PREFENCE: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks

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

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

***Abstract*—Speculative loading of memory, called *hardware prefetching*, is common in modern CPUs and may cause microarchitectural side-channel vulnerabilities. As prior work has shown, prefetching can be exploited to bypass process isolation and leak secrets. However, to this date, no effective and efficient countermeasure has been presented that secures software on affected systems. Often, disabling prefetching permanently is considered the only reasonable defense, despite the significant performance penalties this entails.**

In this work, we propose PREFENCE, a fine-grained and scheduling-aware defense against prefetching-based attacks for any platform where the prefetcher can be disabled. PREFENCE extends the process scheduler to be aware of security requirements of individual processes and to manage the prefetcher's state to protect against malicious parallel processes, even on SMT-enabled platforms. This allows us to efficiently disable the prefetcher only during security-critical operations, with a single system call. Library and application developers can protect their code with minimal changes, and users can protect entire legacy applications using a wrapper program.

We implement our countermeasure for an x86_64 and an ARM processor. We evaluate PREFENCE on two attacks from prior work and find that it reliably stops prefetch leakage with low performance overhead (less than 3%) on the vulnerable functions. In addition, we observe that PREFENCE causes only negligible performance impact when no security-relevant code is executed. Finally, we evaluate the performance of a real-world web-server application that uses PREFENCE to protect security-critical code for HTTPS handling. Compared to disabling the prefetcher permanently, we find that our countermeasure allows the application to significantly benefit from the prefetcher (running up to 15.8% (Intel) and 7.2% (ARM) faster on average), while at the same time achieving the same security.

1. Introduction

Prefetching is an optimization mechanism in modern CPUs that aims to bring relevant chunks of memory into the cache before they are actually loaded by application code. Ideally, this allows applications to benefit from lower memory latency. There are two types of prefetching: software prefetching and hardware prefetching. Software prefetching relies on explicit hints issued by application software that 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.

Hardware prefetching can be further divided into address-based approaches and data memory-dependent approaches. Address-based prefetchers monitor the accessed memory *addresses* and attempt to identify specific access patterns to predict the next addresses to be accessed. When secret-dependent memory accesses occur, these accesses may leave traces in the prefetcher's internal state [10], [42]. Afterward, the prefetcher's predictions may correlate with the secret. An attacker can then observe traces of the prefetcher's activities in the cache and infer the secret. In contrast, data memory-dependent prefetchers (DMPs) derive their predictions from *values* loaded from memory. Thus, DMPs cause leakage when they observe secret-dependent values and process them [8], [40]. Again, attackers can infer the secrets by observing the prefetcher's behavior and the resulting modifications to the cache state.

Prior work has shown that prefetcher-based side channels can be exploited to compromise, e.g., Diffie-Hellman keys [8], [44], RSA private keys [8], [10], or AES keys [42], [51]. In addition, covert channels have been presented [10], [12], [38], [42], in some cases bypassing process isolation guarantees. No defense has been presented to date that protects against such attacks effectively and efficiently. For instance, while most platforms allow hardware prefetchers to be completely disabled, this measure has a significant performance impact, as non-security-relevant code can no longer benefit from prefetching.

In this paper, we propose PREFENCE, our novel countermeasure that allows user-space code to defend itself against the perils of hardware prefetching on affected CPUs with minimal overhead. With PREFENCE, processes gain fine-grained control over the prefetcher. Processes indicate to the scheduler when security-relevant operations are executed, so that the kernel can disable prefetchers temporarily. This signaling requires only minimal code changes and can be integrated at different levels. For example, when integrated at library level, all applications that use the respective library benefit from PREFENCE automatically. Alternatively, signaling can be integrated into application code or even be performed manually on a per-process basis by the user. Our solution has negligible performance overhead on non-security-critical workloads. For an application containing security-critical code sections, we find that our countermeasure allows the applica-

tion to run up to 15.8% (Intel) and 7.2% (ARM) faster on average compared to disabling prefetching permanently.

We systematically analyze existing side-channel attacks that exploit hardware prefetching, identify their differences and similarities, and find suitable entry points for defenses. Based on these insights, we design, implement and evaluate PREFENCE, an enhancement to the process scheduler to make it security-aware. Our approach enables processes to ask the kernel 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 control the prefetcher at runtime 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 make the following contributions:

- We systematize existing prefetching-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 PREFENCE, which enables the mitigation of prefetching-based attacks in a fine-grained manner. Our solution enhances the scheduler to be security-aware, managing prefetcher state for all processes.
- We implement and evaluate PREFENCE for an x86_64 and an ARM processor. We demonstrate that PREFENCE prevents two prior-work attacks, that its performance impact is negligible for non-security-critical workloads, and that a security-critical workload performs significantly better with PREFENCE compared to prefetching being permanently disabled.

Availability. We provide an open-source implementation of PREFENCE at `https://github.com/scy-phy/PreFence`.

2. Background

2.1. Caches and Hardware 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 instruction is significantly faster. Otherwise, the data needs to be fetched from DRAM, which takes more time.

Hardware 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 activity at runtime and predicts addresses that are likely to be accessed next. Prefetching is a completely transparent mechanism from the application's point of view. The prefetcher's prediction mechanisms are often undocumented [42]. While most prefetchers only take memory addresses into account to generate predictions, more powerful *data memory-dependent prefetchers (DMPs)* also consider the data stored at those addresses [8], [40]. 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. This makes 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 [47].

Simultaneous Multithreading (SMT) [48] is a concept that aims to better utilize the resources of a processor core. It schedules multiple threads on a single processor core at the same time, based on the insight that a single thread often cannot utilize all the processing units available in a core. SMT has been adopted by major processor vendors such as Intel (branded "HyperThreading") [30] and AMD [1]. Their implementations expose one physical processor core as multiple 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 cores in the same way as it would on a physical cores [30].

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 system, instructions from parallel threads that are scheduled on sibling cores are processed concurrently and share processor resources, potentially also the prefetcher. We emphasize that, as a result of SMT, instructions issued by multiple processes can run on the same physical core without requiring a context switch.

2.3. Prior Work on Mitigations

Mitigations proposed in prior work can be divided into two classes: software-based and hardware-based. All mitigation approaches either try to prevent the leakage in the first place, suppress potential leakage when crossing privilege boundaries, block building blocks of specific attacks, or detect attacks heuristically. We discuss mitigations from prior work in detail in Section 8 and relate them to our approach.

3. Defending Against Prefetching Attacks

3.1. System and Attacker Model

We assume a system with a processor that performs hardware 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 [42]). 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 user space. In Section 7.3, 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?

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

1) 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) Countermeasures require trustworthy arguments on why they can be expected to prevent current and future attacks. So far, mitigations have been discussed briefly and in the context of specific attacks as part of offensive papers, but never considering the class of prefetching-based side channels as a whole.
3) Any countermeasure will cause a performance impact, which needs to be quantified and minimized.

Proposed Approach. To overcome these challenges, we pursue the following approach. First, we systematize known prefetching-based side-channel attacks and the vulnerabilities they exploit. We identify a minimal set of vulnerabilities that are required for any attack to work. Based on that, we design and implement a solution to prevent exploitation of this minimal set of vulnerabilities, leading to a countermeasure effective against all prefetcher side-channel attacks from user space. We then evaluate the implemented solution on real-world hardware.

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). Inspired by prior works on mitigating other microarchitectural side channels [6], [32], [43], we break down prefetching-based attacks into stages (Section 4.1). We further define the scopes in which those attacks operate (Section 4.2). Finally, we visualize our systematization by plotting the attack sequences, deduce similarities and differences, and expose where software-based mitigations can effectively be applied (Section 4.3).

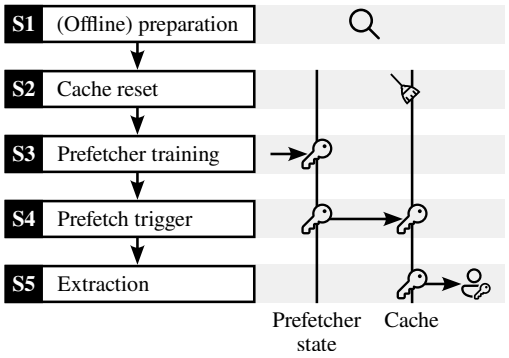


Figure 1. Stages of prefetching-based side channels

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 state, some attacks include actions to reset the initial cache state.
- **S3: Prefetcher Training.** Most prefetchers keep an internal state that governs their behavior. The prefetcher continuously observes memory activity to identify patterns in the addresses or data being accessed. When a new pattern is detected or existing patterns are altered or interrupted, the prefetcher's internal state changes. Thus, the prefetcher's future behavior changes as well. We refer to this state change as *prefetcher training*. 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 state. We refer to this process as *prefetch trigger*. 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, as we show in Section 4.3.

