous design, a vulnerable prefetcher may only be present on some of the cores. In this case, security-critical or untrusted workloads could be assigned to cores that are not vulnerable. Similarly, a vulnerable per-core prefetcher could be disabled on one of the cores. This core could then be reserved for critical workloads. However, it is not trivial in practice to decide which processes can be assigned to which core. In addition, reserving a core for critical operations is likely to reduce overall system performance significantly: If critical workloads are frequent or long-running, assigning them to a single core will limit their throughput. If critical workloads are rare or short-running, reserving one core for them will result in the core idling most of the time.

Oblivious Execution [9]. The idea behind oblivious execution [29] is to eliminate the side-channel effect of a secret-dependent if/else conditional by executing both branches and only persisting the result of the correct branch in memory in the end. In this way, the attacker is no longer able to distinguish both branches: the timing and the memory patterns that are executed are always the same (as long as the branches do not contain more secret-dependent

instructions). However, this approach obviously comes with a significant performance overhead. Rane et al. [29] report a significant mean overhead of 16.1×.

Blinding [7]. A DMP may be triggered by a data value that matches a specific pattern, e.g., that looks like a pointer. To ensure that such a prefetcher is not triggered by an untrusted value, a mask can be added to the value before it is stored in memory and removed after it is loaded again. However, implementing this countermeasure is not trivial and introduces memory and computational overhead.

Disabling the Prefetcher [7–9, 12, 35, 37]. If the CPU exposes a way to control the prefetcher, the most straightforward way to prevent any leakage from the prefetcher is to disable it permanently. Some CPUs expose such configuration options through model-specific registers (MSRs). However, this countermeasure comes with a significant performance decline for workloads that benefit from prefetching, as we show in Section 6.4.

While the general idea of disabling the prefetcher temporarily has been stated in literature before [7, 35], no detailed design, implementation, or evaluation has been provided to date. We fill this gap by presenting Prefence in this paper. Our novel design has low runtime overhead and handles the special case of prefetcher sharing across cores or SMT siblings by extending the process scheduler.

7.2 Conclusion: Prior-Work Countermeasures Are Costly or Incomplete

In summary, every prior-work countermeasure violates one of our design goals stated in Section 5. We consider constant-time programming complex to implement, expensive at runtime, and ineffective against DMP-based side channels; clearing the prefetcher's state on context switches specific to attacks that trigger the prefetcher in the attacker's context, expensive at runtime, and incomplete when using SMT; mitigation of timing sources and anomaly detection inherently incomplete approaches; security-aware scheduling hard to implement in an efficient way; blinding complex to implement, expensive at runtime, and specific to DMP-based side channels; oblivious execution and disabling the prefetcher permanently expensive at runtime.

8 DISCUSSION

8.1 Applicability to Different Scopes

Our current implementation of PREFENCE, as presented in Section 5.2, protects a userspace process from attacks by other userspace processes (scopes SP, CT, and CP). This covers 11 out of 13 attacks that we discussed in Section 4. The general principle of PreFence can also be used to protect kernel code from attacks from userspace (scope KU): the kernel could disable the prefetcher temporarily during security-critical operations. To prevent attacks from an untrusted operating system on a trusted execution environment (scope TO), the prefetcher could be disabled temporarily while securitycritical code is executed in trusted execution. However, because the untrusted operating system can usually interrupt trusted execution [18, 33, 42], the prefetcher's activation state needs to be saved when an interrupt causes a context switch to the untrusted operating system and restored when switching back. In the case of Intel SGX, which is interrupt-unaware [24, 42], this would require additional hardware support.

