ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs

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

Michael Torres
tmux@google.com
Google LLC
Mountain View, California, USA

Daniel Moghimi
danielmm@google.com
Google LLC
Mountain View, California, USA

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

Abstract

Microarchitectural attacks and reverse-engineering efforts rely on inferring the cache state of cache lines. While high-resolution timers traditionally enable this, such timers are increasingly restricted or unavailable to unprivileged users on modern ARM64 systems.

We introduce a fuzzing-based methodology to automatically discover instruction sequences that leak cache state into architectural state—without timing measurements. Our proof-of-concept, ExfilState, uses differential testing, F-score ranking, and covert-channel verification to identify architectural side channels on ARM64 CPUs. Across 160 devices with 37 microarchitectures—including smartphones, laptops, and cloud servers—ExfilState uncovers 5 undocumented side channels, 2 of which are reliably and widely exploitable.

We demonstrate their practical impact with a timer-free Spectre variant, a cache-based AES key-recovery attack, and a novel defense mechanism that aborts sensitive algorithms on eviction of victim cache lines. Our findings show that architectural side channels are both real and exploitable, even in environments without timers, broadening the attack surface on modern ARM64 platforms.

CCS Concepts

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

Keywords

Microarchitecture; Architectural Side Channels; CPU Fuzzing

ACM Reference Format:
Fabian Thomas, Michael Torres, Daniel Moghimi, and Michael Schwarz. 2025. ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3719027.3765061

1 Introduction

Modern processors operate on two levels of state: the architectural state, directly defined by the instruction set architecture (ISA), and the microarchitectural state, comprising internal optimizations such as caches and branch predictors. Traditionally, the separation between these two states ensures that microarchitectural optimizations do not interfere with the predictable behavior of software execution as defined by the ISA. However, the microarchitecture is not entirely invisible. For instance, variations in cache states can affect instruction execution times, which, when measured using a high-resolution timer, can leak information about the microarchitectural state. If the microarchitectural state depends on a secret, an attacker can exploit such a timing measurement to infer the secret using, e.g., a cache attack such as Flush+Reload [77] or Prime+Probe [48]. Notably, transient-execution attacks such as Spectre [33] also rely on such side channels to transmit secrets encoded in the microarchitectural state to the architectural state [9].

Due to such attacks, access to high-resolution timers has been restricted [37, 38, 57, 58] or are entirely unavailable for unprivileged users [24, 31]. On ARM, the ISA provides the `CNTEL0ACR` control register restricting access to high-resolution counters for unprivileged users [3]. The unavailability of high-resolution counters on many devices, and the diversity of microarchitectures in the ARM ecosystem, make it challenging to build reliable and portable timer-based attacks, such as cache attacks or Spectre attacks [24]. In addition to impeding attacks, it hinders microarchitectural research in general. For example, distinguishing cache hits from misses is a generic primitive often used for reverse engineering microarchitectural properties [11, 25, 30, 49, 55, 66, 67, 81].

In this paper, we ask the following research question:

Can we automatically discover instruction sequences that leak microarchitectural cache state into architectural state, without relying on timing measurements?

We answer this question in the affirmative by developing a novel fuzzing methodology to automatically discover *architectural side channels*: instruction sequences that make cache state visible at the architectural level (e.g., register values or exceptions) without relying on timing measurement. Our approach leverages differential testing to identify discrepancies in architectural states arising from variations in cache states. By systematically crafting distinct cache states through controlled memory operations, i.e., memory accesses and cache flushes, we execute identical instruction sequences across these states and observe the resulting architectural outputs. Disparities in these outputs indicate potential architectural side channels. We focus on cache states, as the cache is a widely-used microarchitectural element in microarchitectural research and attacks [33, 38, 48]. Moreover, the cache provides reliable means for modifying its state, i.e., via accessing and flushing data. We focus on instruction sequences that interact with the memory subsystem, as these are more likely to be influenced by cache states and other microarchitectural elements.

To manage the large number of findings and prioritize those with significant security implications, we use the F-score to evaluate the severity of identified leaks. This metric is commonly used for evaluating side-channel primitives [23, 56, 71–73]. We randomly cache or flush a memory address and calculate the F-score based on the precision and recall of detecting the cache state—i.e., whether the respective cache line resides in the cache—with the discovered primitive. A high F-score indicates that we do not accidentally measure unrelated events, such as frequency scaling [69] or speculative execution [33], but effects that are either caused by the cache state or correlate with the cache state. We rank the results based on their reliability for determining the cache state.

We additionally incorporate a verification stage in our methodology, recognizing that some identified side channels may not manifest in real-world scenarios due to factors like prefetching and speculative execution. In this stage, we evaluate the stability of an automatically synthesized covert channel. For this covert channel, we transmit 128 bytes via the cache and measure the bit-error rate of the transmission. This systematic evaluation enables early rejection of non-deterministic results and false positives, streamlining the analysis process to focus on the most reliable side channels.

We implement our proposed methodology in a proof-of-concept tool, ExfilState (Exfiltrate microarchitectural state), that automates the detection, evaluation, and verification of cache state leakage on ARM64. ExfilState continuously repeats 4 stages: case generation and difference discovery, correlation analysis and reduction, verification, and clustering. This structured approach ensures comprehensive coverage and systematic analysis of potential side channels. Although the methodology is generic, we implement ExfilState for ARM64, as it is used in a wide range of devices, including smartphones, laptops, and servers, and the market share is growing rapidly. In contrast to x86, high-resolution timers are not available on all CPUs [24] or are restricted to privileged code [3]. Moreover, the reduced number of instructions interacting with memory enables a more systematic evaluation and better coverage.

We demonstrate the efficacy of ExfilState across 160 devices with 37 unique microarchitectures, resulting in the discovery of 5 novel side channels. Our devices include low- to high-end smartphones, laptops, desktops, and cloud servers. We find 2 architectural side channels that work reliably on most ARM CPUs and 3 side channels that work on a subset of the microarchitectures. All side channels work in unprivileged native and virtualized environments. A manual analysis of the 2 widely available side channels shows the two main effects leading to them are exclusive loads and stores, and incoherence of data and instruction cache. The first category, Lx+Sx, exploits "incorrect" usage of exclusive store and load instructions, resulting in architectural "error codes" depending on the cache state. The second category, Store+Ret, exploits a microarchitectural race condition when executing overwritten code. Because ARM does not guarantee automatic coherence between data and instruction cache, the new instruction is only executed when the cache line was not in the cache before. While these side channels exist on most tested CPUs, they often require subtle changes in the instruction sequence, justifying the use of a fuzzing-based approach for discovering them.

To show real-world applicability, we demonstrate 3 case studies. For comparability with other side channels, we mount a classical attack on AES T-tables on multiple devices and microarchitectures, including Cortex-A72 (Raspberry Pi 4/Marvell Armada 7040) and Cortex-A720 (Pixel 9). Our results show nearly perfect key recovery, outperforming Flush+Reload on the Marvell Armada 7040 due to low timer resolution for unprivileged attackers. We demonstrate that architectural side channels also enable *defenses* against cache attacks by redirecting control flow on cache misses. We demonstrate this allows programs to be protected similarly to x86-based defenses relying on Intel TSX [22] or CPU exceptions [72]. Finally, we use the architectural side channels to build a Spectral attack [80], i.e., a Spectre attack with architectural leakage. Our Spectral attack, based on Spectre-PHT and Lx+Sx, leaks 11 457.9 B/s, outperforming existing Spectre attacks on x86. These results emphasize the versatility and robustness of our approach in identifying practical side-channel primitives across various microarchitectures.

Contributions. The main contributions of this work are:

(1) We introduce a fuzzing-based approach to systematically identify instruction sequences causing cache state leakage to the architectural state without timing measurement.
(2) We present ExfilState, a proof-of-concept implementation for ARM64. ExfilState identifies 5 previously unknown architectural side channels related to cache state leakage on 36 unique microarchitectures, including the AWS Graviton4 and the Qualcomm Snapdragon X Elite.
(3) We utilize the F-score of the side-channel primitive and an automatically synthesized covert channel to rank and cluster detected leaks, prioritizing side channels based on their exploitability and impact. The covert-channel-based verification stage ensures real-world applicability of our findings.
(4) We demonstrate the efficacy of our architectural side channels in two attacks, showing timer-less Spectre and AES key recovery on different devices. Additionally, we use the side channels for demonstrating a defense on AES T-tables implementations that aborts the encryption if a cache miss is detected, similar to existing defenses on x86 [22, 72].

Outline. The remainder of this paper is organized as follows. Section 2 provides the necessary background. Section 3 outlines our methodology. Section 4 presents our implementation, ExfilState. Section 5 evaluates ExfilState and summarizes results of our fuzzing campaign. Section 6 examines the 5 side channels in detail. In Section 7 we illustrate their use through three case studies. Section 8 considers mitigations, limitations, and related work. Finally, Section 9 concludes the paper.

Responsible Disclosure. We reported our findings to Arm. Arm acknowledged our findings but does not plan any mitigations.

Availability. ExfilState and all related artifacts are open-source and available at: https://github.com/cispa/ExfilState-artifacts.

2 Background

In this section, we provide background on ARM64 CPU internals, cache attacks, and fuzzing methodologies.

2.1 CPU Internals

Modern ARM64 CPUs rely on hierarchical memory and cache structures and optimization features to bridge the performance gap between the processor and main memory. They typically feature separate L1 instruction and data caches and shared or private L2 caches [74, 75]. Some high-end cores also employ 3 levels of caches, where the last level is shared [75]. The higher-end ARM64 CPUs, similar to x86, also employ advanced features such as speculative execution, branch prediction, cache prefetching, and out-of-order execution to enhance throughput and reduce latency [75]. The specific features vary between cores and manufacturers.

Memory Load and Store. As ARM64 is a reduced instruction set computer (RISC) architecture, there is a limited number of instructions that can load from and store to memory. Thus, in contrast to x86, most instructions cannot use a memory address as an operand but require a register. ARM64 additionally provides specific instructions for concurrency control, including locked and atomic load/store instructions (e.g., `LDAXR`, `STLXR`).

Memory Coherence. x86 enforces strong memory coherence through Total Store Order (TSO), ensuring writes are immediately visible across cores [42]. In contrast, ARM64 uses a weaker model requiring explicit memory barriers (e.g., DMB, DSB, ISB) for ordering, which necessitates more careful handling in multithreaded software but enables greater efficiency and power savings [3].