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 user-space process to another.
- **CP: Cross-process.** Leaking from one user-space process to another.

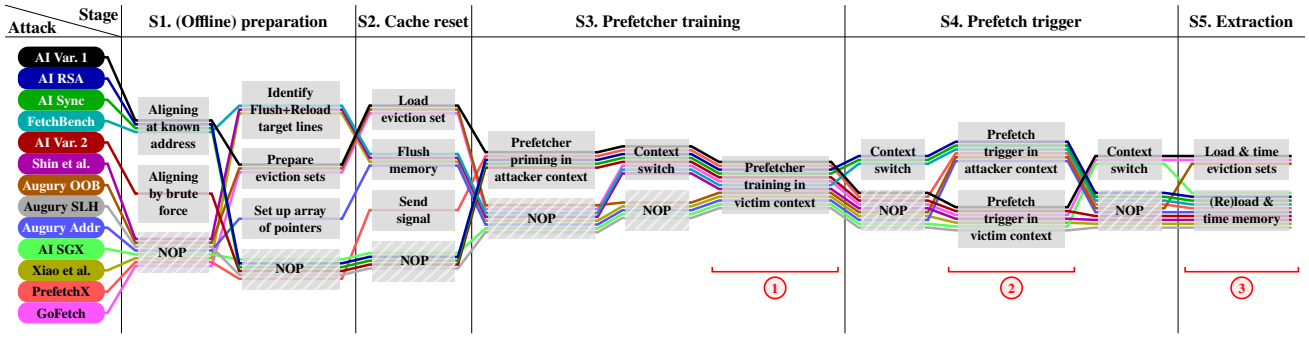


Figure 2. Overview of the sequence of activities in prefetching-based attacks. Core activities required by all are highlighted in red.

- **KU: Kernel to user.** Leaking from kernel to user space.
- **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 operate in different contexts (e.g., in different user-space processes), the attacker has to ensure that the prefetcher keeps its state across the context switch. Especially when leaking between two user-space 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 [10]; others use additional side channels for synchronization [42].

4.3. Prefetching Attack Systematization

We now systematize 13 attacks from prior work to answer RQ1. We list these attacks in Table 1 and denote the prefetcher types they exploit, the scope the attacks operate in, and the attack targets. In addition, we map all attack procedures to the five stages introduced above (details in Appendix A). We summarize our results in Figure 2, which shows 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. We highlight four findings:

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 prefetching-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 [52] and Prime+Probe [33].

Finding: Prefetching Is Triggered by Victim or Attacker. We find 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 [9], [10], [42].

TABLE 1. PREFETCHING-BASED ATTACKS IN PRIOR WORK. PREFETCHERS MARKED WITH † ARE DMPS, ALL OTHERS ARE ADDRESS-BASED. SCOPES ARE EXPLAINED IN SECTION 4.2.

Attack	Prefetcher	Scope	Target
Shin et al. [44]	Intel IP stride	CP	OpenSSL ECDH
Augury [40] OOB	Apple DMP†	SP	Custom
Augury [40] SLH	Apple DMP†	SP	Custom
Augury [40] Addr.	Apple DMP†	SP	—
AfterImage [10] Var. 1	Intel IP stride	CT/CP	Custom
AfterImage [10] Var. 2	Intel IP stride	KU	Custom
AfterImage [10] SGX	Intel IP stride	TO	Custom
AfterImage [10] RSA	Intel IP stride	CT	MbedTLS RSA
AfterImage [10] Sync	Intel IP stride	CP	OpenSSL RSA
Xiao et al. [51]	Intel IP stride	SP	AES
FetchBench [42] AES	ARM SMS	CP	MbedTLS AES
PrefetchX [9]	Intel XPT	CP	MbedTLS RSA, GnuPG RSA
GoFetch [8]	Apple DMP†	CP	Go RSA, OpenSSL DHKE, CRYSTALS

Finding: The 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 prefetching-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 is based on the insight that the prefetcher is trained by the victim process in all attacks. We discuss alternative software-based defense approaches and why we consider them infeasible in Section 8.1.

5.1. Design Considerations

Design Goals. We want our countermeasure to fulfill the following design goals:

- **DG1:** It mitigates all prior prefetching-based attacks conducted from user space.
- **DG2:** It is simple to use for application developers and end users.
- **DG3:** It has minimal runtime overhead.
- **DG4:** It is functional when the prefetcher is shared across physical or SMT sibling cores.

Idea: Disabling the Prefetcher Temporarily. As we have shown in Section 4.3, stage S3 is a critical phase in all attacks. In this stage, the victim process encodes secrets into the prefetcher's state. This is where our countermeasure intervenes: Our idea is to allow user-space applications to disable the prefetcher temporarily in a fine-grained manner, for example while security-critical code is executed. 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. In this way we fulfill DG1, since we avoid secrets being encoded into the prefetcher's state. At the same time, we limit the performance impact of the prefetcher being disabled to the security-critical code only, thus enabling DG3.

Naive Design: Giving User-Space Applications Direct Control. The following naive defense idea sounds straightforward at first: We could directly expose the processor's model-specific registers (MSRs) for prefetch control to user-space applications, giving them full control over the prefetcher's activation state. However, upon closer inspection, we identify four problems.

Problems With the Naive Design. First, we note that exposing the relevant MSRs directly to user-space applications requires application developers to write processor-specific code, since MSRs are not standardized (see Section 7.4). This violates DG2.

Second, we identify two issues that may occur when a security-critical process (which disabled the prefetcher) is descheduled and a non-critical process is scheduled instead: (i) the interrupting process cannot benefit from the prefetcher's performance improvements, violating DG3, and (ii) the interrupting process could maliciously re-enable the prefetcher, potentially violating DG1. Because scheduling events are transparent to applications, applications cannot easily address these issues. Thus, our solution needs to ensure that the prefetcher is re-enabled while other non-critical processes are running and disabled again when the security-critical process is re-scheduled.

Third, we identify a problem with prefetchers that are shared across physical or SMT sibling cores. A co-located malicious process with full control over the prefetcher could re-enable the prefetcher while the security-critical process is running, violating DG4. Consequently, we need to ensure that a prefetcher cannot be enabled if any of the co-located processes sharing that prefetcher requested it to be disabled.

Fourth, we note that 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 to fulfill DG1 and DG3.

The above problems clearly show that we have to dismiss the naive design: We cannot provide user-space applications with direct access to the prefetcher's activation state. In the next section, we propose PREFENCE, our improved mitigation design that addresses these problems.

5.2. PREFENCE Design

Solution: Scheduling-Aware Prefetch Control. Our proposed design, PREFENCE, addresses the problems with the naive design. It provides user-space applications with reliable deactivation of the prefetcher, regardless of the state of parallel processes and scheduling. We achieve this by tightly integrating our solution with the operating system kernel and the scheduler. The kernel can provide a unified interface to the user-space applications, abstracting away any processor-specific implementation details, thus enabling DG2. In addition, by integrating our defense into the scheduler, we can better address DG1, DG3 and DG4 and has to ensure that the prefetcher is disabled exactly as long as required, regardless of intermittent scheduling events.

Final Design. PREFENCE empowers user-space code 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. This signaling requires only minimal code changes and can be integrated at various levels, from very fine-grained to more coarse-grained. If it is integrated at library level (a fine-grained way of applying PREFENCE), all programs that use the respective library benefit from it automatically. Alternatively, PREFENCE can also be integrated at application level, either to protect application-specific security-critical code or to wrap calls to security-critical legacy libraries that do not yet use PREFENCE themselves. Finally, PREFENCE also allows the user to decide that prefetching should be disabled for an entire process (the most coarse-grained way of applying PREFENCE). The implementation effort for library and application developers is low compared to complex re-writes required by other countermeasures (such as constant-time programming, see Section 8.1). In this way, PREFENCE achieves DG2.