8.2 Applicability to Other Architectures And Flags

Other Architectures. In this paper, we implement and evaluate PREFENCE for two specific CPUs. Our targets are chosen to represent popular attack targets (such as the Intel IP stride prefetcher [9, 37, 45]) and cover two popular architectures (x86 64 and ARMv8). However, prior work has revealed that prefetcher implementations can differ widely, even across processors of the same brand or architecture [35]. We are confident that the design of Prefence is general enough to be transferred to other prefetchers as well, as long as those fulfill two basic requirements. First, the processor must expose a way to disable the prefetcher dynamically at runtime, for example by setting a bit in an MSR. Second, the prefetcher must not update its state while disabled. Unfortunately, there are some prefetcher implementations that do not fulfill the first requirement. For instance, Intel does not publicly disclose a way to control the XPT prefetcher, even though this prefetcher can be controlled from BIOS configuration tools [16]. Similarly, there is no publicly known way to disable the DMP prefetcher on Apple M1 and M2 processors [7]. Thus, we strongly encourage processor manufacturers to add such flags to future processor designs and ideally establish a well-documented standard to control prefetchers.

Other Flags. We emphasize that Prefence's general design is not limited to managing prefetching-related flags. Prefence could be adapted to control other microarchitectural components during the execution of security-critical code as well. For instance, ARM and Intel recently introduced the Data-Independent Timing (DIT) flag [1, 4] and the Data Operand Independent Timing Mode (DOITM) flag [15], respectively. Newer processors only guarantee data-independent timing behavior for certain instructions when these flags are set. These flags are meant to be set while cryptographic operations implemented in constant-time code are executed. They temporarily disable a range of optimizations that impact timing behavior (including, but not limited to certain prefetchers [7, 15]). Prefence could be adapted to dynamically set and clear these (or other) flags as well.

8.3 Covert Channels

In this paper, we focus on mitigating prefetching side channels and exclude covert channels from the scope.

Six prefetching-based covert channels have been proposed in recent work. Cronin et al. [12] implement a covert channel using the Intel IP stride prefetcher. They prime the prefetcher from the receiver's end, either evict or keep the primed patterns from the sender process, and finally probe for the existence of the primed patterns in the receiver process. This probing either triggers the prefetcher or not, indicating either a 0-bit or a 1-bit, respectively. Chen et al. [9] exploit the same prefetcher but encode the information to transmit into the stride. Rohan et al. [31] exploit the Intel stream prefetcher. The sender triggers the prefetcher on shared memory. The direction of prefetch (forward or backward) is interpreted as a 1-bit value by the receiver. Schlüter et al. [35] encode bit vectors into the region-based ARM SMS prefetcher to transfer information from trusted execution to the untrusted OS. Chen et al. [8] implement two covert channels based on the Intel XPT prefetcher. In the first scenario, the receiver primes the prefetcher's state and

the sender either idles or changes the state by evicting one of the primed entries. In the second scenario, the sender either trains or resets the prefetcher for a shared page.

All of these attacks have in common that not the victim but the attacker controls the training stage. In fact, there is no victim process at all in a typical covert-channel setting: Covert channels are incapable of leaking secret information on their own. Rather, they are a means of exfiltrating secret information that the attacker has obtained in another way beforehand. As these attacks do not leak information out of a victim process directly, a victim process has no incentive to defend against them. Thus, Prefence is not applicable.

8.4 Hardware-Based Countermeasures

In this paper, we focus on software-based mitigations because they can easily be applied and evaluated on current hardware. However, we also want to briefly discuss countermeasures against prefetching side channels that require hardware modifications.

Choosing a different trigger [35]. Some attacks rely on collisions on the prefetch trigger, e.g., the address of a load instruction. To mitigate such attacks, such collisions should be avoided. For example, the IP stride prefetcher of many Intel CPUs identifies a stride pattern by the partial address of a load instruction that caused the memory access. Because only few bits of that instruction address are considered, an attacker can cause such collisions on purpose.

The first intuition to fix this issue is to use full instruction addresses instead of partial addresses. However, this does not rule out the possibility of collisions completely, for instance in case of a shared library. To avoid this problem, the instruction address must not be used as the only trigger. For example, a process identifier could be added. Only if the process identifier *and* the instruction address match, a prefetch can be triggered.