2.2 Cache Attacks

Cache attacks exploit timing differences resulting from cache hits versus misses to leak memory-access patterns of victim processes. Two well-established techniques are Flush+Reload [77] and Prime+Probe [48]. While these techniques were first shown on x86, they have also been demonstrated on ARM CPUs [38]. Flush+Reload relies on shared memory between attacker and victim, using the `DC CIVAC` instruction on ARM to evict cache lines, then measuring reload latency to infer victim activity. Prime+Probe does not require shared memory; it involves filling cache sets (prime), scheduling the victim, and subsequently measuring access times (probe) to the attacker's own addresses to detect if victim activities have evicted the attacker's cached data.

Timers. Traditionally, accurate timing measurements are critical for cache attacks. Attackers typically use high-resolution timers such as the CPU's timestamp counter (e.g., `PMCCNTR_EL0`) or loop-based timers if direct access to hardware timers is restricted [38]. Restricting timer precision can raise the bar for attacks, but prior work noted that this alone may not be a sufficient defense [40].

Amplification. In particular, attackers can amplify timing differences, for example by duplicating the microarchitectural state [32] or exploiting cache eviction strategies [51, 61].

Architectural Side Channels. On x86, specific architectural features, such as the `mwait` instruction used for optimized power management [80] or transactional memory aborts triggered via Intel TSX [13], introduce architecturally-observable side-channel leakage. However, these techniques are specific to a subset of x86 CPUs, and comparable primitives are uncommon in ARM64 CPUs.[1]

2.3 Hardware Fuzzing

Fuzzing is an automated technique to uncover vulnerabilities by providing complex systems with unexpected or randomized inputs.

Differential Fuzzing. Differential fuzzing is helpful in identifying subtle vulnerabilities in complex systems and protocols [41]. In differential fuzzing, multiple implementations of a specification are tested against each other with identical inputs, identifying discrepancies that indicate potential vulnerabilities or implementation errors. Recently, differential fuzzing has been used to find vulnerabilities in complex CPUs [46, 63].

CPU Fuzzing and Validation. CPU fuzzing extends general fuzzing principles specifically to microarchitectural and architectural components of processors. It involves generating randomized or crafted instruction sequences designed to trigger undocumented behaviors, implementation errors, or subtle timing anomalies within CPUs. By systematically probing the CPU with varied inputs, CPU fuzzing can reveal architectural mismatches [7, 45, 46, 59, 63], transient execution vulnerabilities [26, 43, 44], and microarchitectural side channels [12, 20, 28, 70, 71]. Advanced CPU fuzzing methods often utilize coverage-guided approaches or differential analysis across multiple hardware implementations or simulation models to efficiently detect CPU-specific vulnerabilities [59, 60, 63].

CPU bug testing, using fuzzing and other design validation techniques [36], can be performed pre-silicon or post-silicon. Pre-silicon testing is slow and limited in execution breadth but enables early detection with fine-grained observability of corner cases. Post-silicon testing runs at full speed, exercises entire software stacks and realistic workloads, and often reveals bugs in end-to-end systems, albeit with reduced internal visibility.

3 Methodology: Leaking Cache State

In this section, we present our 4-stage methodology for automatically discovering instruction sequences that leak microarchitectural state (specifically, cache state) via architectural state, without relying on timing measurements. We define the leakage and threat models (Section 3.1) and the process of generating instruction sequences and differential testing ($\mathcal{S}1$, Section 3.2). For filtering out non-deterministic or noisy differences, we employ a correlation oracle and reduction procedure based on the F-score ($\mathcal{S}2$, Section 3.3). In a verification stage, we automatically assess the real-world applicability by synthesizing a covert channel ($\mathcal{S}3$, Section 3.4). Finally, we propose a post-fuzzing clustering stage that minimizes the overhead of manual inspection ($\mathcal{S}4$, Section 3.5).

3.1 Leakage and Threat Model

Microarchitectural components such as caches, branch predictors, and TLBs operate below the abstraction provided by the ISA. Under normal conditions, these components are not expected to influence architectural state, i.e., the state observable by software through

[1]ARMv9 supports the optional transactional memory extension (TME) (FEAT_TME), but we have not seen it in any off-the-shelf hardware today.

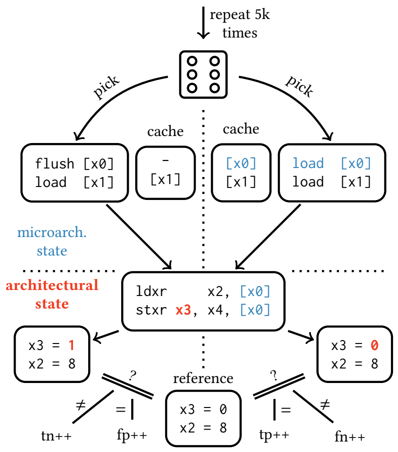

Figure 1: Overview of ExfilState. For computing the F-score, ExfilState randomly picks one of the two discovered cache states 5000 times. The respective cache state instruction sequence sets the microarchitectural cache state. ExfilState runs the generated instruction sequence, records the architectural state, and compares it to the reference state. Depending on the outcome, a true/false positive/negative is recorded.

registers, memory, and architectural flags. However, this abstraction is imperfect. Certain instructions may, under some conditions, reveal the state of internal components through architecturally visible interfaces (e.g., registers or exceptions).

In this work, we target leakage stemming from cache state. We define an *architectural leak* as any observable difference in the architectural state solely caused by earlier changes in the cache state. These differences may manifest in general-purpose registers, memory values, or status flags. We assume an attacker with the ability to execute unprivileged code but without access to timing measurements or performance counters. We assume no hardware bugs (e.g., Meltdown-type transient-execution vulnerabilities [9]) and no control over speculative execution beyond standard, well-known Spectre gadgets applicable to most CPUs [33, 39].

While our methodology is generalizable to other microarchitectural components, we focus on the cache for two reasons. First, the cache is one of the most widely studied sources of microarchitectural leakage, and second, its state can be reliably manipulated from user space using standard instructions.

3.2 $\mathcal{S}1$: Sequence Generation & Difference Discovery

We rely on a differential testing framework as illustrated in Figure 1. For each candidate instruction sequence, we prepare two execution contexts that differ only in their cache state. One version ensures that specific memory addresses are cached, while the other flushes (or evicts) those addresses using instructions such as `DC CIVAC`. After preparing the cache state, the candidate sequence is executed, and the final architectural state is recorded.

The two runs are then compared. If the architectural state differs between the two executions, we mark the sequence as a potential leaker. Since the cache state is the only variable factor, any resulting difference in register or memory content indicates that microarchitectural state has influenced architectural behavior. The comparison covers all general-purpose registers and all memory. In case of an exception, e.g., an illegal instruction exception, it also covers all the signal handling metadata provided by the kernel. We normalize the execution context to ensure deterministic behavior: the same candidate sequence, the same memory mappings, and the same initial register contents. This ensures that observed differences are only due to the modified cache state.

Instruction Sequence Generation. Instruction sequences are generated randomly. The generator produces syntactically valid sequences with constraints designed to target the cache subsystem. In particular, we restrict the generation to sequences of instructions that interact with memory, including loads, stores, and atomic operations. Arithmetic instructions, which do not touch the memory hierarchy, are excluded to reduce noise and increase search efficiency. This design choice is based on the hypothesis that only memory-accessing instructions can trigger observable effects from changes in cache state. To evaluate this hypothesis, we conducted a comparative experiment where we ran fuzzing campaigns with and without memory instructions. As we speculated, only sequences containing memory accesses resulted in architectural differences. The tested microarchitectures show 148 (Cortex-A53, A73, A72) to 198 (Apple M1) memory instructions. The complete set of memory instructions supported by the Apple M1 is provided in Listing 6, located in Appendix A.

3.3 $\mathcal{S}2$: Correlation Oracle & Reduction

We find that not all discovered architectural differences are deterministic or correlate with the microarchitectural states. Therefore, we employ a statistical scoring oracle based on the F-score, which measures the correlation between the microarchitectural and architectural states. We rely on the F-score as it is commonly used for evaluating side channels [23, 56, 71–73].

Figure 1 shows how the candidate sequence is executed multiple times under both cache conditions and how the resulting architectural states are compared against a fixed reference. From these runs, we calculate the number of true positives, false positives, false negatives, and the resulting F-score. A sequence with a high F-score reliably produces distinct architectural states depending on the cache condition, while sequences with low scores are discarded.

Reduction. To reduce the complexity of our findings, we try to reduce the instruction sequences automatically. We iteratively replace each instruction with a `NOP` instruction as long as the F-score does not drop below a defined threshold. Any reduction step that reduces the measured correlation below a fixed threshold is considered a failure and is reverted. This mechanism is similar to the methods used for optimizing cache eviction sets, and it can further benefit from similar optimizations [68]. We reduce the two cache states by testing whether each cache line's cache state affects the architectural difference. For each cache line, we test both cached and non-cached states. If running the candidate sequence under one of these two unified cache states still gives an architectural difference and a high F-score, the reduction step was successful.

3.4 $\mathcal{S}3$: Verification

Although the F-score effectively filters noise, it does not guarantee real-world exploitability. In particular, some sequences may appear to leak due to hardware prefetchers or speculative execution, rather than a direct dependency on cache state. Disabling prefetchers is not always possible, especially on mobile and embedded platforms.

To address this, we introduce a verification stage based on a covert channel scenario. In this setup, a sender encodes a message using a single cache line, and a receiver decodes the message via the architectural side channel. Decoding successfully across repeated runs demonstrates that the sequence relies on deliberate cache state modulation and can be used as a building block for attacks. By measuring the accuracy of message recovery and comparing it against a baseline, we can distinguish actual cache-state-dependent sequences from artifacts introduced by internal CPU mechanisms. Sequences that pass this test, i.e., have a bit-error rate below a chosen threshold, are verified and included in our final results. Additionally, the bit-error rate can be used to rank the side channels.

3.5 $\mathcal{S}4$: Clustering

Many high-scoring sequences turn out to be semantically equivalent, in that they reveal the same underlying root cause or produce the same architectural outcome. To reduce manual analysis effort, we group sequences into clusters based on their architectural effects. Clustering is automatically performed post-fuzzing on a number of key attributes for each reproducer. We mainly use the type of architectural difference, e.g., register or memory difference. We also consider exception types, in case of exceptions, and the number of architectural differences. This process reveals common leakage patterns and instruction sequences, and allows prioritizing novel and unexplored sequences. It also facilitates manual analysis of unique leakage mechanisms by reducing the records requiring inspection.

4 Implementation: ExfilState

In this section, we discuss ExfilState, our proof-of-concept implementation of the methodology in Section 3. We implement ExfilState's 3 fuzzing stages in C and Assembly: sequence generation and difference discovery ($\mathcal{S}1$, Section 4.1), correlation oracle and reduction ($\mathcal{S}2$, Section 4.2), and verification ($\mathcal{S}3$, Section 4.3). We use Python for the final post-fuzzing clustering stage ($\mathcal{S}4$, Section 4.4).