Handling Scheduling. The kernel has to keep track of whether a process is currently in a security-critical code section and ensure that the prefetcher is disabled during this period—even if the process is interrupted by the scheduler. Once the prefetcher has been disabled, it remains disabled until the requesting process no longer executes security-relevant code, or until a different process not running security-relevant code is scheduled. 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 activation state based on the security state of the next process to be executed.

The Simple Case: Per-Core Prefetcher, No SMT. We use the example in Figure 3 to illustrate our countermeasure, starting with the simple case in the upper half (a): 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 re-enabled. 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 pay attention to 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 re-enabled. Similarly, if the prefetcher's state is shared across cores, an attack could be run simultaneously from a different core. For this reason, it is insufficient to update the prefetcher's activation 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 Figure 3 (b). 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 scheduler. Since no more processes request the prefetcher to be disabled after P5 is descheduled, the prefetcher is re-enabled. 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 rescheduled. 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 activation state based on the request of the next process. When a core migration occurs, the prefetcher's activation 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 determines the prefetcher's activation state.

5.3. PREFENCE Implementation

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 user space, we add options to set, clear or query the flag to the `prctl` system call. When the flag is changed through

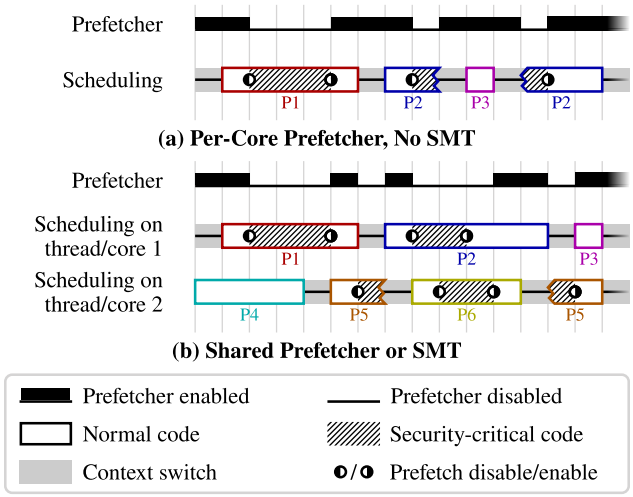


Figure 3. PREFENCE at work: The prefetcher is disabled temporarily while security-critical code runs. On SMT-capable cores and for shared prefetchers, the scheduler considers requested activation states of the relevant parallel processes.

the system call, the kernel updates the `task_struct` of the calling process accordingly. In addition, the kernel changes the prefetcher's activation state on the respective CPU immediately by writing to the corresponding MSRs (bits 1-4 in MSR `0x1A4` on our Intel CPU [21], bits 21, 32, 42, 56 in MSR `CPUACTLR_EL1` on the ARM A72 [3]) before returning to user space. These MSRs can only be modified from kernel space. We provide an overview of prefetch-related MSR flags for various microarchitectures in Section 7.4. We further extend the scheduler's `context_switch` function to update the activation 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 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. These system calls can be issued in a fine-grained manner in library code or application code. PREFENCE can even be made available to users who want to protect legacy software: Users can invoke a wrapper program similar to `taskset` that just sets the *prefetch_disable* flag and executes the target application, which then inherits the flag. This effectively sets the *prefetch_disable* flag in a more coarse-grained way, affecting the entire codebase of the target application.

6. Evaluation

In this section, we evaluate PREFENCE for efficacy and efficiency, based on our implementation. First, as a prerequisite, we evaluate the behavior of a disabled prefetcher to verify that disabling the prefetcher has the

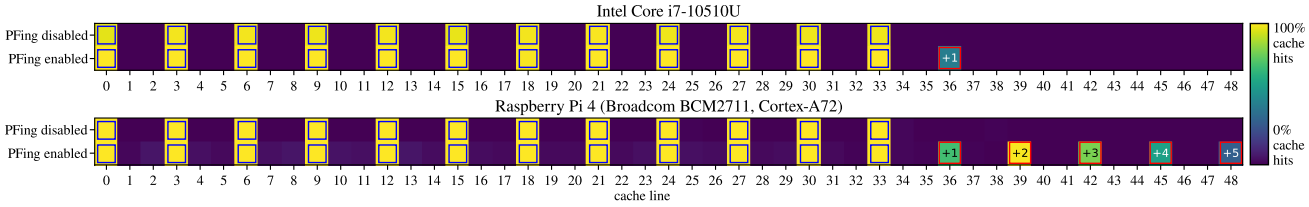


Figure 4. Prefetcher behavior when trained while disabled (compared to the behavior when trained while enabled). Brighter colors denote cache hits, darker colors denote cache misses. Blue frames indicate accesses, red frames indicate prefetch locations. The tested prefetchers cannot be trained while disabled.

expected effects (especially, 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. [44] as well as the end-to-end attack proposed by Schlüter et al. [42] (both reproduced by us). Finally, we show that PREFENCE is also efficient. We investigate 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 represent the currently most viable general countermeasure and 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 requests prefetching to be disabled. 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 system call, we disable the prefetcher during execution of TLS-related code. In addition to cryptographic code, each HTTP request to our server also triggers application code that can still benefit from the prefetcher (e.g., a file upload).

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. We report these additional results 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 [37] 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 [42]. 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 can be used with these prefetcher implementations.

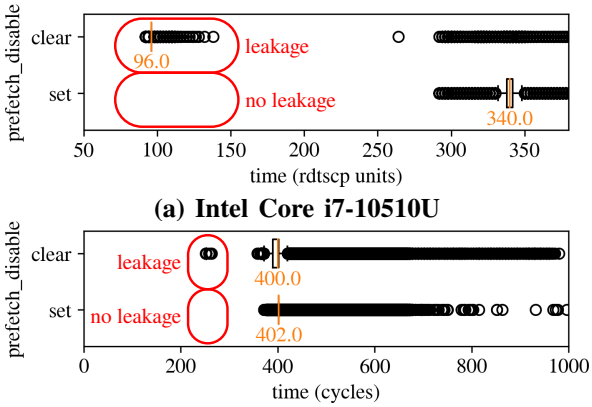
6.3. Efficacy: Protecting OpenSSL

In this experiment, we evaluate the efficacy of our countermeasure using the attack by Shin et al. [44] 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 accesses (by chance) form a regular pattern, the prefetcher is activated and fetches memory lines before and/or after the lookup table. This prefetcher activity 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



(a) Intel Core i7-10510U

(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.

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 certain 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 units), 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 there). Without a countermeasure, the prefetching leakage appears at around 250 cycles. When we set the *prefetch_disable* flag before calling the target function, we observe that the leakage disappears reliably. We conclude that PREFENCE successfully prevents the prefetching-based side channel on both CPUs.

Execution Time Evaluation. For completeness, we also evaluate the temporal overhead of the prefetcher being temporarily disabled while the vulnerable library function is executed. We use the same experimental setup as before, but we additionally measure the execution time of the `BN_GF2m_mod_sqr_arr` function with and without the *prefetch_disable* flag set.

On the ARM CPU, we measure a slowdown of 2.6% when the prefetcher is disabled (the median execution time increases from 550 to 564 cycles). On the Intel CPU, we find that the median execution time even decreases slightly when prefetching is disabled (from 374 to 368 units, a speedup of 1.8%), which we attribute to the prefetcher interfering with non-ideal predictions when it is enabled.

However, these measurements only reflect the performance of PREFENCE in an artificial individual case. Thus, we conduct an in-depth efficiency evaluation based on more complex and realistic workloads in Sections 6.5 and 6.6.

6.4. Efficacy: Protecting MbedTLS

Next, we show that PREFENCE successfully prevents an end-to-end attack from prior work, namely the attack on MbedTLS AES from the FetchBench paper [42]. As this attack exploits ARM's Spatial Memory Streaming (SMS) prefetcher, we can only reproduce it on our ARM-based platform.

Vulnerability. The SMS prefetcher divides memory into fixed-size regions of 1 KiB each. When a load instruction accesses multiple cache lines within the same region (e.g., in a loop), the prefetcher records this access pattern in its internal state. As the vulnerable AES-128 implementation issues key-dependent accesses to lookup tables (which span multiple such regions) during encryption, key-dependent information is encoded into the prefetcher's state. An attacker can extract this state and recover up to half of the secret key bits (i.e., 64 bits) using a properly aligned (aliasing) load instruction in their own code running on the same CPU core.