Partitioning the Prefetcher's State [8, 9, 12, 35]. Similar to adding a process identifier, the prefetcher's state table could be partitioned based on one of multiple criteria. To avoid leakage between different privilege levels, such as kernel and userspace, or trusted execution and untrusted OS, the prefetcher could keep track of the privilege level that a pattern belongs to.

However, merely keeping track of the privilege level does not protect against attacks within the same level, such as leakage between two userspace processes. To separate those, it is again necessary to store an additional process identifier.

Partitioning protects from triggering a prefetch pattern accidentally in a wrong context. This ensures that no information about the stored patterns is leaked across privilege domains. However, depending on the implementation, attackers may still be able to prime the prefetcher with attacker-controlled patterns that are then potentially evicted by victim activity, leaking control flow information.

Extending the Instruction Set [34, 35]. Another option is to extend the instruction set by instructions that change the prefetcher's state or behavior. First, an instruction could be added that flushes the prefetcher's state. This instruction could then be called by the operating system on context switches, or when switching between privilege domains. However, this approach only works when the prefetcher is not shared among multiple cores or SMT

threads. Second, a special load instruction to be used in securitycritical code sections could be added that does not influence the prefetcher's state.

9 CONCLUSION

In this work, we addressed the challenge of efficient defenses against side-channel attacks exploiting the prefetcher to leak secret information from a victim userspace process. We started by providing the first systematic analysis of the existing 13 related attacks from literature, and showed that all of them rely on three main stages: prefetcher training, prefetcher triggering, and cache extraction.

Our proposed countermeasure, PreFence, allows vulnerable programs to ensure that they do not train the prefetcher. More precisely, it enables processes to selectively disable the prefetcher from userspace, while ensuring that parallel processes sharing the same prefetcher on other cores or SMT siblings are considered as well. In addition, issues in re-scheduling of processes are considered, and handled transparently for the vulnerable process. PreFence enables fine-grained control to minimize the time the prefetcher is unavailable, while ensuring that the critical prefetcher training attack stage cannot target the victim process any longer. Our prototype implementation of PreFence for Intel x86_64 and ARM Cortex-A72 processors is a Linux kernel patch to the scheduler to automatically ensure correct prefetcher state on re-scheduling of processes, together with a system call for processes to directly enable or disable the prefetcher. We provide our PreFence implementation as open-source software.

We demonstrated the efficacy of our approach by preventing a prior work attack successfully. Interestingly, our performance evaluation showed that the performance overhead is negligible if only parts of the application execute security-relevant code. In particular, we showed that the performance impact of our solution on non-security-critical code is usually below 1 %. For security-critical workloads, we found its performance impact to be dependent on the way Prefence is applied to the code; for a real-world web server application, we showed that the overall performance impact is negligible (less than 1 % on average) when Prefence is applied only to code sections related to cryptographic operations.

In conclusion, we presented a user-friendly and efficient scheduling-aware countermeasure to protect victim processes against prefetcher side channels, founded on a systematic analysis of prior work attacks and countermeasures. We expect our countermeasure could widely be integrated in commodity OS, and even be extended to signal generally security-relevant code to the kernel to allow coordinated application of countermeasures (e.g., DIT flags).

REFERENCES

- Apple Inc. 2024. Writing ARM64 Code for Apple Platforms. https://developer. apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms
- [2] Arm Ltd. 2020. Arm Architecture Registers for Future Architecture Technologies: CPP RCTX, Cache Prefetch Prediction Restriction by Context. https://developer.arm.com/documentation/ddi0601/2020-12/AArch64-Instructions/CPP-RCTX--Cache-Prefetch-Prediction-Restriction-by-Context
- [3] Arm Ltd. 2023. Architecture Security Advisory: Prefetcher Side Channels. https://developer.arm.com/documentation/109504/0100/
- [4] Arm Ltd. 2024. Arm® Architecture Registers for A-profile Architecture. https://developer.arm.com/documentation/ddi0601/2024-03
- [5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss.

