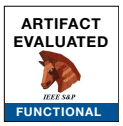

RISCy Cache Coherence: Timer-Free Architectural Cache Attacks via Instruction/Data Cache Incoherence

Fabian Thomas
CISPA Helmholtz Center
for Information Security

Michael Schwarz
CISPA Helmholtz Center
for Information Security

***Abstract*—Caches have long been known to leak information across isolation boundaries, with classic attacks relying on timing to distinguish cache hits and misses. However, modern CPUs and operating systems increasingly limit timer resolution or restrict access to cycle counters, making such attacks less reliable in practice. As an alternative, architectural side channels replace timing with instruction sequences whose architectural outcome depends on cache state, offering higher robustness.**

In this paper, we introduce I^{2}SC, a generic timer-free architectural cache side channel that exploits instruction/data-cache incoherence on RISC-V, ARM, and LoongArch. I^{2}SC leverages a widespread RISC property: stores through the data path are invisible to instruction fetch when the instruction cache holds stale lines, yielding architecturally distinct outcomes that reveal cache state. Unlike prior work that targets only instruction caches, I^{2}SC generalizes this behavior into a timer-free oracle for both instruction and data caches via a transient-execution-based cache-state transfer gadget. We evaluate I^{2}SC on 18 microarchitectures, finding that 12 microarchitectures are affected. To demonstrate the security impact of I^{2}SC, we mount three end-to-end attacks: a timer-free AES key-recovery, a Spectre variant with architectural leakage across all three architectures, achieving reliability on par with or exceeding prior timing-based methods, and a classical side-channel attack on Android shared libraries recovering fine-grained touch-event timing. Finally, we discuss both software and hardware mitigations, noting that full prevention likely requires hardware changes.

1. Introduction

Caches are well-known for side-channel leakage across isolation boundaries. A large body of work has shown how subtle differences in cache state can reveal user behavior [1]–[3] and, in cryptographic settings, secret keys [4]–[7]. Classic techniques measure execution latency to classify cache hits and misses and then compose these primitives into full attacks. However, the reliance on high-resolution timers or cycle counters has become an increasingly brittle assumption: vendors throttle timer precision [8], [9], virtualized environments add noise [10], [11], and some platforms restrict user access to cycle counters [2], [12], [13]. While these defenses do not remove the underlying leakage, they make it harder to mount timing-based attacks. As a motivational example, Figure 1 shows the access-time histogram for cache hits and misses on a modern (2025) ARM CPU, the Qualcomm Snapdragon X. The timer resolution is too low to reliably distinguish cache hits and misses, making cache attacks challenging to mount [2], [13], and also impeding reverse engineering [14]. While attackers can emulate timing through counting loops or parallel threads [2], [13], [15], [16], such workarounds complicate attacks, depend on stable scheduling, and remain noisy on low-power or virtualized systems. This motivates timer-free cache attacks, which infer cache state without any timing measurements.

Prior work demonstrated that *architectural side channels*—rather than timing measurements—can reveal cache state [17]–[23]. Instead of reading out cache state through latency, these techniques craft instruction sequences whose *architectural* outcome depends on whether a cache line is present. In effect, they replace the timer with an architectural oracle, providing a drop-in replacement for many timing-based building blocks. Prior work established the practicality of architectural channels for cache attacks, yielding higher reliability precisely because they avoid noisy timing measurements. Cache Storage Channels (CSC) [20] showed *privileged* timer-free channels on ARM TrustZone that exploit self-modifying code (SMC) or mismatched caching attributes. Synchronization Storage Channels (S^{2}C) [23] later described timer-free channels using synchronization primitives in combination with the specific design of Apple M-series CPUs. More recently, GhostCache [22] demonstrated timer-free I-cache state leakage on ARM and RISC-V CPUs via SMC sequences similar to those of CSC. Further, concurrent work ExfilState [21] proposed a methodology for automatically discovering such timer-free channels on ARM.

In this paper, we systematically study instruction/data-cache incoherence and show that the phenomenon is not limited to individual cores. Across a broad set of recent ARM, RISC-V, and LoongArch microarchitectures, we find that instruction and data caches are not kept coherent under certain SMC sequences, producing a consistent architectural signal exploitable by unprivileged attackers. Stores performed through the data path are not necessarily visible to the instruction fetch unit (IFU) until software executes explicit cache maintenance and barriers. We introduce I^{2}SC (Instruction-Cache Incoherence Side Channel), a generalization of previous work, turning this "incoherence window" into a binary oracle on the state of the I-cache. By

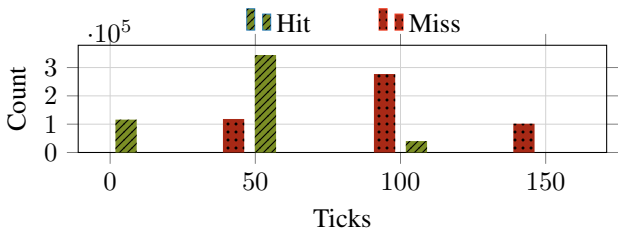

Figure 1: Flush+Reload on the Qualcomm Snapdragon X (X1-26-100) Oryon core. The timer resolution is too low to reliably distinguish cache hits and misses using timing.

overwriting an executable cache line and then transferring control to it, the CPU either executes stale instructions (if the old bytes are still resident in L1i) or the updated bytes (if the IFU observes the store). Which sequence runs is an architectural event we can observe without timing, e.g., by checking a register. While prior work has leveraged this behavior to reveal *instruction*-cache residency to mount control-flow attacks, the applicability to data flow remains unexplored. However, many real-world targets of interest leak side-channel information via their data flow.

I^{2}SC bridges this gap, by introducing a transient-execution-based *transfer* gadget, which transfers the state of an arbitrary *data*-cache line to an attacker-controlled *instruction*-cache line. This transfer gadget acts as a portable, timer-free bridge from D-cache state to the architectural domain. Unlike prior "weird gates" [24]–[27] used to amplify timing differences, our construction yields a direct architectural outcome without relying on any timing measurements. This combination results in an *unprivileged* timer-free probe for *data* accesses, allowing I^{2}SC to serve as a general-purpose replacement for classic timing-based cache attacks on shared memory, such as Flush+Reload [6] or Flush+Flush [28].

A central challenge when implementing I^{2}SC is portability. Despite I^{2}SC being an architectural side channel, it still relies on microarchitectural details, such as out-of-order instruction fetch that can transiently pull in lines, forwarding between the data side and the IFU, and pipeline arbitration between "stale" L1i bytes and newly produced stores. We systematize the requirements of I^{2}SC into two building blocks that can be tested on a given microarchitecture: (1) Forwarding from data to instruction fetch, i.e., the microarchitecture must offer a way for the IFU to "see" stores or data-side state without immediate software-visible cache maintenance. (2) Transient instruction-side population, i.e., the core must be capable of out-of-order execution that allows cache lines to be transiently brought into structures visible to the IFU (e.g., L1i or an IFU-adjacent queue).

We survey 18 microarchitectures across RISC-V, ARM, and LoongArch and instantiate tests for both properties using an automated test harness. We find that 16 microarchitectures satisfy the first property, i.e., allow for architecturally leaking I-cache state, and 13 out-of-order designs provide a usable transient transfer gadget. Taken together, 12 of the 18 microarchitectures we tested are affected by I^{2}SC, spanning LoongArch, RISC-V, and ARM designs. This demonstrates that I^{2}SC is relevant beyond a single CPU family or ISA and arises from common microarchitectural mechanisms rather than ISA-specific behavior. We find that most affected CPUs are new, high-performance designs (e.g., Cortex-X4 used in high-end smartphones and Neoverse-N1 used in the cloud), indicating that this side channel can potentially affect even more CPUs in the future. We release our test harness to facilitate independent verification and to help vendors and researchers assess their platforms.

We demonstrate three end-to-end attacks that illustrate the versatility of I^{2}SC. First, we build a timer-free key-recovery attack against the OpenSSL AES implementation that uses T-tables. The attacker primes or observes the tables and queries the cache state through I^{2}SC. As the readout is architectural, the attack remains stable despite high system load on all three ISAs. The resulting leakage rates match or exceed prior timing-based baselines while not requiring timing primitives. Second, we construct an architectural Spectre variant ("Spectral" [18]) on all three ISAs. Here, speculation transiently encodes victim secrets into the cache state. Rather than measuring latency, we perform architectural readouts using I^{2}SC. This attack shows that Spectral attacks apply to ARM, RISC-V, and LoongArch, not only x86 [18]. Third, we demonstrate a classical side-channel attack against shared Android libraries using I^{2}SC, recovering fine-grained touch-event timing, such as taps and swipes, on a Google Pixel 9 running latest Android 16.

The most direct mitigation is already present in most ISAs: prevent self-modifying-code sequences by not allowing users to map a memory location as writable and executable. Unfortunately, enforcing this policy can penalize legitimate use cases such as JIT compilation, hot patching, and dynamic linking. A more viable solution is preventing the transient forwarding in the hardware, fully mitigating this side channel. However, this requires the redesign of CPUs with potential performance impacts. Generally, I^{2}SC underscores that defenses focused solely on timers leave a large attack surface intact. Architectural oracles provide adversaries with new levers that remain effective despite a large focus on making high-resolution timers unavailable to attackers [8], [9], [15], [29].

Contributions. We summarize our contributions as follows:

- We propose I^{2}SC, an unprivileged timer-free architectural cache side channel that exploits instruction/data cache incoherence on RISC-V, ARM, and LoongArch.
- We identify two main building blocks for I^{2}SC and evaluate their prevalence on 18 microarchitectures, finding that 12 microarchitectures are affected by I^{2}SC, including recent high-performance cores deployed in smartphones, servers, and domestic ISAs.
- We present Spectral attacks and architectural side-channel attacks on AES T-tables on all three ISAs. Further, we show an architectural side-channel attack on shared Android libraries recovering touch-event timing, such as taps and swipes, on the Google Pixel 9 running Android 16.

Responsible Disclosure. We disclosed our findings to Arm, Loongson, and T-Head. Arm and Loongson reproduced the attack but do not plan design changes. T-Head acknowledged the attack and plans to fix it in future designs.
Availability. I^{2}SC and all related artifacts are open-source, at: https://github.com/cispa/RISCy-cache-coherence.