Experiment. We run two experiments: First, as a baseline, we run the end-to-end attack on our patched kernel, but without making any PREFENCE system calls in the victim code. This configuration is expected to show leakage. We record how many secret bits can be recovered successfully. Second, we repeat the attack, but with PREFENCE applied. We set the *prefetch_disable* flag in the victim code before calling the AES encryption function and clear it afterward. Again, we record the leakage.

Implementation. We build upon the proof-of-concept code published by Schlüter et al. [41]. Due to the complex and unreliable synchronization between the attacker and victim processes, the original attack has a low success rate. This is not ideal for our experiment, because a failed attack (due to failed synchronization) is hard to distinguish from a prevented attack. Thus, we strengthen the attack to increase its success rate. We eliminate the need for synchronization through side channels by merging victim and attacker code into a single process and returning from the vulnerable library function right after the secret-dependent accesses have occurred. We further replace the cache inspection mechanism: Instead of Flush+Flush [14], we use the privileged RAMINDEX interface [3]. We emphasize that these changes only strengthen the attacker, not the victim. We will now show that PREFENCE can protect even against this stronger attacker.

Results. We repeat both experiments 200 times each and generate a fresh random key for every run. One execution of the end-to-end attack takes approx. 24.2 minutes on average. Figure 6 illustrates our results. Without protection through PREFENCE (green), we are able to leak 58.8 out of 64 recoverable key bits successfully on average. We leak all recoverable key bits in 35% of the attacks. With PREFENCE applied (purple), we observe a success rate that is essentially equivalent to random guessing,

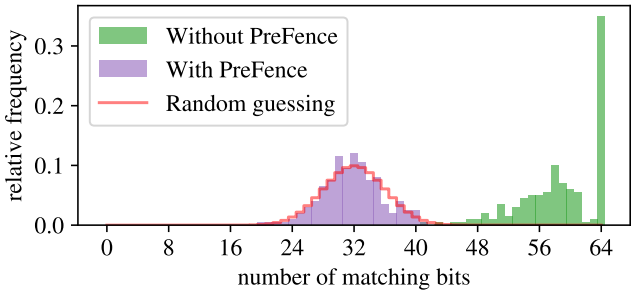


Figure 6. Results of the reproduction of the attack from prior work on MbedTLS AES [42] with 200 repetitions per configuration. The histogram shows how many key bits the attacker is able to extract correctly. When PREFENCE is not applied (green), all 64 key bits can be extracted in 35% of the cases. When PREFENCE is applied (purple), the attack is mitigated and the attacker's success rate drops to the level of random guessing.

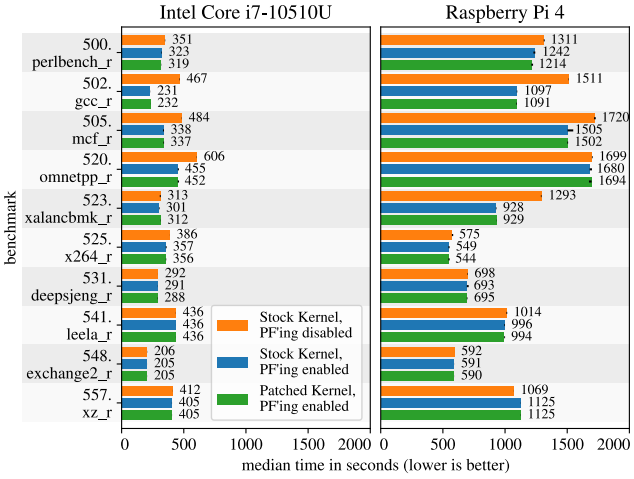


Figure 7. SPEC CPU 2017 benchmark results. Disabling the prefetcher permanently causes significant performance overhead in benchmarks 502 to 523. The performance overhead introduced by our patched kernel is negligible for non-security-critical workloads.

with an average success rate of 31.8 correct key bits per attack. The red line indicates the expected distribution for random guessing, more precisely, a binomial distribution with $n = 64$ independent guesses, where each bit guess is correct with a probability of $p = 0.5$. This expected distribution closely matches the observed distribution with PREFENCE applied. We conclude that PREFENCE successfully mitigates this attack.

Execution Time Evaluation. Finally, we also measure the temporal overhead on the vulnerable library function caused by the lack of prefetching. To this end, we call the function 10,000,000 times with and without the *prefetch_disable* flag set and measure its execution time. We find that the median execution time increases by approx. 2.7% when prefetching is temporarily disabled (from 903 to 927 cycles).

6.5. Efficiency: Non-Critical Workloads (Scenarios 1 and 2)

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

Experiment. We run *SPEC CPU 2017* benchmarks [46] on three different system configurations. As baselines, 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 (the maximum number of iterations in a "reportable run" [45]).

Results. Figure 7 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.

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). Nevertheless, 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.

6.6. 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* [25] (released in March 2024) as an example. Lighttpd ships with plugins for various cryptographic libraries that can act as backends to provide HTTPS support. In the following experiments, we use the OpenSSL plugin for this purpose. Our goal is to protect the key material processed by OpenSSL from prefetch-related side-channel leakage. We compare two approaches of applying PREFENCE to the web server: fine-grained (at plugin level) and coarse-grained (at application level).

Fine-Grained PREFENCE. In this approach, we apply PREFENCE in a more fine-grained manner, i.e., at plugin level. We modify lighttpd's OpenSSL plugin 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 and the hosted web application to benefit from prefetching but causes frequent system calls to enable or disable the prefetcher.

Coarse-Grained PreFence. In this approach, we apply PreFence in a more coarse-grained way, i.e., at application level. We use a wrapper program to set the *prefetch_disable* flag immediately when lighttpd is started. In other words, the prefetcher is disabled for the whole lighttpd application, including all critical and non-critical server code and any hosted web application that is executed in child processes of the web server. This means that the server application cannot benefit from prefetching at all, but fewer system calls are required.

Web Application. In this case study, 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 PHP and attached to lighttpd via CGI. It implements a simple file sharing service that allows the user to upload a file that is then stored on the server. During the HTTPS-encrypted upload, both the web server and the web application will process the file contents from beginning to end. We anticipate that these operations benefit from prefetching, especially for larger files.

Benchmarking Approach. We configure lighttpd to serve our web application via HTTPS over TLSv1.3 and access it through a benchmarking script. The script communicates with the web server via the loopback interface and uploads files of different sizes containing random data. We use the duration of the POST request that performs the upload as a performance measure. We repeat each experiment (i.e., each file upload) 300 times. To evaluate the performance of this setup when protected with PreFence, we evaluate it in scenario 3, i.e., with a patched kernel and prefetching temporarily disabled through our PreFence system calls. 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. We present the results in Figure 8. The x-axis indicates the size of the uploaded file. The y-axis indicates the duration of the upload POST request, relative to a baseline. This baseline is the median duration of the request in scenario 1 (stock kernel) with prefetching permanently disabled, normalized to 1, as shown by the black line. We represent the distribution of measurement values as boxplots. For a clearer presentation, we exclude the top and bottom 2% of values from each boxplot in this figure. The horizontal lines connect the boxplot's medians.

Finding: Application Benefits From Prefetching. Our first finding is that our example application generally benefits from prefetching on both CPUs. With prefetching permanently enabled (i.e., without any protection; blue/green lines), the upload performs 20.3% faster on average on the Intel CPU and 10.7% faster on average on our ARM processor compared to the baseline of disabling prefetching permanently. Thus, disabling prefetching permanently would be a costly measure.

Finding: PreFence Always Faster Than Baseline. All PreFence variants reduce this cost significantly; they always perform better than the baseline (permanently disabling prefetching).