- 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In Proceedings of the USENIX Security Symposium.
- [6] Sunjay Cauligi, Craig Disselkoen, Klaus V. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).
- [7] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In Proceedings of the USENIX Security Symposium.
- [8] Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E. Carlson. 2024. PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks. In IEEE International Symposium on High Performance Computer Architecture (HPCA).
- [9] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2023. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher.
- [10] Chromium Project. 2018. Mitigating Side-Channel Attacks. https://www.chromium.org/Home/chromium-security/ssca/
- [11] Mike Clark. 2016. A New x86 Core Architecture for the Next Generation of Computing. https://old.hotchips.org/wp-content/uploads/hc_archives/hc28/ HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.930-X86core-MikeClark-AMD-final_v2-28.pdf HotChips 28.
- [12] Patrick Cronin and Chengmo Yang. 2019. A Fetching Tale: Covert Communication with the Hardware Prefetcher. In IEEE International Symposium on Hardware-Oriented Security and Trust (HOST).
- [13] gonetx. 2022. gonetx/httpit: A rapid http(s) benchmark tool written in Go. https://github.com/gonetx/httpit
- [14] 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/securecoding/mitigate-timing-side-channel-crypto-implementation.html
- [15] Intel Corp. 2023. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance. https://www.intel.com/content/www/us/en/developer/ articles/technical/software-security-guidance/best-practices/data-operandindependent-timing-isa-guidance.html
- [16] Intel Corp. 2023. Hardware LLC Prefetch Feature on 4th Gen Intel® Xeon® Scalable Processor (Codename Sapphire Rapids). https://www.intel.com/content/ www/us/en/content-details/780991/ Revision: 1.0.
- [17] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the IEEE Symposium on Security and Privacy (S&P).
- [18] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. 2021. Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict. In ACM/IEEE Design Automation Conference (DAC).
- [19] Lighttpd Developers. 2024. 1.4.75 Lighttpd. https://www.lighttpd.net/2024/3/ 13/1.4.75/
- [20] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In Proceedings of the USENIX Security Symposium.
- [21] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In IEEE International Symposium on High Performance Computer Architecture (HPCA).
- [22] LLVM Project. 2024. Speculative Load Hardening LLVM 18.0.0git Documentation. https://llvm.org/docs/SpeculativeLoadHardening.html
- [23] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6, 1 (Feb. 2002).
- [24] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP).
- [25] Matt Miller. 2018. Mitigating Speculative Execution Side Channel Hardware Vulnerabilities. https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/ Microsoft Security Response Center (MSRC) Blog.
- [26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In Topics in Cryptology (CT-RSA).
- [27] Mathias Payer. 2016. HexPADS: A Platform to Detect "Stealth" Attacks. In Engineering Secure Software and Systems (ESSOS).
- [28] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really Are Constant-Time". In Proceedings of the ACM Conference on Computer and Communications Security (CCS).
- [29] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In Proceedings of the USENIX Security Symposium.
- [30] Raspberry Pi Ltd. 2024. Raspberry Pi Documentation The Linux Kernel. https://www.raspberrypi.com/documentation/computers/linux_kernel.html

- [31] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse Engineering the Stream Prefetcher for Profit. In IEEE European Symposium on Security and Privacy (EuroS&P) Workshops.
- [32] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. 2021. SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P).
- [33] Keegan Ryan. 2019. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In Proceedings of the ACM Conference on Computer and Communications Security (CCS).
- [34] 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 Proceedings of the IEEE Symposium on Security and Privacy (S&P).
- [35] 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 ACM Conference on Computer and Communications Security (CCS).
- [36] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. 2021. Specfuscator: Evaluating Branch Removal as a Spectre Mitigation. In Financial Cryptography and Data Security.
- [37] 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 ACM Conference on Computer and Communications Security (CCS).
- [38] Standard Performance Evaluation Corporation (SPEC). 2019. runcpu Using CPU 2017. https://www.spec.org/cpu2017/Docs/runcpu.html#iter
- [39] Standard Performance Evaluation Corporation (SPEC). 2022. SPEC CPU® 2017. https://www.spec.org/cpu2017/
- [40] Andrew S. Tanenbaum and Herbert Bos. 2015. Modern Operating Systems (4th ed.). Pearson. Boston.
- [41] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In Proceedings of the International Symposium on Computer Architecture (ISCA).
- [42] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In Proceedings of the Workshop on System Software for Trusted Execution (SysTEX).
- [43] 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
- [44] Luke Wagner. 2018. Mitigations Landing for New Class of Timing Attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack Mozilla Security Blog.
- [45] Chong Xiao, Ming Tang, and Sylvain Guilley. 2023. Exploiting the Microarchitectural Leakage of Prefetching Activities for Side-Channel Attacks. *Journal of Systems Architecture* 139 (June 2023).
- [46] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the USENIX Security Symposium.
- [47] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).
- [48] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In Proceedings of the ACM Conference on Computer and Communications Security (CCS).