2. Background

2.1. CPU Caches

Modern CPUs hide main-memory latency behind small, fast caches arranged in a hierarchy. The first level typically comprises a private instruction cache (L1i) and a private data cache (L1d), with deeper levels (L2/L3) shared within a core or among cores depending on the design. Many RISC cores implement split instruction and data caches without automatic coherence on self-modifying code: a store via the data path is not guaranteed to be seen by the instruction fetch unit (IFU). Thus, self-modifying code, as used in, e.g., just-in-time (JIT) compilers, requires software to perform explicit cache maintenance and barriers (e.g., data/instruction invalidation plus `DSB`/`ISB` on ARM) [30]. Otherwise, this leads to windows where the IFU can execute stale instructions from L1i while L1d already holds updated bytes. Still, microarchitectures might also provide mechanisms to forward written data to the IFU automatically, e.g., through store-to-IFU forwarding or IFU lookup into L1d. However, microarchitectures differ in how aggressively this forwarding is implemented, if it is implemented at all.

2.2. Cache Attacks

Cache side channels infer a victim's (secret-dependent) memory-access pattern by distinguishing cache hits from misses. Such side channels have been used in various attacks, such as inferring cryptographic keys [4]–[7], spying on user behavior [2], [3], or breaking ASLR [31]. Classic examples of cache attacks include Flush+Reload [6] and Prime+Probe [32], which rely on accurate timing to classify access latency. When high-resolution timers are restricted or unavailable, which is common for unprivileged code on many ARM64 systems [2], [13], timing-based attacks become less portable and less reliable. An emerging alternative are *architectural* side channels: instruction sequences that make a cache condition visible in architecturally observable state such as register values, memory contents, or exception metadata, eliminating the need for fine-grained timers. Prior work has shown that such timer-free primitives exist on x86 [17]–[19], [33], [34]. Such architectural cache side channels act as drop-in replacements for timing primitives in standard cache attacks.

2.3. Transient Execution

Speculative and out-of-order execution can transiently execute instructions that are never committed architecturally (e.g., due to mispredicted branches) [35]. Although architectural state is rolled back, microarchitectural traces (cache fills, predictor state) remain and are inferred using a side channel. Classical transient execution attacks include Spectre [36], Meltdown [37], and MDS attacks [35], [38], [39]. In these attacks, the used side channel is typically a timing-based cache attack. Architectural side channels remove this timing step: the transiently created microarchitectural trace is converted into a register or memory difference by a carefully chosen instruction sequence, e.g., enabling timer-free Spectre-style attacks ("Spectral" [18]). Prior work demonstrates that this approach can outperform classic timing channels on real devices precisely because it avoids noisy timer measurements.

Transient execution has also been used as a building block in other attacks. A recent line of work used transient execution to amplify cache attacks [24], [27], enabling attacks with lower-resolution timers. A different line of work relies on transient execution for safely testing whether a memory access is architecturally valid before accessing the memory location as part of an exploit [40], [41].

2.4. LoongArch

LoongArch is a 64-bit general-purpose ISA developed by China's Loongson Technology Corporation to reduce dependence on foreign processor IP and is supported by mainstream toolchains and Linux. From a microarchitectural perspective, LoongArch CPUs inherit many design choices that make other modern processors vulnerable to side-channel attacks, including shared last-level caches between all CPU cores, hyperthreading, and out-of-order execution. Like other contemporary RISC designs, LoongArch cores typically feature split L1 instruction and data caches. LoongArch CPUs do not expose unprivileged cache maintenance instructions commonly used for side-channel attacks.

3. I^{2}SC: Leaking Cache State Architecturally

In this section, we present I^{2}SC, an unprivileged timer-free architectural cache side channel that targets both the instruction and data cache. I^{2}SC exploits a fundamental difference between x86 and many RISC designs. While x86 guarantees coherence between instruction and data caches on self-modifying code, common RISC ISAs provide no such guarantee. As a result, stores through the data path can leave the instruction cache with stale lines until software explicitly performs cache maintenance. We experimentally confirm that *all* tested RISC microarchitectures can be driven into such incoherent states with self-modifying code.

Prior work has used this incoherence to infer I-cache evictions by observing whether stale or updated instructions execute under self-modifying code [20], [22]. We formalize this behavior as our first building block ($\mathcal{B}1$), an architectural oracle that reports whether a target line resides in the I-cache. Our automated test harness discovers the microarchitecture-specific parameters for triggering this

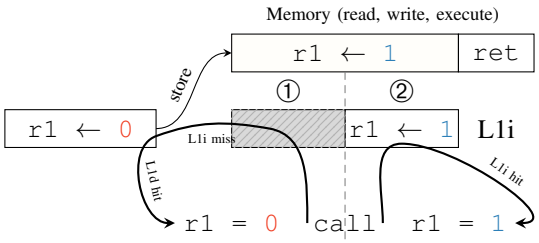

Figure 2: The first building block ($\mathcal{B}1$) of I^{2}SC: the code to set a register to '1' is stored in attacker-controlled memory with read, write, and execute permissions. This memory location is overwritten with code to set the register to '0'. If the code was not cached before ①, a call to this memory location leads to a cache miss in the instruction cache, and the new instruction is used from the data cache. If the code was cached ②, the call uses the original instruction. Thus, after the call, the register value depends on the state of the instruction cache for this cache line.

oracle across ARM, RISC-V, and LoongArch, and shows that it is available on 16 of 18 tested microarchitectures.

However, many real-world targets leak secrets through their data flow rather than their control flow. I^{2}SC closes this gap by introducing a transient-execution-based *transfer* gadget ($\mathcal{B}2$) that copies the residency state of an arbitrary D-cache line into an attacker-controlled I-cache line. Combined with $\mathcal{B}1$, this yields a generic timer-free D-cache oracle. On affected microarchitectures, I^{2}SC thus becomes a drop-in replacement for classic unprivileged timing-based cache primitives such as Flush+Reload, but with architectural outcomes instead of noisy latency measurements.

3.1. $\mathcal{B}1$: Architectural I-Cache Oracle

Previous work finds that when an executable line is present in L1i and we overwrite its bytes via the data path, instruction fetch may still execute the *stale* line unless software performs cache maintenance and barriers [20], [22]. Conversely, if L1i does not hold the line, the front end may fetch the *updated* bytes (either by consulting L1d or store-to-IFU forwarding). When choosing unique instructions mapped as code and overwritten as data, it is architecturally observable what is executed. Thus, with the right scheduling window, the outcome correlates one-to-one with the I-cache residency of the line.

Figure 2 illustrates the basic principle: an attacker-controlled read-write-execute page contains a target instruction `mov r1, '1'` (the *stale* encoding) followed by a return instruction. This target instruction architecturally sets the register `r1` to '1'. We combine it with a return instruction because it makes the overwritten RWX sequence a tiny callable function whose architectural result is easy to observe in a register. Immediately before transferring control to this location, we overwrite those bytes with `mov r1, '0'` (the *updated* encoding). This updated instruction sets the register `r1` to '0'. After the call returns, we get the following architectural result:

$$\texttt{r1} = \begin{cases} 1, & \text{stale L1i line executed (I-cache hit)} \\ 0, & \text{updated bytes executed (I-cache miss)} \end{cases}$$

This yields a timer-free readout of the line's I-cache state.

The primitive relies on two behaviors that are common on modern RISC cores: (1) after a store to code, the IFU can still fetch from L1i unless software performs explicit maintenance, and (2) the frontend can see updated bytes late enough to win the race if L1i does not hold the line (e.g., via L1d snooping or store-to-IFU forwarding). Prior work relied on hand-crafted instruction sequences to create such a race and are as such not portable to other microarchitectures [20], [22]. In contrast, we construct an automated test harness which automatically discovers the microarchitecture-specific parameters to trigger such race in which the presence of the stale line in L1i decides the outcome on the 3 main RISC ISAs—ARM, RISC-V, and LoongArch. This test harness automatically creates and optimizes a short window between the store and control-flow transfer by placing instructions between them, creating a race between store and instruction fetch. We find that a small number of `nops` between store and control flow are sufficient on most microarchitectures, while others like the T-Head XuanTie C910 (RISC-V) require an additional memory barrier. In total, we find a valid race configuration for 16 of the 18 tested microarchitectures, demonstrating that our automatically inferred $\mathcal{B}1$ is highly versatile and portable.

3.2. $\mathcal{B}2$: Cache-State Transfer Gadget

Many real-world targets of interest leak side-channel information via their data flow. However, $\mathcal{B}1$ fundamentally exposes only instruction-cache state: the architectural outcome depends solely on whether the frontend uses a stale or updated instruction line.

To bridge this gap, we introduce a transient execution gadget that *transfers* the state of an arbitrary data-cache line into the instruction path similar to transient-execution-based D-cache re-encoding or amplification techniques [21], [24], [27]. After transferring the residency state from the target D-cache line to an attacker-controlled I-cache line, we use $\mathcal{B}1$ to architecturally recover the I-cache residency state.