4.1 $\mathcal{S}1$: Case Generation & Difference Discovery

In this section, we discuss the instruction-sequence generation, memory mappings and register state generation, and how ExfilState tests and enumerates microarchitectural states.

Instruction Sequence. As discussed in Section 3.2, we restrict the instruction sequences to only memory-interacting instructions. Note that these also include non-load-store instructions such as branches, which also interact with memory. Since the tested systems implement a variety of ARM extensions, we opt for dynamically enumerating all valid memory-interacting instructions. We can enumerate all such instructions by testing if the instruction emits a page fault in any encoding. Therefore, we instantiate each opcode parsed from the ARM specification [4] and check for a page fault, i.e., `SIGSEGV` or `SIGBUS`. We test 20 random encodings of each instruction. If any of these encodings emits a page fault, we use the instruction in the sequence generation.

Given this list of memory-interacting opcodes, we randomly initialize a list of page-faulting instructions for a user-supplied sequence length. We restrict the number of registers used in the encoding of the generated instruction so that dependencies between the instructions are likely. We evaluate the impact of the number of registers and sequence length in an ablation study (Section 5.3).

Architectural State: Memory & Fuzzing Values. The architectural state is directly connected to the instruction sequence, as the instructions use the values set in the registers to interact with memory. Because we aim to find architectural cache state leakage, providing valid and "interesting" memory pointers via the registers is important. We emit pointers to different kinds of memory, i.e., read-only, read-write, and read-write-execute mapped memory, to cover all potential memory configurations available for user-space programs. We specifically supply pointers to page-boundary memory locations, as we hypothesize that these trigger edge-cases in the microarchitecture. We randomly set registers to valid instruction encodings, too, as we also map pages read-write-execute, allowing ExfilState to execute self-modifying code. We fill the mapped memory with the same set of 8-byte pointers and values, allowing ExfilState to generate pointer-chasing sequences.

Microarchitectural State. We represent the microarchitectural state as a bit vector where each bit represents the cache state, i.e., cached and non-cached, for each cache line pointed to by any of the limited number of registers. We set all other cache lines to the non-cached state by flushing them, as keeping other memory cached could potentially evict the cache lines that are actually used.

Setting the cache state of the tracked cache lines is done by selectively flushing or loading a cache line, depending on the state, as shown in Figure 1. We thereby issue the memory accesses and flushes that set the state of the tracked cache lines directly before executing the random candidate sequence. Thus, cache lines are not evicted by unrelated code. We shield the random candidate sequence from this cache-state-setting sequence with a full barrier, i.e., `DSB SY` plus `ISB` [4].

4.2 $\mathcal{S}2$: Correlation Oracle & Reduction

Next, we discuss details of using the F-score-based correlation oracle to evaluate the stability of an architectural difference and the reduction of sequence length and cache state.

Correlation Oracle. We implement the correlation oracle as described in Section 3.3. ExfilState takes 5000 measurements, and randomly uses one of the two cache state sequences. It compares the result against the previously captured architectural state and increments counters for true positives, false positives, and false negatives as shown in Figure 1. ExfilState discards architectural differences that lead to an F-score below 80 %.

Reduction. For non-discarded sequences, ExfilState applies the reductions described in Section 3.3. For reduction of the randomly-generated candidate sequences, we use standard `NOP` instructions and the implemented correlation oracle to detect incorrect reductions. For each differing bit in the two cache states, ExfilState tries both configurations, i.e., both cache states with the respective bit 0 (non-cached) and 1 (cached). If one of the two unified cache states still gives a high F-score, the reduction step is successful.

4.3 $\mathcal{S}3$: Verification

This section discusses the verification stage of ExfilState. We synthesize a cache covert channel that uses the discovered gadget to covertly transmit data via the cache. While we use a covert channel to automatically infer exploitability, other leakage settings, such as Spectral (cf. Section 7.3), could verify real-world applicability.

ExfilState automatically identifies which cache line is the one that architecturally leaks microarchitectural state by diffing the two cache states. This cache line is then used as the target cache line for the covert channel. The sender transfers 128 B of random data over the target cache line by encoding each bit in the microarchitectural state of the target cache line. The receiver runs the discovered instruction sequence and compares the resulting architectural state to a reference result for a cached cache line. If the result is equal to the reference result, the receiver stores a 1; otherwise, a 0. ExfilState calculates and logs the bit-error rate.

4.4 $\mathcal{S}4$: Clustering

The final stage of ExfilState is the post-fuzzing clustering of logged discovered side channels. We mainly rely on the differences between the recorded architectural states for the two possible states of the target cache line. We record the differences (e.g., memory, register, or signal address differences) and take that as the first key for the clustering classes. Further, we add the signal numbers of both results as another key. For each logged reproducer, we extract these keys and cluster the reproducers by the combined key.

5 Evaluation

In this section, we evaluate the efficacy of ExfilState in discovering architectural side channels. Over the course of 326 device-hours, ExfilState discovers 13 116 raw side-channel primitive reproducers. We further assess the clustering step's ability to reduce the number to 32 classes. Ultimately, this results in 5 unique architectural side channels. Finally, we evaluate discovery performance over time, and conduct an ablation study of critical fuzzing parameters.

5.1 Experimental Setup

Hardware and Targets. We execute our experiments on 160 devices encompassing 37 unique microarchitectures, including ARMv8 and ARMv9 CPUs. Devices include smartphones, laptops, servers, and single-board computers. A detailed list of devices can be found in Table 4 in Appendix D. We distribute fuzzing tasks across all machines for a total of 2021 CPU hours. This is performed primarily on a phone farm run by Google, incorporating 136 unique models of Android phones. Additionally, we use ARM CPUs on the Google and AWS cloud, a local lab with 2 laptops, 10 single-board computers, 7 phones, and 2 Macs to add coverage and facilitate targeted testing for discovered side channels. For devices with hybrid CPUs, we ensure ExfilState runs on at least one core of every core type.
Fuzzing Configuration. We configure ExfilState with varying fuzzing parameters, primarily the instruction sequence length and number of registers involved, as explored in the ablation study (Section 5.3). Unless stated otherwise, we use a default configuration of 8 instructions and 4 registers. As seed, we use a combination of on-device Unix milliseconds and the CPU core to make the seed random per device and core. The raw outputs of ExfilState undergo a clustering stage (cf. Section 3.5) that groups similar reproducers and reduces redundancy. We report on both raw output and clustered side channel categories.

5.2 Side-Channel Discovery

Discovering architectural side channels is the primary goal of ExfilState. We can measure its efficacy by evaluating the clustering process, i.e., minimization of results, the number of unique side channels, their complexity, and the time it takes to discover side channels. We analyze each discovered side channel in Section 6.
Clustering. From 13 116 raw reproducers, the clustering stage condenses the output into 32 distinct classes within 13 min, significantly reducing manual triage effort. Manual inspection of class representatives reveals 5 previously unknown architectural side channels with distinct leakage mechanisms.
Categorization. We categorize the 5 discovered side channels into two classes: exception-free and exception-dependent. 2 side channels manifest without triggering faults, while 3 consistently rely on CPU exceptions, such as segmentation faults, to leak information. Table 1 summarizes the prevalence of the 5 side channels across microarchitectures. In total, 36 of 37 microarchitectures are affected by at least one discovered side channel. The 2 *exception-free* side channels Lx+Sx and Store+Ret affect 22 and 31 of 37 tested microarchitectures, respectively. The 3 *exception-dependent* side channels are slightly less versatile, as they require the application to handle or suppress exceptions, which is often only feasible in low-level languages such as C. ExfilState only discovers the 3 exception-dependent side channels on 6 microarchitectures. 2 microarchitectures are affected by Pointer-Chase and 4 by Split-Store and Translation-Race (cf. Table 1). However, while ExfilState discovers the exception-dependent side channels on these microarchitectures, all are also affected by at least one exception-free side channel.
Side-channel Complexity. While our configuration uses 8 instructions and 4 registers, the discovered side channels have been reduced to, on average, 2.31 instructions and 1.02 registers that differ in the cache state. The minimum sequence length for all discovered side channels is 2, whereas the most complex side channel, Lx+Sx on Apple's Icestorm microarchitecture, requires sequences of 5 instructions. Interestingly, some side channels require a certain number of instructions on certain microarchitectures. Listing 7 in Appendix C shows an example of Lx+Sx, where the short sequence works on Cortex-A73, and only the longer sequence makes it work on Apple Avalanche.
Discovery Time. We evaluate how long ExfilState needs to run to discover all side channels. Figure 2 shows the percentage of unique discovered side channels over time. The plot line shows the average over all microarchitectures, while the band shows the standard error when averaging over the microarchitectures. 90 % of the side channels are discovered within the first 75 min, while discovering all side channels takes 7 h on average.

Table 2 lists the time to discovery for each of the 5 side channels, again averaged over all microarchitectures. Split-Store and

Table 1: Overview of which tested microarchitecture is affected by which side channel. ✓ indicates a bit-error rate of at most 25 % in the covert channel verification, while ~ indicates a higher rate. Exponents show the complexity of the discovered sequence in the form of the sequence length. The last row shows if a microarchitecture is affected by any of the 5 side channels.

Side Channel	Cortex-A53	Cortex-A55	Cortex-A510	Cortex-A520	Cortex-A72	Cortex-A73	Cortex-A75	Cortex-A76	Cortex-A77	Cortex-A78	Cortex-A710	Cortex-A715	Cortex-A720	Cortex-A725	Cortex-X1	Cortex-X2	Cortex-X3	Cortex-X4	Kryo
Lx+Sx	✓^{2}	✓^{4}	✓^{4}	✓^{5}	✓^{2}	✓^{2}	✓^{3}	✓^{3}				✓^{3}	✓^{3}	✓^{3}					
Store+Ret	✓^{2}	✓^{3}	✓^{3}	✓^{3}		~^{2}		✓^{2}	✓^{3}	✓^{2}	✓^{3}	✓^{3}	✓^{3}	✓^{2}	✓^{3}	✓^{3}	✓^{3}	✓^{3}	
Split-Store						✓^{2}	✓^{2}												
Translation-Race						✓^{2}	✓^{2}												
Pointer-Chase	✓^{3}				✓^{2}														
Affected	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗

Side Channel	Falkor-V1/Kryo	Kryo-V2	Kryo-3XX-Gold	Kryo-3XX-Silver	Kryo-4XX-Gold	Kryo-4XX-Silver	Carmel	Kunpeng Pro	Oryon	Oryon V2 P-L	Oryon V2 P-M	Exynos M3	Neoverse-N1	Neoverse-V2	Firestorm-M1	Icestorm-M1	Avalanche-M2	Blizzard-M2
Lx+Sx	✓^{2}	✓^{3}	✓^{4}	~^{4}		✓^{4}			✓^{3}	✓^{3}	✓^{3}	✓^{4}			✓^{4}	✓^{5}	✓^{4}	~^{8}
Store+Ret		✓^{2}	~^{2}	✓^{3}	✓^{2}	✓^{3}	✓^{3}	✓^{2}	✓^{4}	✓^{2}	✓^{3}	✓^{3}	✓^{2}	✓^{2}	✓^{2}	✓^{4}	✓^{2}	✓^{3}
Split-Store	✓^{2}		✓^{2}															
Translation-Race	✓^{2}		✓^{2}															
Pointer-Chase																		
Affected	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

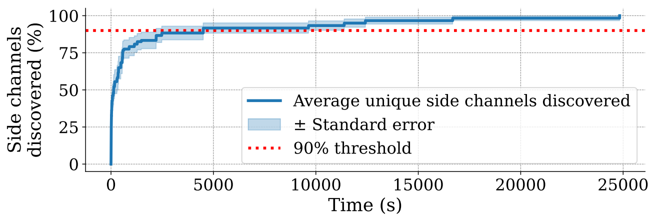

Figure 2: Percentage of unique discovered side channels over time, averaged over microarchitectures. The band shows the standard error. On average, 90 % of the side channels are discovered within 75 min, and all within 7 h.