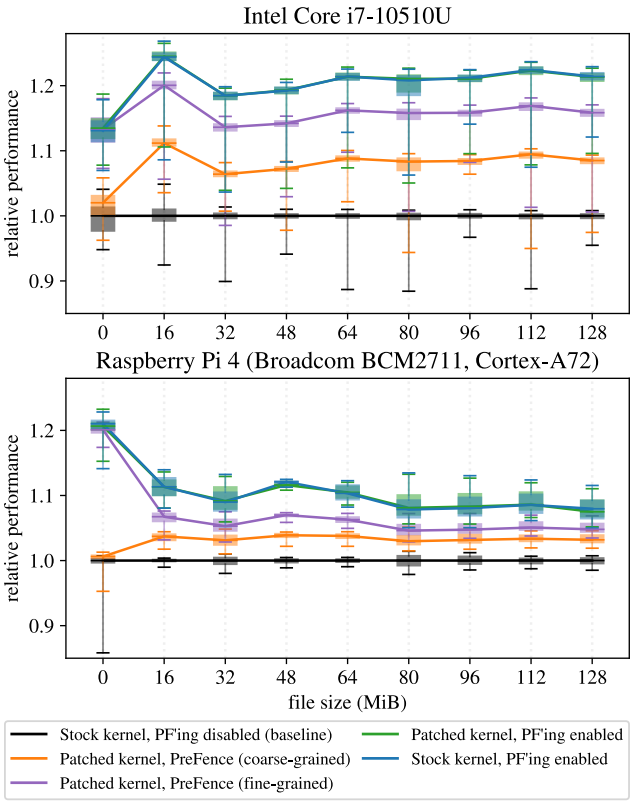


Figure 8. Lighttpd benchmark results, showing the performance benefit of PreFence compared to prefetching being permanently disabled for a security-critical workload (black). Prefetching generally speeds up the process, and once a significant file size is reached, the relative performance benefit is approximately constant. Any form of PreFence performs better than disabling the prefetcher permanently. Fine-grained use of PreFence (purple) improves performance by 15.8% (Intel) and 7.2% (ARM) over the baseline on average, approaching the performance of prefetching being enabled permanently (blue/green).

Finding: Fine-Grained PreFence is Faster. We observe that the fine-grained use of PreFence at plugin level (purple) performs better than the more coarse-grained use of PreFence at application level (orange) in this experiment. The lower number of system calls in the application-level scenario does not compensate for the slowdown of the non-critical parts of the web server and the web application. The better-performing variant, PreFence at plugin level (purple), is not as fast as the insecure default configuration (i.e., having prefetching permanently enabled; blue/green), as the OpenSSL code can no longer fully benefit from prefetching. However, with fine-grained use of PreFence, we are able to reclaim a performance improvement of 15.8% on average over the baseline on the Intel CPU and 7.2% on the ARM processor. For coarse-grained use of PreFence, the average performance improvement is still 7.8% (Intel) and 3.1% (ARM) over the baseline.

Finding: Negligible Performance Impact of Scheduler Patch. We observe again that the overhead of our scheduler patch is negligible for non-critical code, as the blue and green lines overlap in both plots. This confirms our result from Section 6.5 that programs that do not use PreFence do not suffer a noticeable performance drop.

6.7. 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) and mitigates an end-to-end attack from prior work successfully (Section 6.4). 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 6.5). For our example of a security-critical workload, a lighttpd web server serving a web application that benefits from prefetching via HTTPS, we found that PREFENCE always performs better than disabling the prefetcher permanently. More specifically, when PREFENCE is applied in a fine-grained manner at the level of the OpenSSL plugin, our example application runs up to 15.8% (Intel) and 7.2% (ARM) faster on average compared to the prefetcher being permanently disabled (Section 6.6).

7. Discussion

Next, we discuss the applicability of PREFENCE to different types of hardware prefetchers (Section 7.1) and an approach to automate the insertion of its system calls (7.2). In addition, we discuss the applicability of PREFENCE to different scopes (7.3), different processors and processor flags (7.4), and other types of prefetching (7.5). Finally, we discuss PREFENCE in the context of prefetching-based covert channels (7.6).

7.1. Applicability to Hardware Prefetcher Types

When PREFENCE is to be applied to real-world programs, the question arises which code sections should be wrapped in PREFENCE system calls. Naturally, applying PREFENCE to a whole process (coarse-grained) is an easy-to-implement and trivially secure option. However, it is not the most performant one. For more fine-grained use of PREFENCE, the leaking code sections need to be identified and wrapped in PREFENCE system calls. As we found in Section 6.6, this additional effort at development time is rewarded with better performance at runtime.

When PREFENCE is to be used in a more fine-grained manner, the system call locations depend on the prefetcher types that are expected to be present in the target system. More precisely, we need to distinguish between systems with only address-based prefetchers and systems also featuring DMPs.

Address-Based Prefetchers. If only address-based prefetchers are present, it is sufficient to apply PREFENCE to code regions that operate on secret-dependent addresses. We envision that automated taint tracking can facilitate this process, as we discuss in Section 7.2.

DMPs. To also protect against side channels introduced by DMPs, PREFENCE system calls need to be placed more conservatively to account for data-dependent predictions as well. It is intuitive to place system calls around code that operates on secret-dependent values directly. Otherwise, a DMP may activate and operate on the secret-dependent values, potentially leaking them to an attacker. In addition, however, we need to consider cases where a DMP is triggered to operate on secret-dependent values by a memory operation on non-secret values. We consider two such scenarios: (i) adjacent buffers and (ii) multiple levels of pointer indirection.

The first scenario illustrates the problem of adjacent buffers: Consider a non-secret buffer and a secret buffer, located consecutively in memory. When non-critical code traverses the non-secret buffer, the DMP may be activated and continue prefetching beyond the buffer's bounds into the secret buffer. The DMP may then interpret secret-dependent data as a pointer, dereference it, and encode secret-dependent information into the cache state. Thus, memory operations on adjacent buffers should also be wrapped in PREFENCE system calls. While this is a non-trivial task to perform manually, we discuss how this step could be automated in Section 7.2.

The second scenario occurs with DMP implementations that prefetch pointers multiple levels of indirection deep. In this case, an attacker could craft a pointer into security-critical memory and dereference it to trigger the DMP. If the DMP then proceeds to dereference a secret value as well, this action may leave traces in the cache. In this scenario, any pointer dereference in the application code could be exploited to trigger the prefetcher on a security-critical buffer. Thus, PREFENCE should be applied to the whole security-critical process in this case. However, to the best of our knowledge, no current DMP implementation (neither Apple nor Intel) follows pointers multiple layers of indirection deep.

7.2. Automating System Call Placement

Our approach depends on system calls being placed before and after security-critical code sections. While the manual effort required to apply PREFENCE to a whole process (coarse-grained use) is low, placing system calls at finer granularity is expected to cause higher effort.

To reduce the manual effort required for fine-grained use of PREFENCE, we envision that prior research on taint tracking can be leveraged to place system calls automatically. For example, the approach of CryptoMPK [22] could be adapted. CryptoMPK aims to automatically identify code that operates on buffers containing secret or secret-dependent values. Such code is then surrounded with instructions that switch memory protection keys (MPKs). These additional instructions could also be our PREFENCE system calls.

With CryptoMPK, developers annotate buffers that contain *initial secrets* (e.g., keys), which are easier to identify than security-critical code. CryptoMPK then tracks the use of those buffers. It identifies additional secret-dependent buffers as well as all memory operations operating on secret or secret-dependent buffers.

To protect against leaks from address-based prefetchers, CryptoMPK could place the PREFENCE system calls before and after such memory operations. In addition, CryptoMPK could be extended to facilitate fine-grained system call placement for DMPs: To eliminate the problem of a DMP being triggered on adjacent buffers (as

discussed in Section 7.1), such adjacent buffers could be treated as security-critical as well. Consequently, operations on those buffers would also be protected with PREFENCE system calls, preventing prefetcher activation.

7.3. Applicability to Different Scopes

Our current implementation of PREFENCE, as presented in Section 5.3, protects a user-space process from attacks by other user-space processes (scopes SP, CT, and CP, as introduced in Section 4.1). This covers 11 out of the 13 attacks that we discussed in Section 4. The general principle of PREFENCE can also be used to protect kernel code from user-space attacks (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 security-critical code is executed in trusted execution. However, because the untrusted operating system can usually interrupt trusted execution [24], [39], [49], 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. For Intel SGX, which is interrupt-unaware [31], [49], this requires additional hardware support.

7.4. Applicability to Other Processors And Flags

Other Processors. In this paper, we implement and evaluate PREFENCE for two specific CPUs. These CPUs represent popular attack targets (such as the Intel IP stride prefetcher [10], [44], [51]) 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 [42]. We are confident that the design of PREFENCE is general enough to be transferred to other processors as well, as long as those fulfill two basic requirements. First, the processor must expose a way to disable the relevant prefetchers dynamically at runtime, e.g., by setting a bit in an MSR. Second, the prefetcher must not update its state while disabled.