Gadget. Our cache-state transfer gadget (CTG) is illustrated in Figure 3. Like previous work, we open a transient window using return-stack-buffer-based misspeculation [21], [42], [43] ①② and place at its beginning a dependency-creating load from the victim address ③ [21]. If the victim line is a data-cache hit, the dependency resolves quickly and the front end transiently fetches from an attacker-controlled read-write-executable page (the oracle page) ④. If the victim line is a miss, the window is not long enough for the fetch, and the oracle page is not brought into the I-cache. After rollback, we architecturally query the oracle page's I-cache state with $\mathcal{B}1$. Thus, the presence of the read-only victim line in L1d is converted into the architectural output of $\mathcal{B}1$. Conceptually, CTG is related to the "buffer gate" from prior

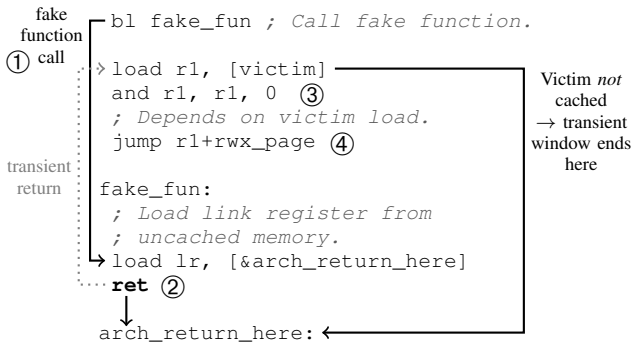

Figure 3: Overview of our transient transfer gadget ($\mathcal{B}2$). A fake function call is used to prepare RSB misspeculation ①. A fake architectural return/link address is loaded from uncached memory ②. As the load is slow, and the RSB is poisoned by the previous call, the microarchitecture transiently executes instructions following the fake function call ③. The jump to the RWX page depends on the victim cache line state ④. If the load is fast, the transient instruction fetch of `rwx_page` succeeds; if it is slow, the transient window ends. Architecturally, the execution resumes at `arch_return_here` and encodes the victim cache state in the I-cache state of `rwx_page`.

work [24], but differs in two key aspects: (i) it transfers state into the instruction cache rather than remaining in the data cache, and (ii) it uses RSB-based misspeculation to avoid microarchitecture-specific fine-tuning.

3.3. I²SC: Combining $\mathcal{B}1$ and $\mathcal{B}2$

On affected microarchitectures, I²SC combines the two building blocks to form a generic architectural side channel. We first apply $\mathcal{B}2$ to encode data-cache state into the oracle page's instruction-cache state and then mount $\mathcal{B}1$ to get the state architecturally. The result is a stable, timer-free cache oracle that can be used as a drop-in replacement for standard cache-attack building blocks, while being portable across different ISAs (cf. Table 2). Figure 4 compares $\mathcal{B}1$ to a timer-based Flush+Reload implementation on a Qualcomm Snapdragon X. The architectural side channel can accurately leak the cache state, while a timer-based Flush+Reload is extremely noisy due to the low timer resolution. Section 6 demonstrates I²SC as a drop-in replacement for classical attacks, such as AES T-table key recovery.

4. Prevalence of I²SC on Modern RISC CPUs

In this section, we evaluate the prevalence of I²SC and its 2 building blocks on 18 microarchitectures and 3 ISAs.

4.1. Devices and Setup

Table 1 shows the microarchitectures we use in our experiments with their respective device name, SoC, available memory, and operating system. In total, we test on 15 devices with 18 different microarchitectures, ranging the 3 major RISC architectures, ARM, RISC-V, and LoongArch. Table 10 in Appendix B shows the special instructions we use for our experiments. On ARM and RISC-V, we use the standard data-cache flushing instructions available to an unprivileged process. On T-Head RISC-V CPUs (C906, C910, and C908), we use instructions from the `XTHeadCmo` vendor extension [44]. On LoongArch, the cache maintenance instructions are privileged. Therefore, we use a kernel module for the experiments. For the final evaluation and case studies, we use eviction of the 64 kB 4-way caches. As memory barriers, we use the standard instructions on all microarchitectures. Modern ARM devices restrict access to a high-resolution timer via `CNTVCT_EL0`, therefore, we use the `clock_gettime` kernel interface [2]. On RISC-V and LoongArch, we use the standard `rdtime` instructions to get a timestamp. Table 9 in Appendix A provides an overview of timer resolutions.

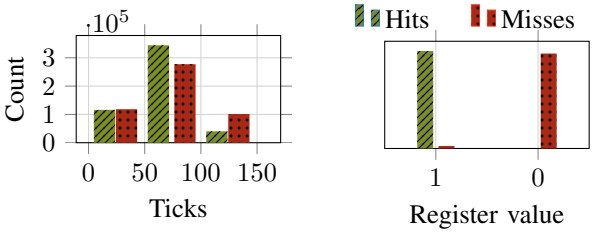

Figure 4: Flush+Reload (left) and I²SC (right) on the Qualcomm Snapdragon X (X1-26-100) ARMv8 Oryon core. The timer resolution is too low to reliably distinguish cache hits and misses using timing, whereas the architectural side channel yields reliable results.

4.2. Prevalence of $\mathcal{B}1$

The first building block of I²SC ($\mathcal{B}1$) requires the ability to introduce inconsistencies in the I-cache.

Inconsistency. We evaluate which architectures and microarchitectures allow for introducing inconsistencies between I- and D-caches. The rationale to test this first is that, if we cannot introduce any inconsistencies, there is nothing to leak. We first discuss this behavior on a theoretical level for the major ISAs: x86, ARM, RISC-V, and LoongArch. x86 guarantees that all instruction fetches following a store observe the updated instructions [45]. It is still recommended to place a barrier (`lfence`) between store and execution of self-modifying code (SMC), as instruction fetches during transient execution can still operate on stale instructions [45]. However, unlike the RISC behavior exploited in this paper, this effect is only transient rather than architecturally visible. The other major ISAs, ARM, RISC-V, and LoongArch, do not give similar guarantees as they are RISC in nature. Thus, in theory, they allow for inconsistencies between the caches.

We evaluate the prevalence of I-cache inconsistencies on all 18 tested microarchitectures. Further, we test one rep-

TABLE 1: List of used microarchitectures, with their respective ISA, device name, SoC, memory size, and operating system.

ISA	Microarchitecture	Device	SoC	RAM	OS
ARM	Cortex-A520	Google Pixel 9	Google Tensor G4	12GB	Android 16
	Cortex-A72	MOCHAbin 5G	Marvell ARMADA	8GB	Ubuntu 18.04.6 LTS
	Cortex-A73	ODROID-N2+	Amlogic S922X	4GB	Ubuntu 20.04.5 LTS
	Cortex-A76	NanoPi R6S	Rockchip RK3588S	8GB	Ubuntu 22.04.2 LTS
	Cortex-A78	Radxa NIO 12L	MediaTek Genio 1200	16GB	Rity Demo Layer 23.2-release (kirkstone)
	Cortex-A720	Google Pixel 9	Google Tensor G4	12GB	Android 16
	Cortex-A725	Poco X7 Pro	Mediatek Dimensity 8400 Ultra	8GB	Android 15
	Cortex-X4	Google Pixel 9	Google Tensor G4	12GB	Android 16
	Neoverse-N1	Ampere Altra Q64-30	Ampere Altra Q64-30	64GB	Ubuntu 24.04.2 LTS
	Oryon	Lenovo ThinkCentre Neo 50q QC	Snapdragon X (X1-26-100)	16GB	Microsoft Windows 11 Pro
	Kunpeng Pro	OrangePi Kunpeng Pro	Huawei Kunpeng	16GB	openEuler 22.03 (LTS-SP3)
	Icestorm	Apple MacBook (M1)	Apple M1	16GB	Arch Linux ARM
	Firestorm	Apple MacBook (M1)	Apple M1	16GB	Arch Linux ARM
RISC-V	C906	Lichee RV Dock	Allwinner D1	1GB	Ubuntu 24.04 LTS
	C908	youyeetoo CanMV-K230	Kendryte K230	1GB	Debian GNU/Linux trixie/sid
	C910	LicheePi 4A	T-Head TH1520	8GB	Debian GNU/Linux 12 (bookworm)
	X60	Banana Pi BPI-F3	SpacemiT K1	4GB	Armbian-bpi-SpacemiT 24.5.0-trunk sid
LoongArch	3A5000-HV	Loongson 3A5000	Loongson 3A5000	32GB	Loongnix GNU/Linux 20 (DaoXiangHu)

TABLE 2: Overview of where the building blocks for I^{2}SC are available. ◐ indicates that we can reliably leak the I-cache state of a cache line architecturally (*B*1). ◑ indicates that the microarchitecture allows for transiently bringing instructions into the I-cache with a dependency on a D-cache line (*B*2). ● indicates that both properties hold, i.e., I^{2}SC is available.

Architecture	**ARM**													**RISC-V**				**LoongArch**
Vendor	**ARM**									**Qualcomm**	**Huawei**	**Apple**		**T-Head**			**SpacemiT**	**Loongson**
Microarchitecture	**Cortex-A520**	**Cortex-A72**	**Cortex-A73**	**Cortex-A76**	**Cortex-A78**	**Cortex-A720**	**Cortex-A725**	**Cortex-X4**	**Neoverse-N1**	**Oryon**	**Kunpeng Pro**	**Icestorm**	**Firestorm**	**C906**	**C908**	**C910**	**X60**	**3A5000-HV**
Building Blocks	◐	◑	◐	●	●	●	●	●	●	●	●	●	●	○	◐	●	◐	●
I^{2}SC				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓

resentative x86 microarchitecture, Raptor Lake (Intel Core i9-13900K). We test an instruction sequence of a store instruction that overwrites the next instruction to be executed, i.e., $pc+4$. We overwrite a move instruction that originally writes '1' (i.e., stale) and, when updated, writes '0' (i.e., updated), and capture the output architecturally in a register. We ensure that the overwritten instructions are in the I-cache by executing the sequence twice. We argue that this is the most likely case to introduce inconsistencies on the microarchitectures as (a) the stale instructions are cached, and (b) the window between store and instruction fetch should be minimal. If the resulting sequence returns '1' (i.e., stale) at least once, we conclude that the microarchitecture allows for inconsistencies. All 18 RISC microarchitectures allow for inconsistencies between I- and D-caches. As expected, we do not observe this behavior on x86.

We inspect the ISA and microarchitecture manuals to confirm that this is intended behavior for 3 representative microarchitectures for each ISA: Arm Cortex-X4 (ARM), T-Head XuanTie C910 (RISC-V) and Loongson 3A5000-HV (LoongArch). The ISA and microarchitecture manuals all document that SMC is not guaranteed to work without cache-maintenance instructions. JIT compilers commonly use `__builtin___clear_cache`, exposed by the compiler, to transparently synchronize a memory range. On ARM, this function flushes the instruction caches and ensures memory ordering with memory and instruction barriers [30]. On LoongArch and RISC-V, the `FENCE.I` and `IBAR` instructions are guaranteed to synchronize the instruction-caches.