Table 2: Average time to discovery, standard error, and number of reproducers for each side channel averaged and summed up over all microarchitectures.

Side Channel	AVG time to discovery (s)	Stderr (s)	Reproducers
Lx+Sx	3821	1520	2160
Store+Ret	526	194	4059
Pointer-Chase	94	67	135
Translation-Race	2	1	3997
Split-Store	2	1	1637

Translation-Race are discovered within seconds, while Pointer-Chase and Store+Ret take minutes. Lx+Sx is the slowest to find, with an average time to discovery of around 1 h. These results suggest that short, targeted fuzzing campaigns can effectively identify architectural side channels without requiring prolonged execution or complex instruction sequences.

5.3 Ablation Study

We perform an ablation study on two key fuzzing parameters: sequence length and register count. Each configuration is tested in isolation to assess its impact on the number of discovered side channels and the number of logged side-channel primitives. For all of the configurations, we run ExfilState 6 times for 20 min, totaling 2 h fuzzing time per configuration. We evaluate the percentage of side channels discovered and the number of reproducers logged for each configuration. We calculate the percentage of discovered side channels by dividing the number of discovered unique side channels by the number of discoverable side channels on the microarchitecture. The results are summarized in Figure 3, where the bar plots show the number of reproducers and the line plot shows the cumulative percentage of unique side channels discovered.

Sequence Length. For the number of generated random instructions (sequence length), we test lengths of 1 to 16 while keeping the number of registers fixed at 4. Instruction sequences shorter than two instructions fail to trigger side effects and thus do not expose side channels. Discovery performance improves with longer sequences, peaking around 8 instructions. Beyond that, performance deteriorates. We attribute this to more involved sequence reduction and performance overhead due to longer sequences.

Register Count. With register count varying from 1 to 8 (fixed sequence length of 8), we observe optimal discovery at 2 registers. Increasing the number of registers further decreases the number of reproducers. We assume the reason is the increased complexity in cache state exploration, leading to diminishing returns.

6 Discovered Side Channels

In this section, we describe each of the 5 discovered side channels in detail, analyze their root causes, and evaluate cross-core availability, speed, error rate, and resolution. We describe the 2 widely available and versatile exception-free side channels we call Lx+Sx and Store+Ret in Section 6.1 and Section 6.2, respectively. The exception-dependent side channels are described in Section 6.3.

6.1 Lx+Sx

In this section, we describe the first of the 2 widely available and exception-free side channels. We dub this side channel Lx+Sx, as all variants of Lx+Sx are based on a combination of an exclusive load and a following exclusive store. All variants exploit "constrained unpredictable" [3] or implementation-specific details [2]. Unlike regular stores, exclusive stores, such as STXR, encode a third general-purpose register in the instruction. This status register encodes whether the store succeeded ('0') or failed ('1'). Exclusive stores are

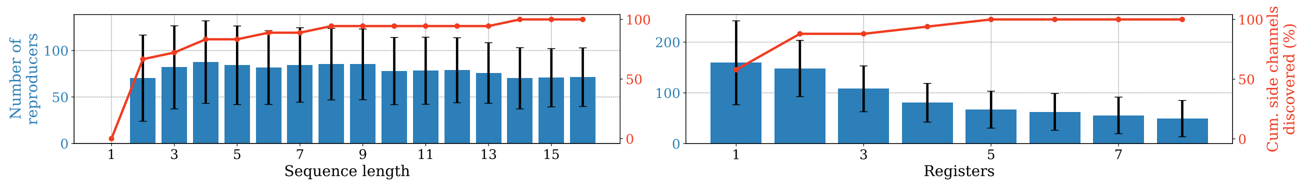

Figure 3: Ablation study results comparing the impact of sequence length and used registers on the number of logged reproducers (bars) and the cumulative rate of discovered side channels (line). Error bars indicate the standard error over 6 runs.

normally combined with an exclusive load (LDXR) and a comparison to implement hardware-backed mutexes [2]. In a typical setup, the exclusive store only succeeds if the block of memory is marked for exclusive access (via an exclusive load), i.e., no other store to this address has happened in the meantime [3].

Listing 1 shows the base variant of Lx+Sx. The exclusive store is the one leaking the microarchitectural state, i.e., cached or non-cached, of the victim cache line architecturally. Register `w0`, the status register of the exclusive store, is '0' (=success) after execution if the victim cache line is in the cache and '1' (=failure) otherwise.

The Cortex-A72 is not affected by the base variant of Lx+Sx, but by a different variant where the exclusive load (LDXR) has to load an *unrelated* cache line. For both variants, the manual does not document that the behavior depends on the cache state of involved cache lines. This is also not an obvious behavior and thus unlikely to be discovered manually without a fuzzer.

6.1.1 Exploitation: RSB-based Cache State Re-encoding. As Lx+Sx uses exclusive stores to leak the cache state of a victim cache line—i.e., whether it resides in the cache or not—mounting Lx+Sx directly on a cache line would require write access. As this is in general not given for a targeted cache attack, we introduce CSC (cache state copier), a primitive to copy the state of a cache line from a read-only victim cache line to an attacker-controlled cache line. CSC uses the idea of "weird gates" [16, 32] to "copy" the microarchitectural state with transient execution, but it relies on return-based misprediction [27, 39], which does not require mistraining. Horowitz et al. [27] also rely on such a return-based misprediction on Intel CPUs to build a NOT gate.

With CSC, we transiently access the victim cache line and access the attacker-controlled cache line with a dependency on the value loaded from the victim cache line. If the victim cache line is in the cache, the data arrives fast, and the second memory access can be transiently performed, so the attacker-controlled cache line is cached. If the victim cache line is not in the cache, the transient execution window ends before the victim cache line data arrives. Because of the dependency, the attacker-controlled cache line is not loaded. This ensures that the attacker-controlled cache line has the victim cache line state after executing our transient-copy gadget. After transiently "copying" over the victim cache line state, we mount Lx+Sx directly on the attacker-controlled and, therefore, writable cache line.

6.1.2 Analysis. We hypothesize that the core of the Lx+Sx side channel is the interaction between exclusive load-store pairs and the cache coherence protocol. ARMv8-A exclusive access primitives comprise two instructions: LDXR, which performs a load and establishes an exclusive reservation on a memory address, and STXR, which attempts to store a value to that address only if the exclusive reservation is still valid. The ARM documentation for synchronization primitives [2] suggests that the monitoring for exclusive loads and stores involving shareable memory can be implemented using the cache coherence protocol. Even though not documented, the behavior of the exclusive store (STXR) fundamentally depends on the internal state of the microarchitecture—particularly the cache state of the accessed line. Our experiments show that the store only succeeds if the cache coherence state of the cache line is not *Shared*, indicating that the cache coherence protocol is indeed used for the implementation. Importantly, LDXR does not necessarily upgrade the cache line to an *Exclusive* or *Modified* state in the cache coherence protocol. Instead, it behaves as a regular load, and the line may remain in the *Shared* state even after execution of the instruction.

The STXR instruction, in turn, attempts a conditional store. If the reservation is still valid, i.e., no other writes to the address were observed by the exclusivity monitor, it proceeds with the store. However, if the cache line is not already in a writable state (e.g., not in *Exclusive* or *Modified*), the operation may incur a coherence miss. In such a case, the store must first invalidate copies in other cores or upgrade the cache line to a writable state, which introduces latency and increases the likelihood of the store failing. While some microarchitectures may attempt to speculatively acquire exclusive access after an LDXR to improve the success rate of the subsequent STXR, such behavior is neither architecturally guaranteed nor documented in a portable way. This implementation detail is crucial for the effectiveness of the Lx+Sx side channel.

Other Variants. The Cortex-A72 is only affected by the unrelated-load variant of Lx+Sx, i.e., the exclusive load and store target different addresses. This behavior is explicitly documented to be unpredictable: "If the target VA of a StoreExcl is different from the VA of the preceding LoadExcl instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE" [3]. Interestingly, the behavior is not unpredictable but depends on the cache state of the address used as store target. No other core is affected by this variant. Some microarchitectures, such as Apple Avalanche or Cortex-A55, need more complex sequences to trigger Lx+Sx (cf. Table 1 and Listing 7 in Appendix C). Thus, while Lx+Sx is straightforward on some devices, it requires tailored sequences on others, which ExfilState uncovers.

Cross-core Exploitability. Depending on the specific microarchitecture, Lx+Sx is not limited to the same core, as our manual analysis of the side channel shows. On the Cortex-A73 (base variant), the side channel works across cores and cross-core-type to

```
LDXR     x1, [victim]
STXR w0, x2, [victim]
```

Listing 1: The base sequence for Lx+Sx. The exclusive store only succeeds when the victim cache line is in the cache, reflected as status in the `w0` register.

a Cortex-A53. Other variants vary, e.g., a more complex variant on the A53 works only cross-same-core-type. We attribute these differences to the cache configurations of the different CPUs.

Speed, Error Rate & Resolution. We test speed, error rate, and resolution of Lx+Sx on the Cortex-A72 and A73 via a simple covert channel over one cache line. On the Cortex-A72, Lx+Sx runs with 277 kbit/s and a bit-error rate of 0 %. This gives a resolution of 3.61 µs. On the Cortex-A76, we see 1038 kbit/s and a bit-error rate of 0.03 %, resulting in a resolution of 0.96 µs.