To get an overview of how widespread processor support for MSRs controlling the hardware prefetchers is, we examined a large corpus of technical reference manuals from Intel, AMD and ARM. We report our detailed findings in Appendix C. Our results indicate that most processors provide a way to control hardware prefetchers, but the relevant MSRs are not unified, neither across vendors nor for processors of the same vendor. This strengthens our position that it is infeasible for user-space applications to interact with the prefetcher directly; instead, the operating system kernel should abstract from implementation-dependent interfaces and provide a unified and easy-to-use way for user-space applications to control hardware prefetchers. The design of PREFENCE allows for this architecture. The Intel XPT prefetcher is an example for a prefetcher that cannot be controlled dynamically, as Intel does not publicly disclose any way to control it [19]. In such cases, PREFENCE cannot be applied.

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* [2], [5] and the *Data Operand Independent Timing Mode (DOITM) flag* [18], respectively. The processors concerned 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 [8], [18]). PREFENCE could be adapted to dynamically set and clear these (or other) flags as well.

7.5. Applicability to Other Prefetching Types

We focus on mitigating vulnerabilities caused by hardware prefetching *on data* in this paper. As prior work has shown, prefetching *on instructions* can also be exploited to leak information [55]. In principle, PREFENCE can also be applied to this kind of prefetching, as long as the prefetcher can be controlled. In fact, our implementations for both architectures also disable the instruction prefetchers when the *prefetch_disable* flag is set. However, when PREFENCE is applied in a very fine-grained manner, i.e., only wrapping carefully selected code sections, the selection of code sections may need to be reassessed with instruction prefetching in mind to make PREFENCE fully effective against instruction prefetching attacks. This is because code sections vulnerable to attacks on instruction prefetching might be different from those that are vulnerable to data prefetching attacks.

Another kind of prefetching is *software prefetching*. While software prefetching may also introduce vulnerabilities [13], [26], those vulnerabilities are not caused by automated prediction mechanisms. Instead, they concern explicit prefetching instructions. Thus, we consider software prefetching vulnerabilities an orthogonal problem that is out of the scope of this paper.

7.6. Covert Channels

In this paper, we focus on mitigating prefetching side channels and exclude covert channels from the scope. 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. [10] exploit the same prefetcher but encode the information to transmit into the stride. Rohan et al. [38] 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. [42] encode bit vectors into the region-based ARM SMS prefetcher to transfer information from trusted execution to the untrusted OS. Chen et al. [9] implement two covert channels based on the Intel XPT prefetcher. For the first channel, the receiver primes the

prefetcher's state and the sender either idles or changes the state by evicting one of the primed entries. For the second channel, the sender either trains or resets the prefetcher for a shared page.

All of these attacks have in common that it is the attacker, not the victim, who 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 in this case.

8. Related Work

In this section, we provide an overview of software- and hardware-based mitigations against prefetching-based attacks that were proposed in prior work, and we relate them to our approach.

8.1. Prior Work on Software-Based Mitigations

We start by discussing mitigations from prior work that can be applied via (modified) software and argue that none of them prevent attacks in an easy and efficient way.

Constant-Time Programming [10], [42], [44], [51]. One way to prevent most prefetching-based side channels, and also other cache-timing side channels, is the programming technique of *constant-time programming*. Despite the name, it not only refers to writing code that executes in the same time regardless of the (potentially secret) information processed, but it also mandates that no secret-dependent control flow or memory access patterns occur [17]. 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 prefetching-based attacks that infer the victim's control flow based on conditionally executed load instructions. However, constant-time programming is ineffective against attacks on DMP prefetchers that exploit secret-dependent *values* instead of addresses or branches [8]. In addition, making code constant-time requires complex re-writes and results in significantly reduced performance [7], [35].

Clearing the Prefetcher's State on Context Switches [10], [12], [42]. Some prefetching-based attacks train the prefetcher in a victim context, then switch into the attacker's context and trigger it there. Relevant context switches are transitions between two user-space processes, transitions between kernel and user space, and returns from trusted execution. To mitigate such attacks, the prefetcher's state could be cleared on context switches. This is straightforward to implement if the CPU provides a suitable instruction, such as `CPP RCTX` on some ARM CPUs [4], [5]. Otherwise, all stored patterns need to be evicted, which is computationally expensive and 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's state 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 [24], [39], [49]. 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 [10], [44]. We found in Section 4.3 that all prefetching attacks rely on cache-timing side channels. 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 [11], [50]. However, attackers may fall back to alternatives such as a counter thread as a timer replacement [27]. General countermeasures against cache-based side channels have been discussed in prior work extensively [28], [33], [52], [54], 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 [10]. During a prefetching-based side channel attack, the attacker may execute code that results in unusual values of performance counters. For instance, counters related to cache activity or prefetch-related events may increase at a higher rate than usual. These counters can thus be observed to detect unusual activities [15], [34], [53]. However, such a heuristic detection system will produce false-positive alerts, miss some 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 after they have started. Thus, we consider this mitigation strategy incomplete.

Security-Aware Core Assignment [8], [40]. For per-core prefetchers, another possible mitigation is more advanced core assignment. On heterogeneous processors, a vulnerable prefetcher may only be present on some of the cores. In this case, security-critical or untrusted workloads could be assigned to invulnerable cores. Similarly, a vulnerable per-core prefetcher could be disabled on one of the cores, which could then be reserved for critical workloads. In practice, however, it is not trivial to decide which processes should be assigned to which core. In addition, reserving a core for critical operations is likely to reduce overall system performance: 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 [10]. Oblivious execution [36] eliminates the side-channel effect of a secret-dependent conditional. It executes both branches but persists only the result of the correct branch in memory. Thus, the attacker can no longer distinguish those branches: The resulting timings and memory access patterns are always the same (as long as the branches do not contain more secret-dependent instructions). However, this approach comes with a significant performance overhead. Rane et al. [36] report a mean overhead of $16.1\times$.

Blinding [8]. A DMP may be triggered by a data value that has a specific property, 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 computational and memory overhead.

Disabling the Prefetcher [8]–[10], [12], [42], [44]. If the CPU allows for controlling the prefetcher, the most straightforward way to prevent any leakage from the prefetcher is to disable it permanently. However, this countermeasure comes with a significant performance decline for workloads that benefit from prefetching, as we show in Sections 6.5 and 6.6.

The general idea of disabling the prefetcher temporarily has been briefly mentioned in the context of offensive papers before [8], [42], but it has not been elaborated further. No detailed design, implementation, or evaluation has been provided. In particular, prior work did not make any considerations beyond the "naive approach", which we dismissed as incomplete in Section 5.1. Notably, no integration with the scheduler was proposed, which is vital to make the defense complete and easy to use. In addition, the special case of SMT was not considered. PREFENCE fills this gap.

Conclusion: Prior-Work Countermeasures Are Costly or Incomplete. In summary, every prior-work countermeasure violates one of our design goals stated in Section 5.1. 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 the state is shared across physical or SMT cores; *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.

8.2. Prior Work on Hardware-Based Mitigations

In this paper, we focus on software-based mitigations that can easily be applied and evaluated on current hardware. However, for completeness, we also briefly discuss countermeasures that require hardware modifications.

Choosing a different trigger [42]. Some attacks rely on collisions on the prefetch trigger, e.g., the (partial or complete) address of a load instruction. To mitigate such attacks, 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 [9], [10], [12], [42]. Partitioning protects against accidentally triggering a prefetch pattern in a wrong context. To avoid leakage between privilege levels, the prefetcher could keep track of the privilege level that a pattern belongs to. However, this approach does not protect against attacks within the same privilege level, e.g., between two user-space processes. To distinguish between those, an additional process identifier must be stored. 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 [40], [42]. An instruction could be added that flushes the prefetcher's state, such as the `CPP RCTX` instruction available on some ARM CPUs [4], [5]. 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. Alternatively, a special load instruction to be used in security-critical code sections could be introduced 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 prefetchers to leak secret information from a victim user-space process. We started by providing the first systematic analysis of the existing 13 related attacks from literature, and we showed that all 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 user space, while ensuring that parallel processes sharing the same prefetcher on other cores or SMT siblings are considered as well. In addition, issues in process re-scheduling 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 an Intel x86_64 and an ARM Cortex-A72 processor is a Linux kernel patch to the scheduler that automatically ensures correct prefetcher activation state on process re-scheduling. In addition, it provides processes with a system call to request the prefetcher to be disabled. Our PREFENCE implementation is open-source software.