LoongArch is, however, a special case. The LoongArch manual documents that if bit `ICHMC` in the `CPUCFG` is set to 1, which is default on the Loongson 3A5000-HV, the "hardware maintains the data consistency between ICache and DCache in one processor core" [46]. However, our previous experiment shows that this is not the observed behavior. The LoongArch manual, however, documents that the `IBAR` instruction should be used to "complete the synchronization

between the store operation and the instruction fetch operation" [46]. We speculate that the semantic difference not encoded in the manual here is the difference between out-of-order instruction fetches and "in-order" instruction fetches: out-of-order, i.e., transient instruction fetches are not guaranteed to see the updated instruction as the instruction fetch is completed before the store retires. However, any instruction fetched after the store retires sees the updated instructions.

We verify that placing the respective synchronization primitives between load and execution removes inconsistencies as expected on all microarchitectures.

Architectural I-Cache Oracle. The previous experiment shows that all 18 tested RISC microarchitectures allow for inconsistencies between I- and D-caches. Now we test which microarchitectures allow for reliably leaking the cache state of I-cache lines. We prepare a similar experiment, but vary the instructions between store and execution of the updated instructions. This ensures that the microarchitecture has enough time to forward updated instructions to the IFU. We insert a memory fence instruction (cf. Table 10) and a variable number of `nop` instructions between overwriting store and execution of the oracle page. We measure the correlation of the architectural leakage, i.e., whether stale or updated instructions are executed, with the state of the I-cache, i.e., whether the stale instructions are present in the I-cache. We count false positives/negatives and true positives/negatives and compute the MCC (Matthews correlation coefficient) as a metric for correlation. In contrast to the previous experiments, we test different sequences to allow for finding the best setup for a microarchitectural race condition. Figure 5 shows this setup. We test power-of-two sequence lengths from 0 to 32 768 `nops`. We test with and without a memory fence (cf. Table 10) between store and execution. Further, we test modifying the oracle instruction in-line, i.e., the instruction directly following the store, memory fence, and `nop` sequence, and out-of-line, on a different page, using a jump to connect both instruction sequences. This experiment can be crafted rather generically, as the tested architectures feature similar instructions, and encode instructions into 4 B.

Figure 6 shows the results of this experiment using the configuration with a jump and memory fence on the remaining microarchitectures. We plot the MCC, 1 for perfect correlation, 0 for no correlation, for each tested number of `nops`. We only provide a plot for this configuration as it shows a successful configuration, i.e., $MCC = 1.0$ for all tested microarchitectures besides the T-Head XuanTie C906 (RISC-V) and Arm Cortex-A72. For these 2 microarchitectures, we cannot find a configuration that shows correlation. In general, a configuration using a memory fence and 0 `nops` works on the majority of the microarchitectures. 4 microarchitectures require a special configuration using a memory fence plus 8 `nops`. The memory fence is only important on the T-Head XuanTie C910 (RISC-V). All other microarchitectures show similar behavior independent of the memory fence. After a microarchitecture-specific number of `nops`, the MCC drops to 0 (no correlation). We find that this is related to eviction of the L1i of the microarchitecture and therefore connected to its size. After at most 32 768 `nops`, all microarchitectures do not show correlation anymore.

```
; initially [rwx_page] = mov reg1, 1; ret
store mov_reg1_0, [rwx_page]
memory barrier ; optional
N * nop ; N=1<<k, 0<=k<=15
jump rwx_page
; observe architectural result in reg1
```

```
store mov_reg1_0, [overwrite_here]
memory barrier ; optional
N * nop ; N=1<<k, 0<=k<=15
overwrite_here:
mov reg1, 1
; observe architectural result in reg1
```

Figure 5: Sequences used to test architectural I-cache leakage ($\mathcal{B}1$). An out-of-line (top) or an inline (bottom) target instruction encodes a move '1' (i.e., stale). A store instruction is used to overwrite this target instruction with a move '0'. An optional memory barrier and N (multiples of 2) `nops` follow. A jump follows for the out-of-line target instruction (top), while the inline target instruction follows for the bottom. Finally, register 1 holds the information if the stale ('1') or updated ('0') instruction was executed.

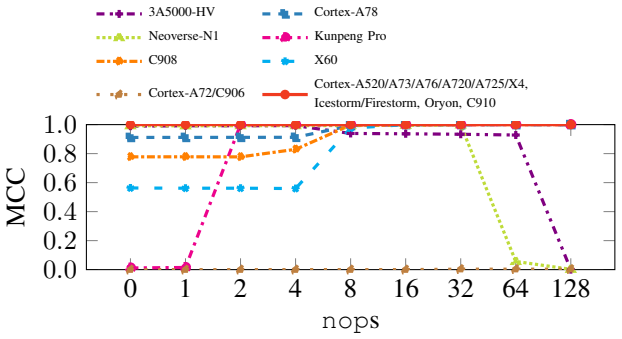

Figure 6: Results for the architectural I-cache-state leakage experiments ($\mathcal{B}1$) using a memory fence, a variable number of `nops`, and a jump instruction. We compute the MCC, which is 1 for perfect correlation and 0 for no correlation. All tested microarchitectures besides Arm Cortex-A72 and T-Head XuanTie C906 (RISC-V) show configurations with perfect correlation. For 4 microarchitectures, a configuration with 8 `nops` is needed; for the others, 0 `nops` works well. The drop after N `nops` is related to eviction of the L1i and therefore connected to its size.

We conclude that we can accurately leak the state of I-cache lines on 16 microarchitectures across the 3 major RISC architectures. Arm Cortex-A72 and T-Head XuanTie C906 are the only microarchitectures that do not show architectural I-cache leakage. Table 2 provides an overview of these results, where ◐ indicates that $\mathcal{B}1$ is available.

4.3. Prevalence of $\mathcal{B}2$

The second building block ($\mathcal{B}2$) of I^{2}SC is CTG, the cache-state transfer gadget, i.e., the ability to transiently bring cache lines into the I-cache, depending on the D-cache state of a victim cache line. As a result, we can use $\mathcal{B}1$ to architecturally leak this "transferred" state to infer the state of the victim D-cache line. $\mathcal{B}2$ works on 13 out of the 14 tested out-of-order or 18 tested microarchitectures. Table 2 shows these results, where ◑ indicates that victim-dependent transient instruction fetches work on a microarchitecture.

To test this building block, we manually craft a transient transfer gadget for each tested architecture based on the generic one shown in Figure 3. Since all microarchitectures are similar in their design, i.e., they all feature the same basic instructions used in Figure 3, the resulting gadgets are also generic. All gadgets are based on RSB-based misspeculation, as this form of transient execution is both reliable and triggerable on many microarchitectures [35].

In this experiment we test if (a) the microarchitecture allows for transiently bringing cache lines into the caches, (b) allows for transient instruction fetches, i.e., transiently bringing instructions into the I-cache, and (c) allows this to be dependent on a victim cache line (=$\mathcal{B}2$). To test which of these properties a microarchitecture satisfies we again test multiple configurations. We use the generic RSB-based misspredict gadget (cf. Figure 3) to transiently execute one of 2 sequences: a load instruction or a jump instruction, preceded by a dependent victim cache line load instruction. After the transient execution of these sequences the cache state of the victim cache line is transferred to either D-cache for the load instruction or to I-cache for the jump instruction. To reason about the success of the primitives we need to uncover this state from the caches.

Since the tested microarchitectures do not provide an architectural interface to read the cache states, we rely on timing measurements to expose the cache state of the caches. Relying on a timer for this measurement is fine, as the execution of the primitive and leakage of the state can be repeated multiple times, and because these results only give a hint about if the above properties are satisfied. For the D-cache leakage, we rely on Flush+Reload, for the I-cache-state leakage, we measure the execution latency of the I-cache line after executing the transient execution primitive.

Table 3 shows the results for this experiment. For each microarchitecture, we show the relative time in-/decrease (%) compared to a D- or I-cache miss. 13 of the 14 tested out-of-order or of the 18 total tested microarchitectures show transient dependent fetch behavior, i.e., a latency difference between a cached dependent transient jump vs. an uncached dependent transient jump. This indicates that $\mathcal{B}2$ is available on these microarchitectures. As expected, we see no transient loads or instruction fetches for the in-order microarchitectures—C906, C908, X60 (RISC-V), and Arm Cortex-A520. The Arm Cortex-A73 is the only out-of-order microarchitecture that does not show any latency reduction for transient instruction fetches, indicating that $\mathcal{B}2$ is unavailable on this microarchitecture. However, transient cached or uncached dependent data load instructions show a clear latency difference, indicating that the RSB-based misspeculation works well. We attribute this behavior to microarchitecture specifics, as other ARM microarchitectures like the Arm Cortex-A76 show $\mathcal{B}2$. Another special case is the T-Head XuanTie C910 (RISC-V), which only shows a latency reduction for the transient instruction fetch gadget, but not the data load gadget.

TABLE 3: Results for the prevalence study of CTG ($\mathcal{B}2$). We test 2 different configurations: a transient victim-dependent data load and instruction fetch. Each row shows the relative latency position of the uncached and cached cases between a measured slow baseline (cache miss) and fast baseline (cache hit): left means miss-like, right means hit-like. $\mathcal{B}2$ is available on 13 of the 14 out-of-order microarchitectures, as they show a latency difference for instruction fetches when the victim is cached or uncached. The in-order microarchitectures do not show transient behavior as expected and are excluded. $\mathcal{B}2$ is not available on Arm Cortex-A73 as it only supports transient loads but not transient instruction fetches.