Unaffected Implementations. On unaffected implementations (cf. Table 1), e.g., Nvidia Carmel or Huawei Kunpeng Pro, the default behavior, independent of the cache state, is that the store succeeds. This is expected from an optimized version of these instructions. However, we still find rare occurrences where the store fails on both unaffected microarchitectures. These occurrences increase slightly on the Huawei Kunpeng Pro when padding Lx+Sx with a flush instruction on the victim cache line.

6.2 Store+Ret

In this section, we give details on Store+Ret, the second widely-applicable exception-free side channel. Store+Ret is available on 31 of 37 microarchitectures.

Store+Ret is lined out in Listing 2. It uses two distinct instruction sequences: the store plus return sequence and a victim-to-be-overwritten instruction sequence. The store rewrites the instructions at the second instruction sequence. The return instruction is used to jump to the second rewritten instruction sequence. We find that affected CPUs execute stale instructions when the victim rewritten cache line is in the data cache and the new updated instructions when the cache line is not in the data cache. ExfilState finds this side channel, as it uses read-write-execute memory mappings, which can include valid instructions, such as `NOP`s. If the store instruction corrupts this valid instruction by overwriting it with an illegal instruction, the CPU executes a different number of instructions, visible via the signal address (`si_addr`).

6.2.1 Exception-free Store+Ret. We can manually craft this gadget to be exception-free and control-flow preserving by using only valid instructions and emitting a branch back to the original code. Listing 2 shows the final form of Store+Ret. We use `MOV` instructions to selectively encode '0' (non-cached) or '1' (cached) into the architectural register `x0`. `MOV x0, #0` is only visible to the IFU if the victim cache line is not in the data cache. A branch register (`BR`) instruction is used to jump back to the original code flow with the leakage encoded in `x0`.

6.2.2 Exploitation. As with Lx+Sx, we cannot mount Store+Ret directly on a victim cache line, as we need read-write-execute access to the victim cache line. Thus, we also use CSC (cf. Section 6.1.1)

```
; w0 encodes 'MOV x3, #0', w1 encodes 'MOV x3, #1'.
; w2 encodes 'BR =come_back_here'.
STR w1, [victim] ; Victim cache line sets x3=1 and returns.
STR w2, [victim, #4]
IC IVAC victim
STR w0, [victim] ; Replace code at victim to set x3=0.
RET victim ; Jump to overwritten (potentially stale) code.
come_back_here: ; x3 is 1 if victim is cached, 0 if not.
```

Listing 2: Store+Ret: The original code at `victim` sets `x3` to '1' and returns. Replacing it with code that sets `x3` to '0' causes the old code ('1') to run if the cache line was in the data cache and the new code ('0') to run if the cache line was not in the data cache, leaking the cache state architecturally.

to copy the microarchitectural state of a victim cache line to an attacker-controlled writable and executable cache line.

6.2.3 Analysis. In this section, we analyze the root cause and important properties of Store+Ret, such as the interplay with the I-cache and whether other jump instructions work.

Root Cause. Store+Ret exploits the incoherence between data and instruction cache on ARM CPUs. On x86, stores to executable memory ranges are guaranteed to always be visible to data loads [42]. However, on ARM CPUs, there is no such guarantee. Thus, instruction and data cache become incoherent on stores to such executable regions. The programmer has to use cache maintenance instructions to sync both caches [3].

Store+Ret exploits a race in the microarchitectural implementation on affected CPUs where newly written (jitted) instructions are only visible architecturally when the written cache line is not in the data cache. When the victim cache line is in the instruction cache, we only see stale instructions executed. This is expected, as cache coherence is not guaranteed, so the instruction fetch unit (IFU) fetches instructions from the instruction cache as long as they are not evicted. That means we need to invalidate the victim cache line from the instruction cache to set up Store+Ret.

Other Jumps. While Store+Ret sounds like it should be generically exploitable, i.e., with other types of jumps or without any jump, we find that Store+Ret exploits the misprediction property of `RET` instructions (cf. Section 6.1.1).

First, we test if overwriting in-stream instructions works, basically concatenating both instruction sequences. This does not trigger architectural differences, and only stale instructions are executed. We suspect this is because the IFU fetches instructions in blocks of entire cache lines (64 bytes) [1]. Therefore, the write cannot be visible to the IFU because the store and patched instruction are in the same cache line and therefore already fetched when the store occurs. This can be fixed by padding the patched instructions with `NOP`s. We find that 300 `NOP`s reliably only give the new updated instructions on the Cortex-A76. Further, we find that Store+Ret can be exploited in-stream, i.e., without any jump, by padding with 200 `NOP`s on the Cortex-A76.

Next, we test if other types of branches work, too, as ExfilState discovers Store+Ret only with `RET` instructions, with just a few exceptions, e.g., Apple M1/M2, where it finds Store+Ret with indirect branches (`BR`). On the Cortex-A76, we find that replacing the `RET` instruction with direct or indirect jumps does not trigger architectural differences. We suspect this is due to the implementation-specific speculative behavior of RET instructions. While direct and indirect jumps are trained always to take the jump in our gadget, custom returns always misspeculate. This triggers a difference in the microarchitectural race that is exploited by Store+Ret. With direct and indirect jumps, the IFU can predict and fetch the victim cache line, executing stale (valid) instructions. For returns, the IFU mispredicts and fetches the wrong instructions. Then the pipeline must be flushed and the correct victim cache lines fetched from memory. We suspect that at this point the write with the new updated instructions has progressed enough to be forwarded to the IFU.

Fences. Adding a Data Synchronization Barrier (DSB SY) between the store and return instruction reliably removes the effect of executing stale instructions. However, a barrier in combination with direct or indirect jumps does not lead to updated instructions being executed, as the IFU operates ahead of the barrier.

Cross-core Leakage. Store+Ret works across cores on the Cortex-A76. However, in contrast to Lx+Sx, Store+Ret is only exploitable across the same CPU-core type on the Cortex-A76.

Speed, Error Rate & Resolution. We measure a transfer rate of 473 kbit/s and a bit-error rate of 0.03 % on the Cortex-A76. This gives a resolution of 2.12 µs.

Other variants. Similar to Lx+Sx, there are multiple variants to trigger Store+Ret on different microarchitectures (cf. Table 1). Again, microarchitectures such as the Cortex-A55 or Oryon need more complex sequences to trigger different architectural results. These include loads to unrelated addresses or more complex store operations, such as store pair (STP). ExfilState uncovers variants of Store+Ret for nearly all microarchitectures.

Unaffected implementations. Again, we test unaffected implementations (cf. Table 1) for their default behavior. We find that the Cortex-A72 always executes the stale instructions. We suspect this is because the Cortex-A72 does not implement the suspected forwarding to the I-cache. Note that this does not mean that the unaffected implementations are entirely unaffected—there might still be sequences that could distinguish cache hits from misses, even if ExfilState does not find any.

6.3 Exception-dependent Side Channels

In this section, we discuss the 3 exception-dependent side channels. We hypothesize that all 3 side channels exploit race conditions in the address translation logic when handling unaligned page-boundary-crossing address translations. ARM CPUs split unaligned memory accesses into multiple loads/stores. For example, when a non-single-copy 8-byte store is executed to the last 4 bytes of a page, the store is split into two 4-byte stores, and each is executed independently. Further, address translation may fail on either side of the page: "An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary" [3]. Thus, the 8-byte store can write partially to memory when one of the pages is not mapped.

6.3.1 Pointer-Chase. The first exception-dependent side channel, Pointer-Chase, exploits the fact that accesses can fail on more than one address. Listing 3 shows Pointer-Chase. The first memory access instruction loads a pointer to a page boundary where both pages are not mapped (e.g., address 0x3fff) from the victim cache line. Depending on the cache state of the victim, we see a segmentation fault on 0x3fff (victim cached) or 0x4000 (victim non-cached) in user space. We suspect this is due to a microarchitectural race condition where the microarchitecture spends more time on the address translation depending on the victim cache state.

```
LDR x0, [victim] ; victim contains 0x3fff
LDR x1, [x0]
```

Listing 3: Pointer-Chase: The second load fails on 0x3fff for cached victim and on 0x4000 for uncached victim.

```
LDR x0, [victim]
STR x0, [page-boundary-ptr]
```

Listing 4: Translation-Race: If the victim is cached, the first partial store succeeds before the second triggers a fault, resulting in bytes written when the victim is cached.

```
LDRB x0, [victim]
STR  x1, [victim]
```

Listing 5: Split-Store: If the victim is cached, the first partial store completes before the second faults, resulting in bytes written if the victim is cached.

6.3.2 Translation-Race. Translation-Race exploits that stores are split and can be partially committed. Translation-Race is depicted in Listing 4. The first load is fast or slow depending on the victim cache state. The dependent store is split into two partial stores because the target points to unaligned page-boundary crossing memory where the first page is mapped, while the second one is not. When the victim cache line is in the cache and the store value arrives fast, the first partial store succeeds before the address translation fails on the second page. Therefore, the partial store is committed, and we can architecturally see that the victim is cached. When the victim cache line is not in the cache, the faulting address translation flushes the pipeline before the partial store is committed. Translation-Race essentially races a load on the victim address against the address translation of the unaligned pointer.

6.3.3 Split-Store. Split-Store (Listing 5) exploits that store operations use a store buffer and can update cached parts before checking page permissions. The victim address spans a page border, where only the first page is mapped. Loading one byte succeeds, but storing 4 bytes fails, as the target address is partially unmapped. Still, the store to the first byte succeeds if the victim cache line is already cached, as we suspect that the store can update this part in the store buffer. Thus, the first byte of the victim data is only modified if the victim is cached.

6.3.4 Analysis. In this section, we analyze key properties such as cross-core leakage for the 3 exception-dependent side channels.

Cross-core Leakage. Translation-Race and Split-Store on the Cortex-A73 work only across the same CPU-core type. Pointer-Chase does not work across cores on the Cortex-A72.

Speed, Error Rate & Resolution. For Pointer-Chase (Cortex-A72), we measure a transfer rate of 11 kbit/s (1 bit every 90 µs) and a bit-error rate of 0.15 %. For Translation-Race (Cortex-A73), we measure 102 kbit/s (9.77 µs) and 0.12 %. For Split-Store (Cortex-A73), we measure 147 kbit/s (6.79 µs) and 0.01 %.

Exploitation. Translation-Race is the only side channel we discover that can be directly mounted on a cache line. This is because we only load a value from the victim cache line and store that, independent of its value. Pointer-Chase cannot be mounted directly, as the page-boundary pointer needs to be specially crafted. We try to craft a sequence of arithmetic instructions that transforms an arbitrary victim value into a vulnerable pointer; however, adding dependent instructions between the loads breaks the side channel. Split-Store can again not be mounted directly, as the victim pointer needs to be at the page boundary. However, for both variants, the RSB-based transient copy gadget CSC (cf. Section 6.1.1) can be used to re-encode and leak a victim cache line.