A PRIOR-WORK PREFETCHING-BASED ATTACKS

In this section, we give a high-level overview of prefetching-based attacks in prior work and explain our dissection of these attacks into the five stages as illustrated in Figure 2.

Shin et al. [37]. This paper exploits the Intel IP stride prefetcher and targets the ECDH implementation in OpenSSL. In the preparation phase (S1), the attacker identifies memory lines the OpenSSL shared library code that are cached only conditionally depending on the supplied input. These cache lines only appear in cache when an internal lookup table is accessed such that a sequence of accesses forms a regular stride pattern. In the reset phase (S2), these memory locations are flushed from the cache by the attacker. Then, the

attacker calls the library. The library trains (S3) and triggers (S4) the prefetcher only if the supplied input leads to memory loads that form a stride pattern. The attacker probes the cache lines through a cache-timing side channel (S5) to decide whether the input triggered the prefetcher or not. This prefetching-based primitive is then embedded in a differential attack to recover the secret.

Augury [34]. This paper is the first to investigates the data memory-dependent prefetcher (DMP) of the Apple M1 SoC. It describes the prefetcher's behavior when multiple pointers that are stored sequentially in memory (e.g., in an array) are loaded and dereferenced: The DMP fetches subsequent pointers and dereferences them. The authors present multiple approaches to exploit this behavior.

The first approach ("Out-of-bounds reads", *Augury OOB*) assumes that a user secretly selects a pointer from a finite list of candidate pointers. This pointer is stored just behind an array of pointers in the victim's memory. The goal of the attacker is to find the chosen pointer without accessing it architecturally. The attack is described with Flush+Reload and Prime+Probe; we discuss the more complex Prime+Probe variant. The attacker sets up an eviction set for each of the candidate pointers (S1) and loads them (S2). Next, all the pointers in the pointer array are dereferenced to train the prefetcher (S3). When the end of the array is reached, the prefetcher is triggered to prefetch past the array bound (S4) and to dereference the user-chosen pointer. By timing the access latency to all candidate pointers using the eviction sets (S5), the attacker decides which of the pointers was chosen.

In a second approach (*Augury SLH*), the authors discuss the impact of the DMP on code that uses speculative load hardening (SLH) [22] to protect against Spectre attacks [17]. The core idea of SLH is to verify untrusted (e.g., user-supplied) memory offsets during speculative execution to prevent speculative out-of-bounds accesses. The compiler adds branchless code that replaces out-of-bounds offsets with a safe value (often 0) using binary arithmetic. In Augury's example, a code snippet trains the prefetcher by iterating over a pointer array (S3). While SLH prevents *speculative* out-of-bounds reads, the prefetcher is still able to prefetch past the array bound when triggered by an access to the last array element (S4). Thus, a pointer that is stored just behind the pointer array can be fetched and dereferenced by the prefetcher, leaving traces in the cache that can be recovered (S5).