	Data Load		Instr. Fetch		$\mathcal{B}2$
Microarch.	**uncached**	**cached**	**uncached**	**cached**	
Cortex-A72	●	●	●	●	✓
Cortex-A73	●	●	●	●	
Cortex-A76	●	●	●	●	✓
Cortex-A78	●	●	●	●	✓
Cortex-A720	●	●	●	●	✓
Cortex-A725	●	●	●	●	✓
Cortex-X4	●	●	●	●	✓
Neoverse-N1	●	●	●	●	✓
Oryon	●	●	●	●	✓
Kunpeng Pro	●	●	●	●	✓
Icestorm	●	●	●	●	✓
Firestorm	●	●	●	●	✓
C910	●	●	●	●	✓
3A5000-HV	●	●	●	●	✓

4.4. Prevalence of I^{2}SC

Only microarchitectures that show both building blocks, i.e., the architectural I-cache-state leakage oracle ($\mathcal{B}1$) and the transient cache-state transfer gadget CTG ($\mathcal{B}2$), are affected by I^{2}SC. Table 2 summarizes which microarchitectures are affected by I^{2}SC, indicated by ●. We find that in total 12 of 18 tested microarchitectures are affected by I^{2}SC. This is based on 16 of the 18 microarchitectures that show $\mathcal{B}1$ (◐), i.e., architectural I-cache-state leakage, and 13 that show $\mathcal{B}2$ (◑), i.e., victim-dependent transient fetching of instructions to the I-cache. We identify at least one affected microarchitecture for each of the major RISC architectures—ARM, RISC-V, and LoongArch. Only one microarchitecture, the T-Head XuanTie C906 (RISC-V), shows none of the 2 building blocks.

TABLE 4: Performance of $\mathcal{B}1$ on the affected microarchitectures measured using an I-cache covert channel transferring 1 Mibit using the architectural I-cache-state leakage primitive. We compare $\mathcal{B}1$ to timer-based execution latency measurements. 4 microarchitectures (†) use a memory barrier plus 8 `nops` while the other microarchitectures use 0 `nops`. $\mathcal{B}1$ performs well, with low error rates and high throughput, and outperforms timer-based measurements in most cases.

	Error Rate (%, ↓)			**Throughput** (kbit/s, ↑)			**Resolution** (µs, ↓)		
Microarch.	**Timer**	$\mathcal{B}1$	Δ	**Timer**	$\mathcal{B}1$	Δ%	**Timer**	$\mathcal{B}1$	Δ%
Cortex-A520	2.83	0.00	-2.8	615.21	777.95	+26.5	1.63	1.29	-20.9
Cortex-A73	0.01	0.18	+0.2	956.99	1410.35	+47.4	1.04	0.71	-31.7
Cortex-A76	15.79	0.03	-15.8	889.83	1036.97	+16.5	1.12	0.96	-14.3
Cortex-A78†	1.32	0.00	-1.3	1360.69	1712.61	+25.9	0.73	0.58	-20.5
Cortex-A720	2.67	0.00	-2.7	922.13	1212.07	+31.4	1.08	0.83	-23.1
Cortex-A725	0.67	0.00	-0.7	481.46	539.06	+12.0	2.08	1.86	-10.6
Cortex-X4	12.89	0.04	-12.9	1343.73	1547.63	+15.2	0.74	0.65	-12.2
Neoverse-N1	1.66	0.15	-1.5	756.59	842.06	+11.3	1.32	1.19	-9.8
Oryon	17.76	0.00	-17.8	925.73	1905.49	+105.8	1.08	0.52	-51.9
Kunpeng Pro†	0.01	0.03	+0.0	618.42	684.64	+10.7	1.62	1.46	-9.9
Icestorm	0.02	0.00	-0.0	1100.62	1277.07	+16.0	0.91	0.78	-14.3
Firestorm	0.05	0.00	-0.1	1372.07	1534.83	+11.9	0.73	0.65	-11.0
C908†	0.61	0.20	-0.4	913.77	824.89	-9.7	1.09	1.21	+11.0
C910	30.69	0.07	-30.6	542.79	538.74	-0.7	1.84	1.86	+1.1
X60†	1.13	1.26	+0.1	213.68	211.55	-1.0	4.68	4.73	+1.1
3A5000-HV	0.01	0.36	+0.3	2846.13	2961.38	+4.0	0.35	0.34	-2.9

† Memory barrier + 8 `nops`.

5. Evaluation

In this section, we evaluate the performance of $\mathcal{B}1$, $\mathcal{B}2$, and I^{2}SC using a covert channel.

5.1. Performance of $\mathcal{B}1$

To evaluate the performance of $\mathcal{B}1$, we build an I-cache covert channel using the best primitive of our architectural I-cache-state leakage gadget for each microarchitecture (cf. Figure 6). We use a jump to the RWX memory page and insert a memory barrier between the store and jump instructions. We use the suitable number of `nops` for each microarchitecture (cf. Figure 6). We prime (message bit '1') or do not prime (message bit '0') the I-cache with the RWX page. We transfer 1 Mibit over the covert channel.

Table 4 shows the results for this experiment. We see low error rates ($< 1.5\,\%$) on all microarchitectures. The throughput ranges from 211.55 kbit/s on the SpacemiT X60 (RISC-V) to 2961.38 kbit/s on Loongson's 3A5000-HV microarchitecture (LoongArch). Respectively, the resolution of $\mathcal{B}1$ ranges from 4.73 µs to 0.34 µs. We also compare $\mathcal{B}1$ to a timer-based primitive similar to the one used in Section 4.3, i.e., we measure execution latency and compare it to a threshold. $\mathcal{B}1$ outperforms the timer-based measurements on most microarchitectures, with the most significant improvement on the T-Head XuanTie C910, where the error rate drops from 30.96 % to 0.07 %. Thus, $\mathcal{B}1$ leaks the I-cache state both accurately and efficiently and outperforms timing-based measurements.

TABLE 5: Performance of $\mathcal{B}2$ on the affected microarchitectures measured using an I-cache covert channel transferring 1 Mibit via the D-cache using the CTG to transfer the state. We compare I-cache-state leakage via $\mathcal{B}2$ to timer-based execution latency measurements. $\mathcal{B}2$ performs well, with low error rates and high throughput. The architectural leakage via $\mathcal{B}1$ outperforms the timer-based measurements.

	Error Rate (%, ↓)			**Throughput** (kbit/s, ↑)			**Resolution** (µs, ↓)		
Microarch.	**Timer**	$\mathcal{B}1$	Δ	**Timer**	$\mathcal{B}1$	Δ%	**Timer**	$\mathcal{B}1$	Δ%
Cortex-A72†	7.13	-	-	232.41	-	-	4.30	-	-
Cortex-A76	16.66	0.58	-16.08	459.96	492.79	+7.14	2.17	2.03	-6.45
Cortex-A78	0.91	0.94	+0.03	617.83	678.59	+9.83	1.62	1.47	-9.26
Cortex-A720	0.88	0.53	-0.35	394.68	448.47	+13.63	2.53	2.23	-11.86
Cortex-A725	6.88	0.04	-6.84	239.71	188.65	-21.30	4.17	5.30	+27.10
Cortex-X4	1.35	0.09	-1.26	572.73	608.73	+6.29	1.75	1.64	-6.29
Neoverse-N1	12.19	8.17	-4.02	321.56	334.86	+4.14	3.11	2.99	-3.86
Oryon	19.30	0.51	-18.79	539.98	741.84	+37.38	1.85	1.35	-27.03
Kunpeng Pro	2.79	3.28	+0.49	305.98	321.76	+5.16	3.27	3.11	-4.89
Icestorm	0.43	0.43	+0.00	467.51	487.75	+4.33	2.14	2.05	-4.21
Firestorm	0.61	0.61	+0.00	605.41	624.82	+3.21	1.65	1.60	-3.03
C910	31.99	0.70	-31.29	57.56	57.35	-0.36	17.37	17.44	+0.40
3A5000-HV	10.74	3.66	-7.08	425.07	398.69	-6.21	2.35	2.51	+6.81

† $\mathcal{B}1$ does not work on Cortex-A72.

5.2. Performance of $\mathcal{B}2$

To evaluate the performance of $\mathcal{B}2$, we construct an I-cache-based covert channel, but use D-cache lines to encode the message. We use the transient cache-state transfer gadget (CTG) to bring the instructions of an RWX target page into the I-cache, depending on the cache state of a victim D-cache line. We rely on $\mathcal{B}1$ to architecturally recover this I-cache state. We compare $\mathcal{B}1$ to execution time latency timing measurements, and transfer 1 Mibit over the covert channel.

Table 5 shows the results for this experiment. $\mathcal{B}2$ generally performs well when using $\mathcal{B}1$, with bit error rates of $< 1.5\,\%$. However, on the Loongson 3A5000-HV, Kunpeng Pro, and Neoverse-N1, we observe higher bit error rates ranging from 3.66 % (Loongson 3A5000-HV) to 8.17 % (Neoverse-N1). Timer-based measurements perform worse with bit error rates up to 31.99 % on the T-Head XuanTie C910. These results can be explained by the timer resolution on these devices (Table 9 in Appendix A). Throughput ranges from 57.35 kbit/s on the T-Head XuanTie C910 (RISC-V) to 741.84 kbit/s on Qualcomm's Oryon microarchitecture (ARM). Respectively, the resolution ranges from 17.44 µs to 1.35 µs. Again, $\mathcal{B}2$ significantly outperforms the timer-based measurements. We conclude that $\mathcal{B}2$ accurately and efficiently transfers D-cache state to the I-cache, which can be accurately and efficiently leaked via $\mathcal{B}1$. Timer-based measurements perform worse with high bit error rates and on-par throughput.

5.3. Performance of I^{2}SC

Finally, we plug $\mathcal{B}1$ and $\mathcal{B}2$ together to form and evaluate I^{2}SC. To evaluate I^{2}SC, we again use a covert channel where a D-cache line is used to encode the signal. The receiver transfers the D-cache state to the I-cache via $\mathcal{B}2$ and then

TABLE 6: Performance of a D-cache-based covert channel transferring random bits via I^{2}SC. We transfer 1 Mibit. I^{2}SC performs well, with bit error rates $<1.5\,\%$ and high throughput on most microarchitectures. The low bit error rate leads to a high channel capacity.

Microarch.	Capacity (kbit/s, ↑)	Error Rate (%, ↓)	Throughput (kbit/s, ↑)	Resolution (µs, ↓)
Cortex-A76	542.02	0.46	566.11	1.77
Cortex-A78	565.40	4.82	783.87	1.28
Cortex-A720	544.64	0.01	545.39	1.83
Cortex-A725	207.79	0.05	209.07	4.78
Cortex-X4	824.14	0.05	829.67	1.21
Neoverse-N1	453.30	0.37	469.93	2.13
Oryon	702.85	1.04	766.85	1.30
Kunpeng Pro	268.82	5.58	390.00	2.56
Icestorm	543.29	0.45	567.07	1.76
Firestorm	691.00	0.49	723.33	1.38
C910	53.95	0.98	58.59	17.07
3A5000-HV	366.05	1.23	404.90	2.47

TABLE 7: I^{2}SC compared to Flush+Reload for a D-cache-based covert channel transferring 1 Mibit. I^{2}SC significantly outperforms Flush+Reload in capacity with bit error rates $<1.5\,\%$ compared to $17.52\,\%$ to $32.08\,\%$ for Flush+Reload.