We demonstrated the efficacy of our approach by successfully mitigating two prior-work side-channel vulnerabilities with low overhead (less than 3%) on the vulnerable functions. Our performance evaluation showed that the performance impact of our solution on non-security-critical code is around 1%. For security-critical workloads, its performance impact depends on the way PREFENCE is applied to the code; for a real-world web server application, we showed that security-critical code runs 15.8% (Intel) and 7.2% (ARM) faster on average compared to prefetching being permanently disabled when PREFENCE is applied in a relatively fine-grained manner.

In conclusion, we presented an easy-to-use and efficient scheduling-aware countermeasure to protect victim processes against prefetcher side channels, founded on a systematic analysis of prior work on attacks and countermeasures. We expect our countermeasure could be extended to the general signaling of security-relevant code to the kernel to allow for coordinated application of countermeasures (e.g., DIT flags).

References

[1] Advanced Micro Devices, Inc. A new x86 core architecture for the next generation of computing. HotChips 28, August 2016. https://old.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.930-X86-core-MikeClark-AMD-final_v2-28.pdf.

[2] Apple Inc. Writing ARM64 code for Apple platforms, 2024. https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms.

[3] Arm Ltd. ARM® Cortex®-A72 MPCore processor technical reference manual, December 2016. https://developer.arm.com/documentation/100095/0003/.

[4] Arm Ltd. Architecture Security Advisory: Prefetcher side channels, 2023. https://developer.arm.com/documentation/109504/0100/.

[5] Arm Ltd. Arm® architecture registers for A-profile architecture, 2024. https://developer.arm.com/documentation/ddi0601/2024-03.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the USENIX Security Symposium*, 2019.

[7] Sunjay Cauligi, Craig Disselkoen, Klaus V. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[8] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *Proceedings of the USENIX Security Symposium*, 2024.

[9] Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E. Carlson. PREFETCHX: Cross-core cache-agnostic prefetcher-based side-channel attacks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2024.

[10] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[11] Chromium Project. Mitigating side-channel attacks, 2018. https://www.chromium.org/Home/chromium-security/ssca/.

[12] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2019.

[13] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, July 2016.

[15] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning, 2019. https://arxiv.org/abs/1907.03651.

[16] Intel Corp. Data dependent prefetcher, November 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html.

[17] Intel Corp. Guidelines for mitigating timing side channels against cryptographic implementations, June 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[18] Intel Corp. Data operand independent timing instruction set architecture (ISA) guidance, February 2023. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[19] Intel Corp. Hardware LLC prefetch feature on 4th gen Intel® Xeon® scalable processor (codename Sapphire Rapids), June 2023. https://www.intel.com/content/www/us/en/content-details/780991/.

[20] Intel Corp. Hardware prefetch control for Intel® Atom® cores. Whitepaper, December 2023. https://cdrdv2-public.intel.com/795247/357930-Hardware-Prefetch-Controls-for-Intel-Atom-Cores.pdf.

[21] Intel Corp. Intel® 64 and IA-32 architectures software developer's manual, combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, June 2024. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[22] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[23] 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. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[24] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-Step: A precise TrustZone execution control framework for exploring new side-channel attacks like Flush+Evict. In *ACM/IEEE Design Automation Conference (DAC)*, 2021.

[25] Lighttpd Developers. 1.4.75 - Lighttpd, 2024. https://www.lighttpd.net/2024/3/13/1.4.75/.

[26] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *Proceedings of the USENIX Security Symposium*, 2022.

[27] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the USENIX Security Symposium*, 2016.

[28] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[29] LLVM Project. Speculative load hardening — LLVM 18.0.0git documentation, January 2024. https://llvm.org/docs/SpeculativeLoadHardening.html.

[30] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[31] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[32] Matt Miller. Mitigating speculative execution side channel hardware vulnerabilities. Microsoft Security Response Center (MSRC) Blog, March 2018. https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology (CT-RSA)*, 2006.

[34] Mathias Payer. HexPADS: A platform to detect "stealth" attacks. In *Engineering Secure Software and Systems (ESSoS)*, 2016.

[35] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the USENIX Security Symposium*, 2015.

[37] Raspberry Pi Ltd. Raspberry Pi documentation - The Linux kernel, 2024. https://www.raspberrypi.com/documentation/computers/linux_kernel.html.

[38] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse engineering the stream prefetcher for profit. In *IEEE European Symposium on Security and Privacy (EuroS&P) Workshops*, 2020.

[39] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[40] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[41] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. FetchBench code artifact, 2023. https://github.com/scy-phy/FetchBench.

[42] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. FetchBench: Systematic identification and characterization of proprietary prefetchers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2023.

[43] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. Specfuscator: Evaluating branch removal as a Spectre mitigation. In *Financial Cryptography and Data Security*, 2021.

[44] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[45] Standard Performance Evaluation Corporation (SPEC). runcpu - using CPU 2017, 2019. https://www.spec.org/cpu2017/Docs/runcpu.html#iter.

[46] Standard Performance Evaluation Corporation (SPEC). SPEC CPU® 2017, 2022. https://www.spec.org/cpu2017/.

[47] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, 4th edition, 2015.

[48] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.

[49] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[50] Luke Wagner. Mitigations landing for new class of timing attack. Mozilla Security Blog, January 2018. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack.

[51] Chong Xiao, Ming Tang, and Sylvain Guilley. Exploiting the microarchitectural leakage of prefetching activities for side-channel attacks. *Journal of Systems Architecture*, 139, June 2023.

[52] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the USENIX Security Symposium*, 2014.

[53] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 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)*, 2016.

[54] Yinqian Zhang and Michael K. Reiter. 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)*, 2013.

[55] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the instruction prefetcher. In *Proceedings of the USENIX Security Symposium*, 2023.

Appendix A.
Prior Work on 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. [44]. 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 in the OpenSSL shared library code that are cached only conditionally depending on the supplied input. These cache lines only appear in the 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 [40]. This paper is the first to investigate the data memory-dependent prefetcher (DMP) of the Apple M1 SoC. It describes the prefetcher's behavior when multiple pointers, which 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) [29] to protect against Spectre attacks [23]. 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 [10]. This work exploits the Intel IP stride prefetcher in 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 same-process 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 space. The authors attack a system call handler that operates on a memory buffer passed in from user space. The idea is similar to variant 1. However, aligning to a kernel instruction 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 from a shared buffer that is passed from user space into the enclave. The stride of the loads is secret-dependent. This memory activity trains (S3) and triggers (S4) the prefetcher. After returning to user space, 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 monitor. 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 prefetching-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. [51]. 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 [42]. 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 [9]. 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 is 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 [8]. 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.

Appendix 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. We evaluate the fixed overhead that our PREFENCE implementation adds to every context switch. To this end, we measure the execution time of a context switch in the stock kernel and compare it to the execution time in our patched kernel.

We implement two user-space processes that share a memory page. Both processes are pinned to the same CPU core. The first process constantly writes the current value of a high-resolution timer (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 9. 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 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.

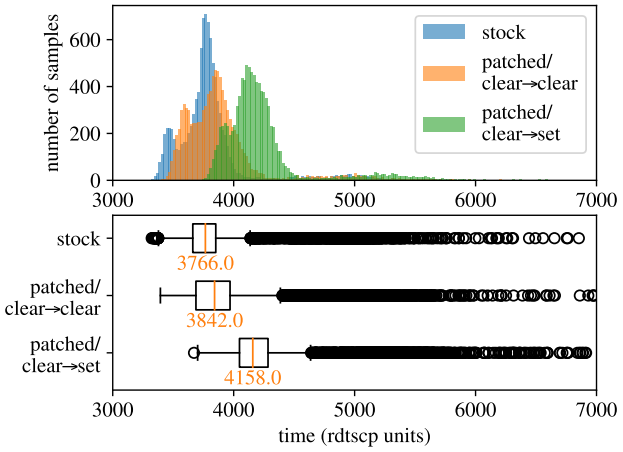


Figure 9. Context switch overhead on the stock kernel and on our patched kernel. For the patched kernel, we evaluate transitions between normal processes (*prefetch_disable* bits cleared) and the transition into a security-critical process (from bit cleared to bit set). The added overhead is negligible.