A third approach (*Augury Addr*) describes how the DMP can be used to determine whether an address is a valid (mapped) virtual memory address or not. To this end, the attacker sets up an array of 3 pointers, where the third pointer is the address to test (S1). The attacker ensures that the array is not cached (S2). Next, the attacker traverses the array in speculative execution and within the context where the mapping is to be checked. This trains the prefetcher (S3). As the DMP requires at least three valid addresses to be triggered for the first time (S4), prefetching will only happen if the address is valid. The attacker tests the cache state of the first out-of-bounds element after the array of pointers (S5). If this element is cached, the tested address is valid, otherwise, it is invalid.

AfterImage [9]. This work exploits the Intel IP stride prefetcher five different ways. Generally, these approaches exploit collisions on the instruction pointer (IP) address. The exploited prefetcher identifies patterns stored in its internal state based on the instruction

address of the load instruction that caused the load. However, this instruction address is internally truncated to the 8 least-significant bits. Consequently, the attacker can cause a collision by aligning a load instruction in their own code such that its 8 least significant bits match the respective bits of the instruction address in the victim code. The prefetcher is then unable to distinguish those two instructions from different contexts.

In AfterImage variant 1, the prefetcher is used to leak the control flow of a victim process. This attack is described with sameprocess and cross-process scope as well as using Flush+Reload and Prime+Probe for extraction. We focus on the cross-process, Prime+Probe variant, which we consider the more complex one. The attacker's goal is to determine whether a branch in a victim process is taken or not taken. To this end, the attacker selects one load instruction from each of the two potential code flows in the victim process and aligns two load instructions in their own process to them (S1). The attacker further prepares (S1) and loads (S2) eviction sets on the load targets in the victim process. The attacker then primes the prefetcher (S3) by training it in the attacker's process. Then, a context switch to the victim process happens, where the branch is either taken or not taken and the respective load instruction is executed, further (mis)training the prefetcher (S3). As the prefetcher was pre-trained, this load will further trigger prefetching after the load target of the victim instruction (S4). The attacker extracts those prefetching effects from the cache by reloading the eviction sets (S5).

AfterImage variant 2 exploits the stride prefetcher to determine whether a branch is taken in the kernel. The authors attack a system call handler that operates on a memory buffer passed in from userspace. The idea is similar to variant 1, however, aligning to a load instruction in the kernel is more difficult as the instruction address is unknown. For this reason, the attacker first determines the offset by testing all $2^8 = 256$ possibilities in a process called *IP matching* (S1). Then, the prefetcher is primed in the attacker process (S3). The system call is issued. If the branch is taken, the prefetcher will be further (mis)trained (S3) and triggered (S4) to prefetch memory from the buffer. After returning from the system call, the attacker probes the cache state of the respective locations in shared memory (S5).

In addition, AfterImage describes an attack on *SGX*. This attack does not exploit a collision. The goal is to leak control flow from an enclave. In the described setting, a load instruction is executed in a loop within the victim enclave. The instruction loads with a secret-dependent stride from a shared buffer that is passed from userspace into the enclave. This loop trains (S3) and triggers (S4) the prefetcher. After returning to userspace, the attacker process recovers the stride by inspecting the cache state of the shared buffer (S5).

The paper further presents an attack on the RSA implementation of MbedTLS. The victim code contains secret-dependent branches that the attacker wants to track. To this end, the attacker first identifies suitable load instructions to align to by reverse-engineering the victim binary (S1). Next, the attacker primes the prefetcher in their own memory to a high confidence level (S3) and switches to the victim code. The victim executes a colliding load instruction and re-trains the prefetcher (S3). As the loaded address will likely not match the previously trained stride, the confidence will be lowered.

After switching back to the attacker process, the attacker tries to trigger the prefetcher in their own memory again (S4) and extract the prefetcher's behavior from the cache state (S5). The attacker will only observe prefetching effects if the confidence was not lowered by the victim, i.e., if the monitored branch was not taken.