	Capacity (kbit/s, ↑)			Error Rate (%, ↓)			Throughput (kbit/s, ↑)		
Microarch.	F+R	I^{2}SC	Δ%	F+R	I^{2}SC	Δ	F+R	I^{2}SC	Δ%
Oryon	270.56	702.85	+159.8	17.52	1.04	-16.5	818.65	766.85	-6.3
C910	46.55	53.95	+15.9	32.08	0.98	-31.1	491.16	58.59	-88.1
3A5000-HV	213.22	366.05	+71.7	29.97	1.23	-28.7	1790.52	404.90	-77.4

architecturally leaks that state via $\mathcal{B}1$. We again transfer 1 Mibit over the covert channel.

Table 6 shows the final performance results for I^{2}SC. Throughput ranges from 58.59 kbit on the T-Head XuanTie C910 (RISC-V) to 829.67 kbit on the Arm Cortex-X4. The error rate is generally low ($< 1.5\,\%$). Arm Cortex-A720 shows the lowest bit error rate of $0.01\,\%$. Kunpeng Pro has the highest bit error rate of $5.58\,\%$. The resulting capacity ranges from 53.95 kbit (T-Head XuanTie C910) to 824.14 kbit (Arm Cortex-X4). The resolution of I^{2}SC ranges from 17.07 µs (T-Head XuanTie C910) to 1.21 µs (Arm Cortex-X4). We conclude that I^{2}SC efficiently and accurately leaks the D-cache state architecturally.

5.4. Comparison to Flush+Reload

To compare the performance of I^{2}SC to timer-based cache side channels, we build a Flush+Reload-based covert channel. We automatically infer the Flush+Reload threshold using reference measurements for a cached and uncached cache line. We transfer 1 Mibit over the covert channel. For timer-based measurements, we use `clock_gettime` on ARM, as access to `CNTVCT_EL0` is restricted, and `rdtime` on RISC-V and LoongArch (cf. Table 10 in Appendix B). For both channels, we use the same minimal binary encoding: one bit is transmitted via the cache state of a single cache line, without amplification, framing, or error-correction. In line with recent work comparing side channels [47], we derive the Shannon capacity assuming a binary symmetric channel. The reported capacity thus reflects the theoretical upper bound given the measured throughput and bit error rate, independent of a specific transmission protocol.

For the remainder of the paper, we select one reference microarchitecture per ISA: Qualcomm Oryon (ARM), T-Head XuanTie C910 (RISC-V), and Loongson 3A5000-HV (LoongArch). Table 7 shows the results for this experiment in comparison to the data from the previous section (cf. Section 5.3). I^{2}SC outperforms Flush+Reload, as it has a much lower bit error rate, resulting in a higher true capacity, even if the raw throughput is higher for Flush+Reload. For example, on Qualcomm Oryon (ARM), we see a $159.8\,\%$ increase in capacity, although the throughput is $6.3\,\%$ lower compared to Flush+Reload, as the error rate drops from $17.52\,\%$ to $1.04\,\%$. The high bit error rate of the Flush+Reload-based covert channel can again be explained by the low timer resolution on the tested machines (cf. Section 5.2 and Table 9 in Appendix A). We conclude that our timer-free primitive I^{2}SC outperforms existing cache side-channel attacks.

6. Case Studies

In this section, we use I^{2}SC to mount a Spectral attack, to attack AES T-tables, and to recover touch-event timing on Android.

6.1. Architectural Spectre: Spectral

In this section, we demonstrate a I^{2}SC-based variant of the Spectral attack [18] that works on ARM, RISC-V, and LoongArch. Spectral is a Spectre variant where the side channel does not rely on timing but uses architectural leakage. For this case study, we use a classical Spectre-PHT attack [36] with I^{2}SC instead of timing. While other variants of Spectre can also be used, Spectre-PHT works on a wide range of devices, reducing the engineering effort when building a cross-ISA PoC.

Threat Model. Following prior work [18], [27], [36], [48], [49], an attacker executes unprivileged code. We assume no access to architectural timing sources, i.e., no access to high-resolution timers such as `PMCCNTR` [2] or counting threads [15], as well as coarse timers with microsecond-scale resolution or worse [2], [50], [51]. As is standard for Spectre, the attacker and the victim share memory. For controlled experiments, we use an injected leakage gadget. Unlike earlier Spectral work on x86 [18], our approach works with conventional byte-wise gadgets [36] rather than requiring bit-wise encoders.

Attack Methodology. I^{2}SC can replace timing-based channels, enabling reuse of otherwise *unchanged* Spectre exploits targeting ARMv8 [13]. Additionally, it supports RISC-V and LoongArch. In contrast to approaches built around instructions such as `umwait` [18], we do not depend on transient stores or special-purpose gadgets. Moreover, the

coordination is straightforward, as the attacker triggers and observes the gadget within the same flow.

Setup. Our PoC mirrors Spectre-PHT PoCs by leaking out-of-bounds array bytes [13], [35]. We apply in-place mistraining [35], issuing 10 in-bounds accesses per out-of-bounds access. The leakage gadget follows the original Spectre design [36], distributing the secret across 256 pages. To prepare the cache state, we use ISA-appropriate maintenance (cf. Table 10) and then read back the state with I^{2}SC.

Results. Our evaluation runs on the reference microarchitectures: Qualcomm Oryon (ARM), running Windows 11 (WSL), T-Head XuanTie C910 (RISC-V), running Debian 12, and Loongson 3A5000-HV (LoongArch), running Loongnix 20. On all ISAs, we achieve leakage rates that outperform previous timing-based Spectre-PHT attacks. We rely on the majority vote for the leaked characters to ensure we always leak the correct secret. In total, we leak 6 kB. On Qualcomm Oryon (ARM), we measure a leakage of 2065 B/s (±6.15). The T-Head XuanTie C910 (RISC-V) is the slowest core, and we measure only 17 B/s (±0.46), with zero observed errors. On the Loongson 3A5000-HV (LoongArch), there is no unprivileged cache flush instruction. Thus, we must rely on cache eviction, and only measure a leakage of 287 B/s (±2.36), but still with zero observed errors. On ARM, the I^{2}SC-based variant exceeds the best prior timing-based Spectre-PHT result [13], and is on-par with Spectre results on x86 (cf. Schwarzl et al., Table I [51]).

TABLE 8: Results for the AES T-Tables attack evaluated on the reference microarchitectures. The attack generally works well, with success rates >90 %. I^{2}SC outperforms Flush+Reload in success rate while falling short in attack time.

	Success Rate (%, ↑)			Time (ms, ↓)			
Microarch.	**F+R**	**I^{2}SC**	**Δ**	**F+R** (±, n=100)		**I^{2}SC** (±, n=100)	**Δ%**
Oryon	1	100	+99	1417	(1.11)	1447	(0.61) +2.11
C910	47	94	+47	2847	(23.53)	22018	(122.41) +673.4
3A5000-HV	2	100	+89	58	(0.02)	193	(0.03) +232.8

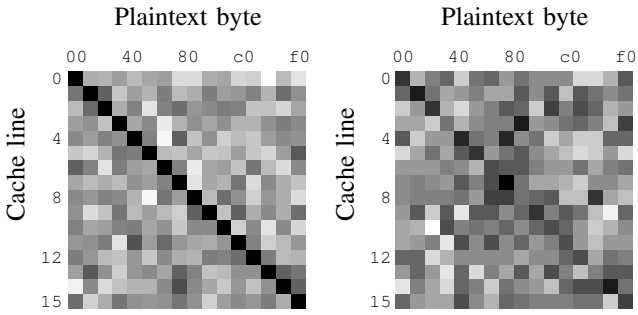

Figure 7: AES T-table cache-access pattern on T-Head XuanTie C910 with I^{2}SC (left) and Flush+Reload (right). The patterns for the other reference microarchitectures can be found in Figure 10 in Appendix C.

6.2. AES T-Tables

To demonstrate how noise-free I^{2}SC is, we mount an AES T-table attack, a "benchmark" commonly used for cache attacks [28], [52]–[54]. Our attack targets the T-table implementation of OpenSSL 3.5.2 (August 2025).

Setup. We mount the attacks on the 3 reference microarchitectures: Qualcomm Oryon (ARM), running Windows 11 (WSL), T-Head XuanTie C910 (RISC-V), running Debian 12, and Loongson 3A5000-HV (LoongArch), running Loongnix 20. On all microarchitectures, we apply I^{2}SC without relying on timers of any kind.

The attack follows the structure of a Flush+Reload attack on T-tables but replaces the measurement primitive with I^{2}SC. $\mathcal{B}2$ transiently copies the state of the table's cache lines into the I-cache using an attacker-controlled RWX memory region and $\mathcal{B}1$ leaks this I-cache state architecturally. Our implementation is inspired by the Flush+Reload baseline from Gruss et al. [28], but adapted to use I^{2}SC.

Evaluation. We repeat the AES T-table attack 100 times. I^{2}SC yields a clear binary distinction between cache hits and misses, avoiding the fluctuations inherent to timer-based approaches. We recover the AES key without errors in 100 % of runs on the Loongson 3A5000-HV (LoongArch), and Qualcomm Oryon (ARM), and in 94 % of runs on the T-Head XuanTie C910 (RISC-V). The average key extraction times are 1447 ms, 193 ms and 22 018 ms for Oryon, 3A5000-HV, and C910, respectively.