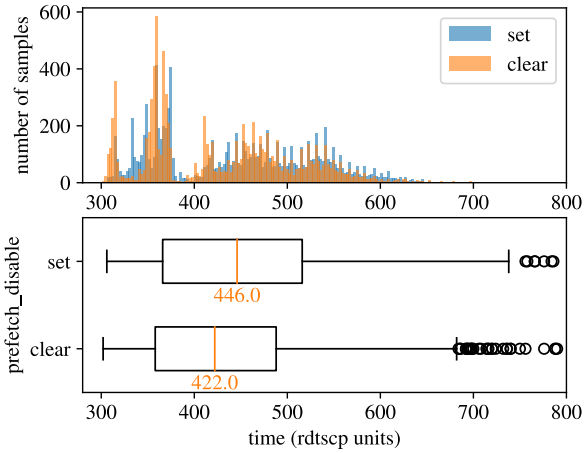


Figure 10. Overhead of a system call that sets or clears the *prefetch_disable* bit.

B.2. One-off Overhead of the System Call

Experiment. We set up a user-space 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, 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. Figure 10 shows the results. We note that the median duration of 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).

Appendix C. Prefetch Disable Flags on Various Processors

In Tables 2 to 4, we list all MSR flags related to disabling hardware prefetchers that we could find in processor documentation provided by Intel, AMD and ARM, respectively. For Intel, we considered the latest *Software Developer's Manual* (June 2024) [21] as well as additional documentation [16], [18], [20]. For AMD, we considered all *BIOS and Kernel Developer's Guides* and *Processor Programming References* available from the AMD Documentation Hub[1] (as of July 2024). For ARM, we considered all *Technical Reference Manuals* available from ARM's documentation page[2] starting from the Cortex-A53 (as of July 2024). We note that we only include officially documented disable flags here, as those are the flags that operating system vendors could safely rely on; processors may implement additional (undocumented) flags.

1. https://www.amd.com/en/search/documentation/hub.html
2. https://developer.arm.com/documentation/

We find that the interfaces for controlling hardware prefetchers differ widely between processors, even for models from the same vendor. Thus, abstraction at the operating system kernel level is required to provide a unified interface to user-space processes.

Appendix D. Statement on Data Availability

Alongside this paper, we publish an artifact package that contains the proof-of-concept implementations of PREFENCE used in this paper (suitable for some Intel and ARM processors), as well as the code to run the experiments in Section 6 and Appendix B. For license reasons, we cannot include the source code of SPEC benchmarks (used in Section 6.5). However, we provide the configuration files that we used to run those benchmarks, so that other researchers can still reproduce those experiments with their own copy of SPEC benchmarks.

Our artifact package is available at `https://github.com/scy-phy/PreFence`.

TABLE 2. PREFETCH DISABLE FLAGS ON INTEL PROCESSORS

Microarchitecture	Model-specific registers, bits
NetBurst	IA32_MISC_ENABLE[9,19]
Core	IA32_MISC_ENABLE[9,19,37,39]
Nehalem	MSR_MISC_FEATURE_CONTROL[0:3]
Sandy Bridge	MSR_MISC_FEATURE_CONTROL[0:3]
Ivy Bridge	MSR_MISC_FEATURE_CONTROL[0:3]
Haswell, Haswell-E	MSR_MISC_FEATURE_CONTROL[0:3]
Broadwell	MSR_MISC_FEATURE_CONTROL[0:3]
Skylake	MSR_MISC_FEATURE_CONTROL[0:3]
Cascade Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Copper Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Caby Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Coffee Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Cannon Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Comet Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Ice Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Tiger Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Alder Lake P-Core	MSR_PREFETCH_CONTROL[0:3,5]
Alder Lake E-Core	MSR_0x1A4[0,2:5], MSR_0x1320[43], MSR_0x1321[43]
Raptor Lake P-Core	MSR_PREFETCH_CONTROL[0:3,5]
Raptor Lake E-Core	MSR_0x1A4[0,2:5], MSR_0x1320[43], MSR_0x1321[43]
Meteor Lake (Core 7 Ultra)	MSR_PREFETCH_CONTROL[0:8]
Xeon Phi Processors 06_57H/06_85H	MSR_PREFETCH_CONTROL[0:1]
Silvermont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Goldmont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Goldmont Plus (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Tremont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
CPUs with CPUID.(EAX=0x07,ECX=2):EDX[3]=1	IA32_SPEC_CTRL[8] (DDPD_U)
CPUs with CPUID.(EAX=0x07,ECX=0):EDX[29]=1	IA32_UARCH_MISC_CTL[12] (DOITM)

TABLE 3. PREFETCH DISABLE FLAGS ON AMD PROCESSORS

Processor family	Model-specific registers, bits
Family 10h (K10)	MSRC001_1022[13]
Family 11h (K8/K10)	MSRC001_1022[13]
Family 12h (K10)	MSRC001_1022[13]
Family 14h Models 00h-0Fh (Bobcat)	MSRC001_1022[13]
Family 15h Models 00h-0Fh (Bulldozer/Piledriver)	MSRC001_1022[13]
Family 15h Models 10h-1Fh (Piledriver)	MSRC001_1022[13]
Family 15h Models 30h-3Fh (Steamroller)	MSRC001_1022[13]
Family 15h Models 60h-6Fh (Excavator)	MSRC001_101C[23], MSRC001_102B[18]
Family 15h Models 70h-7Fh (Excavator)	MSRC001_101C[23], MSRC001_102B[18]
Family 16h Models 00h-0Fh (Jaguar)	MSRC001_1022[13], MSRC001_10A0[7]
Family 16h Models 30h-3Fh (Puma)	MSRC001_1022[13], MSRC001_10A0[7]
Family 17h Models 01h, 08h, Revision B2 (Zen/Zen+)	—
Family 17h Model 18h, Revision B1 (Zen+)	—
Family 17h Model 20h, Revision A1 (Zen)	—
Family 17h Model 31h, Revision B0 (Zen 2)	—
Family 17h Model 60h, Revision A1 (Zen 2)	—
Family 17h Model 71h, Revision B0 (Zen 2)	—
Family 17h Model A0h, Revision A0 (Zen 2)	—
Family 19h Model 01h, Revision B1 (Zen 3)	MSRC000_0108[0:3,5]
Family 19h Model 11h, Revision B1 (Zen 4)	MSRC000_0108[0:3,5]
Family 19h Model 21h, Revision B0 (Zen 3)	—
Family 19h Model 51h, Revision A1 (?)	—
Family 19h Model 61h, Revision B1 (Zen 4)	MSRC000_0108[0:3,5]
Family 19h Model 70h, Revision A0 (Zen 4)	MSRC000_0108[0:3,5]

—: No documented prefetch disable flags.

TABLE 4. PREFETCH DISABLE FLAGS ON ARM PROCESSORS

Processor	Model-specific registers, bits
Cortex-A53	CPUACTLR_EL1[13:15,22]
Cortex-A57	CPUACTLR_EL1[21,32,56]
Cortex-A72	CPUACTLR_EL1[21,32,42,56]
Cortex-A73	L2CTLR_EL1[21], ECTLR[7,8,10]
Cortex-A75	CPUECTLR[6:10]
Cortex-A76	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A76AE	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A77	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A78	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A78C	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A510	AArch64_imp_cpuectlr_el1[13:14], AArch64_imp_cmpxectlr_el1[26]
Cortex-A520	AArch64_imp_cpuectlr_el1[13:14], AArch64_imp_cmpxectlr_el1[26]
Cortex-A710	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A715	AArch64_imp_cpuectlr_el1[12,41:52], IMP_CPUECTLR2_EL1[8:9]
Cortex-A720	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Cortex-X1	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X1C	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X2	IMP_CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X3	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Cortex-X4	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Cortex-X725	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Cortex-X925	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Neoverse E1	CPUACTLR_EL1[10:15,22]
Neoverse N1	CPUECTLR_EL1[5,8,15,51]
Neoverse N2	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Neoverse N3	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Neoverse V1	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Neoverse V2	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Neoverse V3	CPUECTLR_EL1[4:5,8,9,15,51,61:63]