Unaffected Implementations. Again, we test the behavior of other microarchitectures. For Pointer-Chase, other microarchitectures always emit the fault on the first page, i.e., `0x3fff`. For Translation-Race and Split-Store, the partial store never succeeds on other microarchitectures.

7 Case Studies

In this section, we use the discovered side channels to mount attacks on AES, mitigate these attacks with the same side channels, and craft a fast architectural Spectre attack.

7.1 AES T-table Attack (Lx+Sx & Store+Ret)

In this section, we demonstrate a practical AES T-table attack leveraging Lx+Sx and Store+Ret as architectural side channels. Our attack uses the T-table implementation of OpenSSL 3.5.0 (April 2025), which is essentially unchanged since 1.0.1e, a version commonly used for benchmarking side channels [18, 19, 21, 52].

Setup. As the main system for our case study, we use an ARM64 Cortex-A72 CPU with 4 cores and 8 GB RAM, running Ubuntu 18.04 as operating system. Additionally, we test the case study on multiple devices: Cortex-A720 (Pixel 9 with Android 15), A715 (Pixel 8a with Android 15), A72 (Raspberry Pi 4 with Ubuntu 24.04), and A76 (Pixel 6a with Android 14 and Raspberry Pi 5 with Ubuntu 24.04). For the Cortex-A76, we use Store+Ret instead of Lx+Sx.

The attack scenario is modeled after traditional Flush+Reload techniques but replaces timing measurements with Lx+Sx. However, as Lx+Sx requires a writable cache line, we cannot directly apply it to the AES T-table. Thus, we rely on CSC to copy the cache state to an attacker-controlled writable cache line and mount Lx+Sx on this cache line. Our attack implementation is inspired by the baseline provided by Gruss et al. [21], with modifications tailored specifically to exploit Lx+Sx.

Evaluation. We execute our AES T-table attack 100 times. With Lx+Sx, we reliably distinguish cache hits from misses at the architectural level, eliminating timing-related noise. In 99 % of cases, we fully recover the AES key without errors. Our method's key

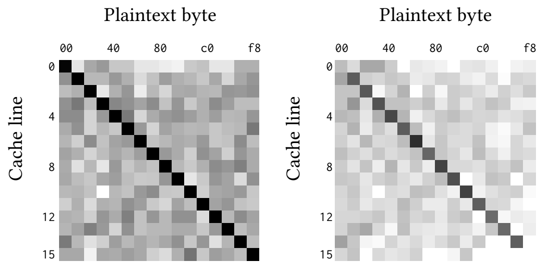

Figure 4: AES T-table cache-access pattern on an ARM64 Cortex-A72 with Lx+Sx (left) and Flush+Reload (right).

recovery takes 764 ms (±0.83, n=100) on average. While Flush+Reload is slightly faster with 607 ms (±0.86, n=100) on this CPU under identical conditions, we only get the correct key in 39 % of the runs. The main reason is the limited timer resolution. On this SoC (Marvell Armada 7040), the best timer has a resolution of only 40 ns, making classical timing-based cache attacks such as Flush+Reload unreliable. We additionally evaluate the attack under stress by running the `stress` utility on 1 and 2 cores, respectively. As expected, this only has a negligible impact on Lx+Sx, showing that the side channel is robust in presence of noise. Figure 4 shows the typical heatmap for the T-table accesses, with the expected diagonal when the first key byte is '0'.

For the other devices, we confirm that Lx+Sx (Cortex-A720, A715, and A72) and Store+Ret (Cortex-A76) both give a clear diagonal, indicating that the side-channel attack works well. Overall, this case study underscores the practical implications of Lx+Sx and Store+Ret, highlighting their efficacy in real-world attacks, such as on cryptographic key extraction and real-world devices.

7.2 AES T-table Mitigation (Lx+Sx)

In this section, we use Lx+Sx to harden the OpenSSL AES T-table implementation against the attack discussed in Section 7.1. We preload the entire T-tables into the cache and use Lx+Sx when accessing the T-tables to abort the algorithm if the cache line is not in the cache. Our mitigation joins previous mitigations relying on keeping the T-tables cached. These mitigations range from preloading the T-tables into the cache and keeping them cached [8, 22, 47, 64, 72], to using transactional memory to ensure execution only terminates on preloaded and still cached T-tables [22], to using performance-counter overflows and other interfaces to abort encryption when under attack [54, 72].

Setup. We again evaluate this Lx+Sx-based T-tables mitigation on the Cortex-A72 of a Marvell Armada 7040, the same microarchitecture and SoC as in the previous case study. We make sure that the entire T-tables are in the cache by accessing each of the 128 cache lines before the encryption. We shield each access to the T-tables in the encryption with a probing step based on Lx+Sx. If this probing step finds that the cache line is not cached anymore, we stop the encryption and report that an attack was discovered.

Evaluation. We run 1 000 000 AES encryptions and choose randomly for each if the attack from the previous case study is active. To account for camouflaged attacks [35], we only attack a single cache line, as the detection is built on the cache-line level either way. We count true positives (attack executed and discovered), false positives (no attack but reported), and false negatives (attack executed but not discovered). Based on these numbers, we calculate that our mitigation runs with a precision of 99.68 % and a recall of 99.92 %. Combined, this gives an F-score of 99.8 %. This shows that the discovered architectural side channels can not only be used for attacks but for defenses as well.

7.3 Spectral: Architectural Spectre (Lx+Sx)

In this section, we present an ARM64-variant of the Spectral attack introduced by Zhang et al. [80]. Spectral is a variant of Spectre-type attacks, using architectural leakage instead of a traditional timing-based side channel. We rely on a classical Spectre-PHT attack [33] but use Lx+Sx instead of conventional timing methods. Combinations with other Spectre variants are possible, but we opt for Spectre-PHT, as this works reliably on a wide range of devices. Our Spectral attack is the only variant that works on ARM64, as `umwait` or similar instructions are unavailable on ARM64 [80]. Spectral on ARM64 is highly efficient, leaking 11 457.9 B/s (±27.26 , n=150 000) on average without errors. This is more than a factor of 7 faster than the previously fastest Spectre-PHT attack on ARM64 [24].

Threat Model. Consistent with prior work [25, 27, 33, 34, 80], we assume an attacker with unprivileged code execution. The attacker lacks architectural timing primitives. This includes high-resolution primitives, such as access to the `PMCCNTR` register [38] or a counting thread [57], and coarse-grained timers with microsecond or lower resolution [38, 58, 61]. As with typical Spectre attacks, we use shared memory between the attacker and the victim, without the necessity of hyperthreading [6] or OS-controlled timeouts [80]. For a reproducible and controlled evaluation, we rely on an injected leakage gadget. In contrast to previous work on Spectral attacks [80], we do not require a bit-wise leakage gadget but can use traditional byte-wise leakage gadgets [33].

Attack Methodology. Our side channel is a drop-in replacement for timing side channels, so we can use *unmodified* Spectre attacks targeting ARM64 [24] or supporting ARM64 [9]. Since the attacker controls the encoding buffer, we can directly use Lx+Sx without CSC. We also do not require calibration, as our side channel's output is binary. Unlike prior approaches using instructions such as `umwait` [80], we need no special gadgets or Spectre variants that transiently write to memory. Thus, existing Spectre gadgets can be readily exploited with our variant. Synchronization is straightforward, as the attacker executes and observes the gadget.

Setup. We use a simple Spectre-PHT attack that leaks out-of-bounds bytes from an array akin to previous proof-of-concept implementations [9, 24]. For mistraining, we rely on in-place mistraining [9] and access 10 inbound values for each out-of-bound value. We use a leak gadget like the original Spectre paper [33], spreading the leaked value to 256 pages. We use `DC CIVAC` to flush the array and Lx+Sx to measure the cache state.

Results. For the evaluation, we use a Cortex-A73 running Ubuntu 20.04.5 LTS. We achieve a leakage rate of 11 457.9 B/s (±27.26 , n=150 000) without errors. This is more than a factor of 7 faster than the previously fastest Spectre-PHT attack on ARM64 [24]. In fact, this leakage rate outperforms all timing-based Spectre attacks on x86, too, and is nearly 3 times as fast as the fastest attack (cf. Schwarzl et al. [58] Table I). Moreover, our Spectral implementation is more than 50 % faster than the fastest Spectral attack on x86 [80].

8 Discussion

In this section, we discuss mitigations, limitations, and related work.

8.1 Mitigations

Our findings challenge spot mitigations, such as restricting timer accuracy [24, 29, 31, 37, 38, 57]. However, effective mitigations exist.

Software. Still, in line with other side channels, secrets in programs can be protected against our side channels by relying on constant-time programming techniques [5, 50].

Operating System. Our side channels exploit the same leakage as other cache attacks, differing only in the extraction method. Thus, mitigations focusing on cache leakage [22, 72] and not on the extraction method also protect against our side channels. Similarly, detection methods [35] can also detect our side channels.

Hardware. Even though the side channels are present on nearly all tested microarchitectures, they are not inherent to the ISA specification. As not all microarchitectures are affected by all side channels, it is possible to implement the microarchitecture so that it does not suffer from these side channels. Thus, CPU vendors can change their designs to mitigate the side channels at the hardware level.

8.2 Limitations

Our approach has two main limitations, one inherent to fuzzing-based techniques, the other an engineering limitation.

Incompleteness. As with all fuzzers, our approach is not complete. Thus, if we do not find an architectural side channel on a CPU, this does not mean there is none. However, our evaluation (cf. Section 5) also shows that on CPUs where we find an architectural side channel, it typically does not take more than 7 h.

Instruction Limitation. Our current proof-of-concept implementation is limited to instructions documented in the machine-readable ISA specification [4]. Thus, in case there are architectural side channels in custom or undocumented instructions, our fuzzer cannot find them. However, this is primarily an engineering problem that can be solved, as also shown in previous work [62, 63].

8.3 Related Work

In this section, we discuss related work on CPU fuzzing, architectural side channels, and architectural CPU vulnerabilities.

Black-box CPU Fuzzing. Early black-box fuzzers, such as Covert Shotgun [17], used handpicked instructions to uncover covert timing channels. ABSynthe [20] generalized this by testing the full ISA for contention. Osiris [71] introduced ISA-aware fuzzing for non-contention timing channels. BETA [10] added coverage-guided mutation to accelerate black-box discovery of timing leaks. In contrast, our focus is not on timing but on making microarchitectural state architecturally observable.

Other fuzzers explore speculative execution (Transynther [43], Revizor [44]), undocumented instructions (Domas [14]), or target-specific ISAs like RISC-V (RISCover [63]) for data leakage or crashes.