For comparison, we mount a Flush+Reload variant on the reference microarchitectures. On the C910, Linux kernel 6.8 restricts access to the high-resolution timer `rdcycle` by default via `/proc/sys/kernel/perf_user_access`. Thus, we must rely on the coarser-grained `rdtime` instruction that only updates every 333 ns (cf. Table 9 in Appendix A). The Flush+Reload-based attack only recovers the key in 47 % of runs and completes in 2847 ms. Figure 7 shows the histogram of the recovered cache access patterns, clearly showing the diagonal expected for a first-key-byte value of '0', with much less noise for the I^{2}SC-based variant (left). The patterns for the other reference microarchitectures can be found in Figure 10 in Appendix C. On Qualcomm Oryon, Flush+Reload only succeeds in 1 % of runs and is only slightly faster (1417 ms). This is again due to the coarse timer on the SoC (Snapdragon X). On the Loongson 3A5000-HV, we have to rely on eviction as the flush instructions are privileged (cf. Table 10). The Flush+Reload-based attack takes 58 ms and recovers the key in only 2 % of the runs. This case study shows that I^{2}SC outperforms traditional timing-based attacks, especially on CPUs without unprivileged high-resolution timers.

We further stress-test the setup with one and two CPU cores fully utilized using the `stress` utility. The I^{2}SC-variant remains unaffected, confirming that I^{2}SC maintains accuracy under realistic system noise. Thus, this case study demonstrates that I^{2}SC is suitable for practical cryptographic key extraction attacks on commodity hardware without depending on any timing primitive.

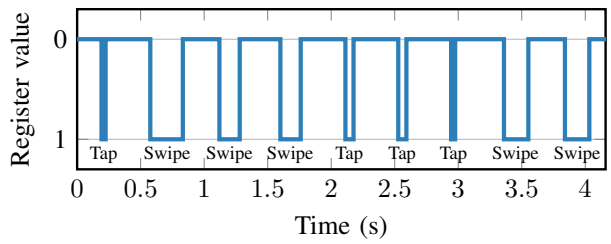

Figure 8: Register value after leaking the cache state of victim cache line `0x1639d0` of `libinput.so` using I^{2}SC. The leakage correlates with user input, revealing taps (short peaks) and swipes (longer peaks).

6.3. Shared Library Touch-Event Timing Attack

As a final case study, we revisit a classical attack against shared Android libraries [2]. It exploits that system libraries such as `libinput.so`, which handles touchscreen events, are mapped into all apps. By monitoring cache activity on specific functions, an attacker can infer fine-grained touch-event timing, including taps and swipes. We replace the original Flush+Reload attack by I^{2}SC as a timer-free replacement on a recent Android device, a Google Pixel 9.

Threat Model. We assume a malicious unprivileged Android app co-located with a victim app on the same device. Both apps share `libinput.so` through the system loader. The attacker aims to infer touchscreen interactions without privileged access, relying only on unprivileged code execution and a shared library memory mapping.

Setup. We target the Cortex-A720 cores on a Google Pixel 9 smartphone running the latest version of Android 16 with 12GB of RAM. We identify suitable cache lines within `libinput.so` by scanning for instruction lines that correlate with touchscreen activity. Following the methodology of ARMageddon, we locate a hot cache line inside the function `InputConsumer::resampleTouchState`. To simulate taps and swipes reproducibly, we use Android's `input` tool, but we also verify the signal with real user interaction.

Attack Methodology. We map `libinput.so` into the attacker's process space and apply I^{2}SC on the identified cache line. Specifically, the attacker queries the line's cache state using the I-cache oracle, yielding a binary signal corresponding to whether the victim touched the screen. Unlike timing-based Flush+Reload, I^{2}SC is noise-free and architectural, avoiding timer-precision issues on ARM.

We additionally verify that we exploit the same event as in the case study by Lipp et al. [2]. For this, we reverse engineer that the location in `libinput.so` on the Alcatel One Touch Pop 2 firmware image (version `5.0.2`, 18.08.2025) lies in the `InputConsumer::resampleTouchState` C++ function. Our identified offset `0x1639d0` on the Pixel 9 (Android 16) maps to the same function.

Results. Figure 8 shows the leakage trace from our attack on the Pixel 9. We recover taps (short peaks) and swipes (longer peaks) with no observed false positives. Our attack achieves a temporal resolution of 9 µs, allowing us to recover fine-grained touch-event timing. The architectural signal remains stable even under moderate system load and across repeated trials. This leakage reveals high-fidelity event metadata (timing, count, rhythm, and tap-vs-swipe segmentation), enabling behavior inference. As human interaction differences are in the millisecond range [55], [56], our microsecond-level signal provides ample granularity for downstream inference pipelines. Prior work has shown that such fine-grained timing traces can be post-processed to infer typed text and aid password recovery [56], [57]. Compared to the original ARMageddon implementation, which required fine-grained timers, our approach requires no timing source at all.

7. Mitigations

7.1. Software Mitigations

A direct mitigation direction is to prevent unprivileged code from mapping memory WX, as this eliminates $\mathcal{B}1$. However, on Linux this is not an immediately deployable drop-in mitigation, as dynamic code generation in JIT compilers, runtime patching, and some debugging tools require writable and executable memory. Still, this is a viable design-level direction: major applications such as Google Chrome and Mozilla Thunderbird also support platforms such as macOS and OpenBSD, where stricter W⊕X enforcement is already common [58], [59].

Implementation. For Linux, we prototype a kernel module that hooks the syscalls `mmap`, `mprotect`, and `shmat` using kprobes. This kernel module terminates processes that request W⊕X mappings, with optional allowlisting by PID for trusted processes. This enables a selective W⊕X enforcement, retaining compatibility for known-safe workloads while preventing its abuse by attackers.

Evaluation. We evaluate our kernel module with respect to security, performance, and compatibility.

Security. We verify that our kernel module successfully prevents all our attacks. As we require memory that is WX, this mitigation is effective on all CPUs.

Performance. To assess the performance overhead of the kernel module, we run the SPEC CPU 2017 benchmark with and without the kernel module. We do not see any performance overhead, and there was no `mmap` call with writable and executable mapping during the benchmark.

Compatibility. For evaluating the compatibility with existing applications, we slightly modify the kernel module to only log applications violating its rules. We use the kernel module on one of the author's work laptops for one week to acquire real-world data. Surprisingly, we see that many applications still use mappings that are WX, including KDE Plasma, Mozilla Thunderbird, and Google Chrome. Thus, these applications must be added to the allowlist.

Limitation. As our kernel module is only a proof-of-concept implementation, it does not consider sophisticated variants of making executable memory writable. For example, an attacker could create two virtual mappings to the

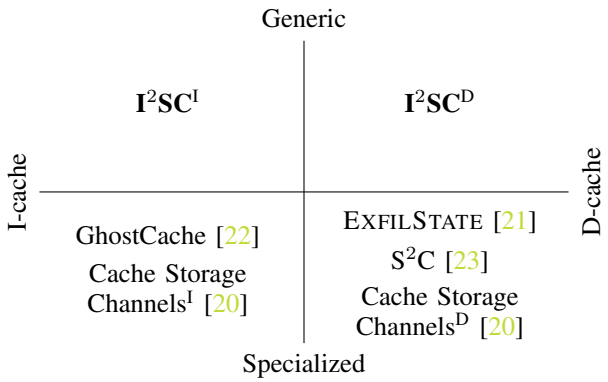

Figure 9: Comparison of I²SC to other architectural side channels on RISC CPUs. The labels "I" and "D" refer to the I-cache and D-cache primitives introduced in the work. GhostCache and Cache Storage Channels provide an I-cache primitive similar to $\mathcal{B}1$, but use specialized sequences. ExfilState, S²C and Cache Storage Channels provide specialized primitives for D-cache leakage, using microarchitectural-specific behavior, special features, or privileged interfaces. I²SC provides a generic I- and D-cache leakage primitive.

same physical page, where one is writable and the other is executable and read-only. Detecting such setups would require more complex logic, with higher overheads.

7.2. Hardware Mitigations

Fundamentally preventing I²SC requires eliminating the instruction/data cache incoherence. Vendors could achieve this by modifying their hardware implementation. For mitigation, it is sufficient to stop any of the properties exploited by I²SC. For example, vendors could ensure that stores to executable memory are immediately visible to the instruction fetch unit, either by invalidating corresponding I-cache lines or synchronously updating them. Alternatively, vendors could disable out-of-order instruction fetches that can transiently populate I-cache structures. While such changes close the side channel at its root, they can incur measurable performance penalties in self-modifying workloads (e.g., JITs) and may not be retrofittable to existing CPUs.

8. Related Work

Cache-based Side Channels. Early cache attacks measure access latency to distinguish hits from misses, enabling key extraction and fine-grained profiling via Flush+Reload [6], Prime+Probe [32], and variants [2], [5], [28] on many architectures. As a response, several platforms reduced timer precision or restricted user-level access to cycle counters, especially on ARM [2], [13], which complicates portable timer-based attacks and motivates timer-free alternatives. I²SC can be used as a drop-in replacement for cache attacks on affected CPUs, without requiring *any* timing primitive.

Timer-free (Architectural) Cache Side Channels. Multiple works have shown timer-free cache state leakage. Figure 9 gives an overview of the approaches on RISC CPUs and categorizes them into I-cache state (left) and D-cache (right) state leakage. They vary highly in their generality.

I-cache. Prior work on timer-free I-cache state leakage exploits I-cache incoherence as an oracle on the state of I-caches [20], [22]. $\mathcal{B}1$ of I²SC generalizes this leakage using an automated test harness, making the primitive applicable to the three major RISC ISAs and 16 of 18 tested microarchitectures, stripping the need for microarchitecture-specific adaptions. I²SC further bridges this leakage to the D-caches.

D-cache. Timer-free D-cache state leakage has been demonstrated from unprivileged contexts [17], [18], [21], [23] and from privileged contexts to attack trusted execution environments [19], [20], [33], [34]. The existing unprivileged architectural side channels require specific ISA extensions [17]–[19], specific ISA features [21], [23], or exploit microarchitecture-specific race conditions [21]. In contrast, I²SC exploits design decisions in many RISC CPUs, making it generic and highly applicable.

Exploiting Self-modifying Code. Self-modifying code (SMC) has also been exploited by previous work on x86. Ragab et al. [60] demonstrated that self-modifying code on x86 can be used to induce transient execution, leading to the execution of stale data and injection of transient floating-point values. Similarly, Son et al. [61] showed that on x86 CPUs, self-modifying code can be used to refine cache attacks. In addition to attacks, Aldaya et al. [62] demonstrated that self-modifying code can be used to slow down applications, extending the window for side-channel attacks. In contrast, I²SC focuses on RISC-style CPUs and exploits the incoherence between data and instruction cache.