Finally, the authors present a prefetcher-based *synchronization primitive* that operates similar to the RSA attack. They envision this primitive could be used as a trigger for a power-based side-channel attack, for example to detect the beginning of a cryptographic operation. Again, the attacker begins by identifying a load instruction to align to in the victim code (S1). The prefetcher is then primed on a colliding load instruction in the attacker's process to a high confidence level (S3). The attacker now switches frequently between victim and attacker code. As soon as the victim executes the target instruction, the prefetcher will be trained (S3) and the confidence will be lowered. The attacker tries to trigger the prefetcher (S4) and inspects the prefetcher's activity in the cache (S5). Once the prefetcher can no longer be triggered, the attacker knows that the victim executed the target instruction. In that case, the attacker raises a trigger signal.

Xiao et al. [45]. This paper uses the Intel IP stride prefetcher to attack an (undisclosed) AES implementation. To this end, the attacker monitors the cache state of the two memory lines just before and after the S-box. Depending on the access pattern to the S-box, which depends on plaintext and key, different prefetching activity in those cache lines is to be expected. After identifying the cache lines to monitor (S1), the attacker flushes them (S2). Then, the encryption is performed, potentially training (S3) and triggering (S4) the prefetcher. Finally, the cache state is inspected using a timing-based side channel (S5).

FetchBench [35]. This paper exploits the Spatial Memory Streaming (SMS) prefetcher in the ARM Cortex-A72 processor to attack the T-table-based AES implementation of MbedTLS. The attacker's goal is to extract the encryption key from the victim process. To this end, an instruction address collision is exploited. The authors first align load instructions in the attacker process with those in the victim process that load secret-dependent values (S1). They further identify a cache line that is accessed shortly before the victim process processes the secret (S1). This line, as well as a local probe array in the attacker process, are then flushed (S2). Next, the victim process encrypts an attacker-supplied plaintext using its own secret key. During encryption, the victim accesses multiple elements of a lookup table. The accesses to the lookup table train the prefetcher (S3). The attacker then switches into their own process using an inter-processor interrupt and triggers the prefetcher there (S4), making the prefetcher transfer the pattern learned in the victim context into the attacker's context. Finally, the attacker deduces the prefetcher's activity from the cache state (S5).

PrefetchX [8]. This paper uses the Intel eXtended Prediction Table (XPT) prefetcher to exploit the RSA implementations of MbedTLS and GnuPG. The XPT prefetcher is the only known prefetcher that is attached to the last-level cache (LLC) and thus shared across cores. It keeps a list of recently accessed pages and counts the number of cache misses per page. Once such a miss counter surpasses a fixed threshold for a page, the prefetcher effectively bypasses the LLC for future loads from that page.

The attacker and victim processes run on different cores. To start from a clean cache state, the attacker sends a signal to the victim process (S2). This enforces a context switch into the kernel and the resulting overhead evicts the target cache line. Next, the attacker fills the prefetcher's state with pages that they control (S3). Then, the victim executes a code section containing a secret-dependent load. If the load is performed, the prefetcher's state is updated and one of the attacker's pages are evicted from the state (S3). The attacker then checks the prefetcher's state by triggering it on their own pages (S4) and measures whether the prefetcher still triggers or not (S5).

GoFetch [7]. This paper revisits the DMP prefetcher of the Apple M1 SoC and its successors and discovers that the DMP is more than a pointer array prefetcher: It can be triggered by merely loading a single address from memory, even without dereferencing it.

The paper presents attacks on four real-world targets: the Go implementation of the RSA cryptosytem, the OpenSSL implementation of the Diffie-Hellman key exchange, and implementations of the post-quantum algorithms CRYSTALS-Kyber and CRYSTALS-Dilithium. All attacks are based on the same idea. The attacker crafts malicious inputs that, when combined with secrets during computation of the target algorithm, result in intermediate values that form a valid pointer if and only if the secret fulfills a certain condition. The prefetcher is only activated if the condition is fulfilled, leaking information about the secret. On a high level, those attacks perform the following steps. First, an attacker process prepares (S1) and loads (S2) eviction sets that evict the pointer's anticipated location and target address. Then, the input is crafted and supplied to the victim process. If the condition is fulfilled, an intermediate value in the victim context forms a pointer. Once this pointer is loaded, the prefetcher is trained (S3, it updates its history) and triggered (S4, it dereferences the pointer). Finally, the attacker process re-loads the eviction sets (S5) to determine whether the condition was fulfilled or not.