We, instead, identify architectural exposure of the microarchitectural state. SiliFuzz [59] detects CPU correctness bugs using differential fuzzing. While occasionally revealing security issues (e.g., Reptar [45]), its primary focus is to find defective hardware.

Architectural Side Channels. Yu et al. [78] use exclusive load and store instructions on Apple M1 to build a cache side channel similar to Lx+Sx, but exploit eviction of attacker cache lines rather than direct state leakage. Xu et al. [76] and Van Bulck et al. [65] exploit page-fault behavior and page-table bits, respectively, for architectural leakage, assuming a privileged attacker. Prime+Abort [13] uses Intel-only deprecated TSX aborts as an architectural feedback mechanism. UMWAIT [80] enables cache state monitoring via architectural flags, but only for writes and only on recent Intel CPUs. PMU-Leaker [53] uses hardware counters for leakage, typically requiring privileged access. Our approach targets EL0, enabling unprivileged usage on commodity ARM systems.

Architectural CPU Vulnerabilities. Several architectural bugs have been found recently: ÆPICLeak [7] leaks superqueue contents, Zenbleed [46] leaks registers. CacheWarp [79] exploits stale cache behavior. GhostWrite [63] enables arbitrary memory writes. EntrySign [15] undermines CPU integrity via microcode. Unlike our primitives, these are specific implementation flaws, while we focus on general architectural exposure mechanisms.

9 Conclusion

We presented a fuzzing-based methodology for discovering architectural side channels that leak cache state without relying on timing measurements. The evaluation of our proof-of-concept implementation ExfilState across 160 devices with 37 unique microarchitectures revealed 5 architectural side channels, including the widely applicable Lx+Sx and Store+Ret, which exploit exclusive load-store interactions and data/instruction cache incoherence, respectively. Using the side channels, we demonstrated a timer-free Spectral attack that outperforms existing Spectre attacks and a classical AES T-table attack without timers. We also explored the potential for these architectural side channels to develop defenses against cache attacks. Our results highlight a previously underexplored class of side channels. We believe this work opens up new directions for both hardware security research and the design of future mitigations on modern CPUs.

References

[1] ARM. 2018. Arm Cortex-A76 Core Technical Reference Manual.

[2] ARM. 2018. ARMv8-A synchronization primitives.

[3] ARM. 2023. Arm Architecture Reference Manual for A-profile architecture.

[4] ARM Limited. 2018. ARM A64 Instruction Set Architecture.

[5] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES.

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*.

[7] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*.

[8] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *Cryptology ePrint Archive, Report 2006/052* (2006).

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

[10] Congcong Chen, Jinhua Cui, and Jiliang Zhang. 2024. BETA: Automated Blackbox Exploration for Timing Attacks in Processors. *arXiv:2410.16648* (2024).

[11] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects. In *USENIX Security Symposium*.

[12] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. 2024. Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *S&P*.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*.

[14] Christopher Domas. 2017. Breaking the x86 ISA. *Black Hat US* (2017).

[15] Josh Eads, Tavis Ormandy, Matteo Rizzo, Kristoffer Janke, and Eduardo Vela Nava. 2025. *Zen and the Art of Microcode Hacking.*

[16] Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A Eitel, Angelo Sapello, and Abhrajit Ghosh. 2021. Computing with time: Microarchitectural weird machines. In *ASPLOS*.

[17] Anders Fogh. 2016. Covert Shotgun: automatically finding SMT covert channels. https://cyber.wtf/2016/09/27/covert-shotgun/

[18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[19] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. 2024. Efficient and Generic Microarchitectural Hash-Function Recovery. In *S&P*.

[20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[22] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*.

[23] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2019. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv:1907.03651* (2019).

[24] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*.

[25] Lorenz Hetterich, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, Nils Bernsdorf, Eduard Ebert, and Michael Schwarz. 2025. ShadowLoad: Injecting State into Hardware Prefetchers. In *ASPLOS*.

[26] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. 2023. Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions. In *USENIX Security*.

[27] Gal Horowitz, Eyal Ronen, and Yuval Yarom. 2024. Spec-o-Scope: Cache Probing at Cache Speed. In *ACM CCS*.

[28] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *MICRO*.

[29] Wei-Ming Hu. 1992. Reducing Timing Channels with Fuzzy Time. *Journal of Computer Security* (1992).

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conference on Digital System Design*.

[31] Dougall Johnson. 2021. Apple Firestorm/Icestorm CPU microarchitecture docs. https://github.com/dougallj/applecpu

[32] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *USENIX Security Symposium*.

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

[34] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2023. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *USENIX Security*.

[35] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. 2024. SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters?. In *AsiaCCS*.

[36] William K Lam. 2008. *Hardware design verification: simulation and formal method-based approaches.* Prentice Hall PTR.

[37] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security*.

[38] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[39] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.

[40] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution.

arXiv:1902.05178 (2019).
[41] William M McKeeman. 1998. Differential testing for software. (1998).
[42] Memory Ordering [n. d.]. Intel 64 Architecture Memory Ordering White Paper.
[43] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium.*
[44] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*
[45] Tavis Ormandy. 2023. Reptar. https://lock.cmpxchg8b.com/reptar.html
[46] Tavis Ormandy. 2023. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html
[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA.*
[48] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan.*
[49] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security.*
[50] Thomas Pornin. 2022. Why Constant-Time Crypto? https://www.bearssl.org/constanttime.html
[51] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. 2023. ShowTime: Amplifying arbitrary CPU timing side channels. In *AsiaCCS.*
[52] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS.*
[53] Pengfei Qiu, Qiang Gao, Dongsheng Wang, Yongqiang Lyu, Chunlu Wang, Chang Liu, Rihui Sun, and Gang Qu. 2023. PMU-Leaker: Performance monitor unit-based realization of cache side-channel attacks. In *ASP-DAC.*
[54] Fan Sang, Jaehyuk Lee, Xiaokuan Zhang, Meng Xu, Scott Constable, Yuan Xiao, Michael Steiner, Mona Vij, and Taesoo Kim. 2024. SENSE: Enhancing Microarchitectural Awareness for TEEs via Subscription-Based Notification. In *NDSS.*
[55] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. 2023. FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers. In *CCS.*
[56] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS.*
[57] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC.*
[58] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. 2022. Robust and Scalable Process Isolation against Spectre in the Cloud. In *ESORICS.*
[59] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. Silifuzz: Fuzzing cpus by proxy. *arXiv:2110.11519* (2021).
[60] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security.*
[61] Stephen Röttger and Artur Janc. 2021. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html
[62] Fredrik Strupe and Rakesh Kumar. 2020. Uncovering hidden instructions in Armv8-A implementations. In *HASP.*
[63] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. 2025. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. In *CCS.*
[64] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (July 2010), 37–71.
[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium.*
[66] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium.*
[67] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. 2017. RevAnC: A framework for reverse engineering hardware page table caches. In *Proceedings of the 10th European Workshop on Systems Security.*
[68] Pepe Vila, Boris Köpf, and Jose Morales. 2019. Theory and Practice of Finding Eviction Sets. In *S&P.*
[69] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *USENIX Security.*
[70] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS.*
[71] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security.*
[72] Daniel Weber, Leonard Niemann, Lukas Gerlach, Jan Reineke, and Michael Schwarz. 2024. No Leakage Without State Change: Repurposing Configurable CPU Exceptions to Prevent Microarchitectural Attacks. In *ACSAC.*
[73] Daniel Weber, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. 2023. Indirect Meltdown: Building Novel Side-Channel Attacks from Transient Execution Attacks. In *ESORICS.*
[74] Wikichip. 2012. Cortex-A53 - Microarchitectures - ARM. https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a53
[75] Wikichip. 2021. Neoverse V1 - Microarchitectures - ARM. https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_v1
[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P.*
[77] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium.*
[78] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. 2023. Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *USENIX Security.*
[79] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security.*
[80] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security.*
[81] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security Symposium.*

```
BLRA, BLR, BRA, BR, RETA, RET, LD1_ADVSIMD_MULT, LD2_ADVSIMD_MULT,
LD3_ADVSIMD_MULT, LD4_ADVSIMD_MULT, ST1_ADVSIMD_MULT, ST2_ADVSIMD_MULT,
ST3_ADVSIMD_MULT, ST4_ADVSIMD_MULT, LD1_ADVSIMD_MULT, LD2_ADVSIMD_MULT,
LD3_ADVSIMD_MULT, LD4_ADVSIMD_MULT, ST1_ADVSIMD_MULT, ST2_ADVSIMD_MULT,
ST3_ADVSIMD_MULT, ST4_ADVSIMD_MULT, LD1R_ADVSIMD, LD1_ADVSIMD_SNGL,
LD2R_ADVSIMD, LD2_ADVSIMD_SNGL, LD3R_ADVSIMD, LD3_ADVSIMD_SNGL, LD4R_ADVSIMD,
LD4_ADVSIMD_SNGL, ST1_ADVSIMD_SNGL, ST2_ADVSIMD_SNGL, ST3_ADVSIMD_SNGL,
ST4_ADVSIMD_SNGL, LD1R_ADVSIMD, LD1_ADVSIMD_SNGL, LD2R_ADVSIMD,
LD2_ADVSIMD_SNGL, LD3R_ADVSIMD, LD3_ADVSIMD_SNGL, LD4R_ADVSIMD,
LD4_ADVSIMD_SNGL, ST1_ADVSIMD_SNGL, ST2_ADVSIMD_SNGL, ST3_ADVSIMD_SNGL,
ST4_ADVSIMD_SNGL, CASB, CASH, CAS, CASP, LDAPURB, LDAPURH, LDAPURSB,
LDAPURSH, LDAPURSW, LDAPUR_GEN, STLURB, STLURH, STLUR_GEN, LDRB_IMM,
LDRH_IMM, LDRSB_IMM, LDRSH_IMM, LDRSW_IMM, LDR_IMM_GEN, LDR_IMM_FPSIMD,
STRB_IMM, STRH_IMM, STR_IMM_GEN, STR_IMM_FPSIMD, LDRB_IMM, LDRH_IMM,
LDRSB_IMM, LDRSH_IMM, LDRSW_IMM, LDR_IMM_GEN, LDR_IMM_FPSIMD, STRB_IMM,
STRH_IMM, STR_IMM_GEN, STR_IMM_FPSIMD, LDRA, LDRB_IMM, LDRH_IMM, LDRSB_IMM,
LDRSH_IMM, LDRSW_IMM, LDR_IMM_GEN, LDR_IMM_FPSIMD, STRB_IMM, STRH_IMM,
STR_IMM_GEN, STR_IMM_FPSIMD, LDRB_REG, LDRH_REG, LDRSB_REG, LDRSH_REG,
LDRSW_REG, LDR_REG_GEN, LDR_REG_FPSIMD, STRB_REG, STRH_REG, STR_REG_GEN,
STR_REG_FPSIMD, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, LDTR, STTRB, STTRH,
STTR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, LDUR_GEN, LDUR_FPSIMD, STURB,
STURH, STUR_GEN, STUR_FPSIMD, LDAXP, LDXP, STLXP, STXP, LDAXRB, LDAXRH,
LDAXR, LDXRB, LDXRH, LDXR, STLXRB, STLXRH, STLXR, STXRB, STXRH, STXR,
LDNP_GEN, LDNP_FPSIMD, STNP_GEN, STNP_FPSIMD, LDARB, LDARH, LDAR, LDLARB,
LDLARH, LDLAR, STLLRB, STLLRH, STLLR, STLRB, STLRH, STLR, LDPSW, LDP_GEN,
LDP_FPSIMD, STP_GEN, STP_FPSIMD, LDPSW, LDP_GEN, LDP_FPSIMD, STP_GEN,
STP_FPSIMD, LDPSW, LDP_GEN, LDP_FPSIMD, STP_GEN, STP_FPSIMD, LDADDB, LDADDH,
LDADD, LDCLRB, LDCLRH, LDCLR, LDEORB, LDEORH, LDEOR, LDSETB, LDSETH,
LDSET, LDSMAXB, LDSMAXH, LDSMAX, LDSMINB, LDSMINH, LDSMIN, LDUMAXB,
LDUMAXH, LDUMAX, LDUMINB, LDUMINH, LDUMIN, SWPB, SWPH, SWP
```