Notably, these attacks apply even to constant-time implementations, which only guarantee constant execution time and (architectural) memory access locations, but do not constrain intermediate values.

B PREFENCE EFFICIENCY EVALUATION: OVERHEAD ON CONTEXT SWITCH AND SYSTEM CALL

In this section, we perform two additional experiments on the efficiency of our Prefence implementation. We measure the fixed overhead caused by the additional kernel code that needs to be executed on every context switch and the overhead of performing a system call in order to set or clear the prefetch_disable bit of a process. Both experiments were only performed on the Intel CPU.

B.1 Fixed Overhead on Context Switch

Experiment. In this section, we evaluate the fixed overhead that our Prefence implementation adds to every context switch. We first measure the execution time of a context switch in the stock kernel and compare it to the execution time in our patched kernel.

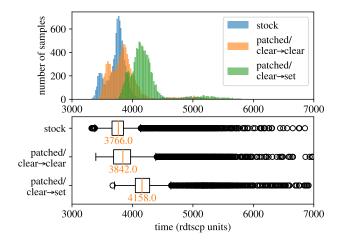


Figure 8: Context switch overhead on the stock kernel and on our patched kernel. For the patched kernel, we evaluate transitions between normal processes (prefetch_disable bits cleared) and the transition into a security-critical process (from bit cleared to bit set). The added overhead is negligible.

To this end, we implement two userspace processes that share a memory page. Both processes are pinned to the same CPU core. The first process constantly writes the current high-resolution timer value retrieved from the rdtscp instruction to shared memory. The second process reads the timer value from shared memory and computes the difference to the current timestamp. When the first process is scheduled, it keeps incrementing the timestamp written to memory until it is descheduled. Next, the second process is scheduled, computes the timestamp difference and logs the result. We filter out "zero samples" caused by the second process being re-scheduled before the first one, i.e., where the timestamp in memory has not been incremented compared to the last execution of the second process. We run this experiment on an idle system to maximize the probability of a context switch between our two processes.

Results. We measure the execution time of a context switch in three scenarios on our Intel CPU: (1) with the stock kernel, (2) with our patched kernel, switching between two processes with the prefetch_disable bit cleared, (3) with our patched kernel, switching from a process with the prefetch_disable bit cleared into a process with the bit set. We repeat each experiment until 10,000 non-zero samples have been collected.

We present our results in Figure 8. Not surprisingly, the stock kernel has the smallest median context switch execution time of 3766 time units. Switching between two normal processes on our patched kernel requires 3842 time units (median), an negligible increase of 76 units or 2 %. When the prefetcher's state needs to be changed on context switch, the median execution time is 4158 units, an increase of 392 units or 10 % compared to the stock kernel.

B.2 One-off Overhead of the System Call

Experiment. We set up a userspace process that performs the prctl system call twice, once to set the prefetch_disable bit, and once to clear it again. Before and after each of the system calls,

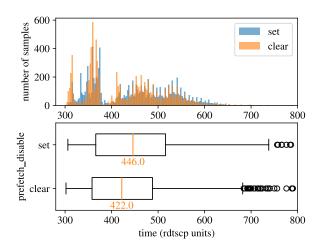


Figure 9: Overhead of a system call that sets or clears the prefetch_disable bit.

we use the rdtscp instruction to get high-precision timestamps. Finally, we compute the difference between the timestamps.

Results. We repeat the experiment 10,000 times on our Intel CPU. The results are plotted in Figure 9. We note that the median duration for both system calls is around 430 units. The median duration of the system call to set the flag is negligibly longer than the median duration of the system call to clear it. For comparison, the overhead of the system call is roughly in the same order of magnitude as a memory load that misses the cache (in the OpenSSL experiment in Section 6.3, we observed that a cache miss takes around 340 units on the same system).