Listing 6: List of the 198 memory instruction mnemonics that ExfilState discovers and uses on Apple M1.

A Used Instructions

Listing 6 lists the 198 memory instruction mnemonics discovered and used on Apple M1.

Table 3: Overview of which tested microarchitecture is affected by which side channel. ✓ indicates a bit-error rate of at most 25 % in the covert channel verification, while ∼ indicates a higher rate. Exponents show the complexity of the discovered sequence in the form of the sequence length. The first per-column number is the bit-error rate (%), the second is the F-score (%) in the verification stage. The last full column shows if a microarchitecture is affected by any of the 5 side channels.

Microarchitecture	Lx+Sx			Store+Ret			Split-Store			Translation-Race			Pointer-Chase			Affected
Cortex-A53	0.00	100.00	✓^{2}	0.00	100.00	✓^{2}							0.00	99.42	✓^{3}	✓
Cortex-A55	0.00	99.86	✓^{4}	0.00	100.00	✓^{3}										✓
Cortex-A510	19.63	86.31	✓^{4}	1.27	98.27	✓^{3}										✓
Cortex-A520	16.90	80.59	✓^{5}	1.47	96.85	✓^{3}										✓
Cortex-A72	0.00	100.00	✓^{2}										0.00	99.94	✓^{2}	✓
Cortex-A73	0.00	100.00	✓^{2}	27.45	88.44	∼^{2}	0.00	99.70	✓^{2}	0.40	98.71	✓^{2}				✓
Cortex-A75	0.00	99.05	✓^{3}				0.00	99.57	✓^{2}	0.00	100.00	✓^{2}				✓
Cortex-A76	7.04	92.90	✓^{3}	0.00	99.92	✓^{2}										✓
Cortex-A77				0.00	99.81	✓^{3}										✓
Cortex-A78				0.00	99.77	✓^{2}										✓
Cortex-A710				0.00	100.00	✓^{3}										✓
Cortex-A715	0.00	99.90	✓^{3}	0.00	100.00	✓^{3}										✓
Cortex-A720	0.00	94.46	✓^{3}	0.00	99.74	✓^{3}										✓
Cortex-A725	0.00	88.43	✓^{3}	0.00	99.74	✓^{2}										✓
Cortex-X1				0.00	99.89	✓^{3}										✓
Cortex-X2				0.00	99.94	✓^{3}										✓
Cortex-X3				0.20	99.44	✓^{3}										✓
Cortex-X4				0.00	99.98	✓^{3}										✓
Kryo																
Falkor-V1/Kryo	0.00	99.85	✓^{2}				0.00	99.61	✓^{2}	0.20	98.21	✓^{2}				✓
Kryo-V2	0.20	99.83	✓^{3}	0.00	99.36	✓^{2}										✓
Kryo-3XX-Gold	0.10	99.00	✓^{4}	35.45	98.43	∼^{2}	0.00	99.50	✓^{2}	0.00	99.56	✓^{2}				✓
Kryo-3XX-Silver	30.57	95.93	∼^{4}	1.27	95.55	✓^{3}										✓
Kryo-4XX-Gold				0.00	99.93	✓^{2}										✓
Kryo-4XX-Silver	1.96	98.92	✓^{4}	0.98	98.41	✓^{3}										✓
Carmel				4.59	97.46	✓^{3}										✓
Kunpeng Pro				0.00	100.00	✓^{2}										✓
Oryon	0.10	99.26	✓^{3}	0.10	96.19	✓^{4}										✓
Oryon V2 Phoenix L	0.49	83.42	✓^{3}	0.00	98.81	✓^{2}										✓
Oryon V2 Phoenix M	0.00	81.40	✓^{3}	0.00	99.00	✓^{3}										✓
Exynos M3	1.57	97.54	✓^{4}	2.06	99.77	✓^{3}										✓
Neoverse-N1				0.00	99.89	✓^{2}										✓
Neoverse-V2				0.00	99.88	✓^{2}										✓
Firestorm-M1	0.00	98.52	✓^{4}	0.00	80.47	✓^{2}										✓
Icestorm-M1	7.43	90.80	✓^{5}	0.00	99.88	✓^{4}										✓
Avalanche-M2	5.47	90.15	✓^{5}	0.59	84.46	✓^{2}										✓
Blizzard-M2	45.32	80.79	∼^{8}	5.28	93.27	✓^{3}										✓

```
1 STTRH      w1, [x2]
2 LDAXRH     w2, [x1]
3 LDRSB      w0, [x3]
4 STLXRB w0, w3, [x1]
```

Listing 7: Lx+Sx on Apple Avalanche. x1 is the victim cache line. x2 and x3 are unrelated cache lines. Instructions 1 and 3 are not needed on Cortex-A73.

B Full Results

Table 3 shows which microarchitectures are affected by which side channel, with the error rate and F-score of the verification stage.

C Example of Complexity

Listing 7 shows an example of Lx+Sx, where the short sequence works on Cortex-A73, while only the longer sequence works on Apple Avalanche.

Table 4: List of devices used for the evaluation of ExfilState.

Manufacturer	Models
Apple	Mac Mini (M1), Mac Mini (M2)
ASUS	ZenFone Max M1, ZenFone Max M2
AWS	EC2 M8g Graviton4
FriendlyELEC	NanoPi R6S
Fujitsu	Arrows Be3
Globalscale	MOCHAbin
Google	Pixel Watch, Pixel Watch 2, Pixel, Pixel 2, Pixel 2 XL, Pixel 3, Pixel 4a, Pixel 5, Pixel 5a, Pixel 6, Pixel 6 Pro, Pixel 6a, Pixel 7, Pixel 7 Pro, Pixel 7a, Pixel 8, Pixel 8 Pro, Pixel 8a, Pixel 9, Pixel 9 Pro, Pixel 9 Pro Fold, Pixel 9 Pro XL, Pixel Fold, Pixel Tablet
Google Cloud	C4A Axion, T2A Ampere
Hardkernel	ODROID-N2+, Odroid-C4
HUAWEI	P20 Lite, Mate 9
Lenovo	Tab P11, Tab P12
LG	Velvet
Motorola	Moto G31, Moto G54 5G, Moto G30, Moto G 5G (2022), Moto G Play (2024), Razr 5G, Razr+ (2024), Moto Z Force
Nokia	C31
Nothing	Phone (1)
NVIDIA	Jetson Orin Nano
OnePlus	10T, 11, Nord CE 3 Lite, 7 Pro (US Version), 6T, Nord 2T, Nord 3, Ace 2V, 9 Pro
OPPO	A53, A79 5G, Find X3 Lite, Reno10 Pro+ 5G
OrangePi	Kunpeng Pro
Poco	X6 Pro, M6 Pro 5G, F3
Qualcomm	Dell XPS 13
Radxa	ROCK 3A, ROCK 5C
Raspberry Pi	4 Model B, 5
Realme	GT Neo 3T, GT Master Edition
Samsung	Galaxy S7 edge (Verizon), Galaxy Note 5 (Verizon), Galaxy Tab S3 (Verizon), Galaxy A02s, Galaxy A8, Galaxy A10, Galaxy A12, Galaxy A14 5G, Galaxy A14 5G, Galaxy A15, Galaxy A25 5G, Galaxy A35 5G, Galaxy A51, Galaxy A52s 5G, Galaxy A54 5G, Galaxy Z Flip 3 5G, Galaxy Z Flip 4, Galaxy Z Flip 5, Galaxy Z Fold 2 5G, Galaxy Z Fold 4, Galaxy Z Fold 5, Galaxy J7 Prime, Galaxy A51 5G, Galaxy S8, Galaxy S8+, Galaxy S9, Galaxy S9, Galaxy S9+, Galaxy S20 5G, Galaxy S21 5G, Galaxy S21+ 5G, Galaxy S21 Ultra 5G, Galaxy M11, Galaxy Note 9, Galaxy Note 9, Galaxy S23 FE, Galaxy S22 5G, Galaxy S22+ 5G, Galaxy S22 Ultra 5G, Galaxy S23+, Galaxy S23 Ultra, Galaxy S23 Ultra, Galaxy S24, Galaxy S24 Ultra, Galaxy S24 Ultra, Galaxy Tab A7 Lite LTE, Galaxy Tab A 10.1 (2016) Wi-Fi, Galaxy Tab S7 FE 5G, Galaxy Tab A8 Wi-Fi, Galaxy Tab A9 Wi-Fi, Galaxy Tab S9 FE+, Galaxy Tab S7 Wi-Fi, Galaxy Tab S8 Ultra Wi-Fi
Sharp	AQUOS sense2
Sony	Xperia XZ1 Compact, Xperia 10 V, Xperia 1 V, Xperia XZ, Xperia 10 II
Vivo	Y20s [G], Y31 (2021), Y73, X60, Y02s, Y95, Y12, Y15, U3x, Y17
Xiaomi	Mi 11i, Mi A2 Lite, Redmi Note 13 Pro+ 5G, Redmi Note 11 5G, Redmi Note 11 Pro, Redmi Note 12, Redmi Note 12 Pro+
Zebra	TC77

D Devices

Table 4 lists which devices we used for the evaluation of ExfilState.