Architectural CPU Vulnerabilities. A growing body of work shows that ISA-microarchitecture mismatches can leak state through architectural channels. ÆPICLeak [63] exploited uninitialized microarchitectural buffers in x86 APIC registers to read stale data architecturally. GhostWrite [64] leveraged undocumented vector store instructions on RISC-V to bypass memory protection. CacheWarp [65] demonstrated privilege escalation by architecturally replaying stale cache lines after targeted cache downgrades. ZenBleed [66] abused mis-speculated register renaming on AMD Zen to architecturally exfiltrate register contents without timing. These attacks share with I²SC that they do not require timing, but they typically rely on rare microarchitectural bugs fixed in newer CPUs. In contrast, I²SC exploits intentional and widespread design choices in RISC CPUs, making it portable across vendors and generations.

9. Discussion

In this section, we discuss implications for defenders, requirements and limitations, and generality beyond caches.

Implications for Defenders. Timer throttling alone is insufficient once attackers can switch to architectural side channels, as with I²SC. Software defenses should therefore combine W⊕X (with careful handling of dual mappings),

speculation controls, and, where feasible, code hardening that avoids attacker-controlled self-modifying sequences. From the hardware side, vendors can (i) make stores to executable memory synchronously visible to the IFU, (ii) prohibit speculative store-to-fetch forwarding, or (iii) constrain out-of-order front ends that populate IFU-visible structures from data-side events in attacker code. Each of these mitigations carries performance costs for legitimate SMC-heavy workloads. However, they are necessary, as our results suggest that partial measures (e.g., only coarsening timers) leave a usable and robust attack surface.

Requirements and Limitations. I^{2}SC relies on WX mappings. In hardened environments with strict W$\oplus$X, this may not be possible, making exploitation of I^{2}SC impossible. However, Linux does not enforce this, and there is also no possibility in the upstream kernel to enable such a policy.

$\mathcal{B}2$ requires transient execution and is, therefore, sensitive to hardware fixes that remove the source of transient execution, e.g., speculation, entirely. In contrast, Spectre defenses such as retpoline or speculative load hardening (SLH) do not stop $\mathcal{B}2$, as they are inherently designed to protect victims rather than to remove attacker capabilities. Additionally, while architectural feedback avoids cycle-level noise in timers, it can still be affected by system load. For example, a high system load might thrash the cache and thus remove the signal, or affect the speculation in $\mathcal{B}2$. However, this limitation is not specific to I^{2}SC, but also affects traditional cache attacks.

Generality Beyond Caches. While we instantiate I^{2}SC for cache state, the technique highlights a broader pattern of exploiting race conditions when synchronizing microarchitectural elements. Whether similar constructions exist for other structures (e.g., predictor tables or TLBs) depends on undocumented paths and arbitration policies. We encourage future work to investigate similar channels.

10. Conclusion

We presented I^{2}SC, a new architectural cache side channel that leverages instruction/data incoherence on modern RISC CPUs. Unlike timing-based approaches, I^{2}SC provides a stable, timer-free oracle for both I- and D-cache state, making classical cache attacks viable even when high-resolution timers are unavailable. Our survey of 18 microarchitectures shows that 12 are affected, including recent high-performance cores deployed in smartphones, servers, and domestic ISAs. We demonstrated practical attacks, such as Spectre variants, AES T-table key recovery, and touch-event timing recovery, highlighting that I^{2}SC is not only theoretically possible but also exploitable in practice. Overall, our findings show that architectural side channels bypass timer-based defenses and remain a pressing concern for RISC platforms, motivating immediate software hardening and longer-term hardware changes.

Ethics Considerations

Stakeholders and Impacts. Our work affects end users, hardware/SoC vendors, OS developers, cloud/mobile operators, the security research community, and the public. End users could face confidentiality risks if attackers adopt I^{2}SC to extract secrets in multi-tenant or mobile settings. Vendors and OS developers may incur engineering costs to assess and mitigate the issue, but benefit from concrete guidance and a reusable detection harness. Operators may face short-term operational costs (patching, performance trade-offs) with long-term gains from defenses beyond timer throttling. Researchers and educators gain tools to study architectural channels, with a dual-use risk if PoCs are misused. Society benefits from improved platform security and more accurate threat models.

Principles. Guided by the Menlo Report (*Beneficence*, *Respect for Persons*, *Justice*, *Respect for Law and Public Interest*) and recent ethical guidance for security research, we structured experiments to maximize public benefit while minimizing risk. No human subjects or third-party data were involved. All experiments ran on devices, keys, and software under our control, with permission and in accordance with applicable laws and institutional policies.

Potential Harms. The primary harms are the possible acceleration of side-channel or Spectre-style attacks and short-term disruptions during mitigation deployment, including performance regressions. We identify no rights-based harms: we did not collect personal data, interact with unsuspecting parties, or access unauthorized systems.

Mitigations. We practiced coordinated disclosure with the three affected and responsible vendors—Arm, T-Head, and Loongson—before publication. Each vendor received and acknowledged receipt of all technical details to support triage and remediation. We release a detection-focused harness and minimally sufficient PoCs that operate only on self-owned data/keys and avoid exploitation. We additionally provide defensive guidance for software and hardware. We did not conduct internet-wide probing, cloud-tenant testing, or analysis of third-party workloads. All measurements were local and sandboxed.

Decision to Proceed and Publish. We weighed risks against benefits (Beneficence) and upheld individual rights (Respect for Persons). Given cross-ISA impact, the inadequacy of timer throttling alone, and the likelihood of independent discovery, we concluded that publication—paired with vendor coordination and non-weaponized artifacts—maximizes security benefits and serves the public interest.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Acknowledgment

We thank the reviewers and our shepherd for their valuable feedback and suggestions. We are grateful to Lorenz

Hetterich for insightful discussions at the early stages of this project. This work was supported in part by a Google Research Scholar Award. The views expressed are those of the authors and do not necessarily reflect those of the sponsors.

References

[1] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.

[2] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security*, 2016.

[3] Y. Lin, J. Wong, X. Li, H. Ma, and D. Gao, "Peep with a mirror: breaking the integrity of android app sandboxing via unprivileged cache side channel," in *USENIX Security Symposium*, 2024.

[4] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[6] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[8] Mozilla, "performance.now resolution," 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[9] Chrome for Developers, "Disable JavaScript," 2019. [Online]. Available: https://developer.chrome.com/docs/devtools/javascript/disable

[10] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS*, 2009.

[11] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *CCS*, 2012.

[12] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[13] L. Hetterich and M. Schwarz, "Branch Different - Spectre Attacks on Apple Silicon," in *DIMVA*, 2022.

[14] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[15] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[16] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, 2017.

[17] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security Symposium*, 2017.

[18] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.

[19] P. Qiu, Q. Gao, D. Wang, Y. Lyu, C. Wang, C. Liu, R. Sun, and G. Qu, "Pmu-leaker: Performance monitor unit-based realization of cache side-channel attacks," in *ASP-DAC*, 2023.

[20] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *S&P*, 2016.

[21] F. Thomas, M. Torres, D. Moghimi, and M. Schwarz, "ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs," in *CCS*, 2025.

[22] Y. Jin, M. Sun, D. Wang, P. Qiu, Y. Zhang, and S. Deng, "GhostCache: Timer-and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips," in *CCS*, 2025.

[23] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions," in *USENIX Security*, 2023.

[24] D. Katzman, W. Kosasih, C. Chuengsatiansup, E. Ronen, and Y. Yarom, "The gates of time: Improving cache attacks with transient execution," in *USENIX Security Symposium*, 2023.

[25] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, "Computing with time: Microarchitectural weird machines," in *ASPLOS*, 2021.

[26] P.-L. Wang, R. Paccagnella, R. S. Wahby, and F. Brown, "Bending microarchitectural weird machines towards practicality," in *USENIX Security*, 2024.

[27] G. Horowitz, E. Ronen, and Y. Yarom, "Spec-o-scope: Cache probing at cache speed," in *ACM CCS*, 2024.

[28] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[29] W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *Journal of Computer Security*, 1992.

[30] ARM, "Arm Architecture Reference Manual for A-profile architecture," 2023.

[31] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.

[32] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[33] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, 2015.

[34] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *USENIX Security Symposium*, 2017.

[35] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security*, 2019, extended classification tree and PoCs at https://transient.fail/.

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

[38] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[39] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *S&P*, 2019.

[40] E. Göktaş, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative Probing: Hacking Blind in the Spectre Era," in *CCS*, 2020.

[41] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," in *USENIX Security Symposium*, 2022.

[42] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[43] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv:1806.07480*, 2018.

[44] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs," in *S&P*, 2023.

[45] Intel, "Intel 64 and IA-32 Architectures Software Developer′s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," 2024.

[46] L. T. C. Limited, "Loongarch reference manual - volume 1: Basic architecture, version 1.10," 2023. [Online]. Available: https://loongson.github.io/LoongArch-Documentation/LoongArch-Vol1-EN.pdf

[47] F. Rauscher, C. Fiedler, A. Kogler, and D. Gruss, "A systematic evaluation of novel and existing cache side channels," in *NDSS*, 2025.

[48] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, "ShadowLoad: Injecting State into Hardware Prefetchers," in *ASPLOS*, 2025.

[49] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.

[50] Stephen Röttger and Artur Janc, "A Spectre proof-of-concept for a Spectre-proof web," 2021. [Online]. Available: https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[51] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz, "Robust and scalable process isolation against spectre in the cloud," in *ESORICS*, 2022.

[52] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[53] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[54] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and Generic Microarchitectural Hash-Function Recovery," in *S&P*, 2024.

[55] M. Schwarz, M. Lipp, D. Gruss, T. Prescher, C. Maurice, and S. Mangard, "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks," in *NDSS*, 2018.

[56] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.

[57] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.

[58] Apple, "Porting just-in-time compilers to Apple silicon," 2020. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/porting-just-in-time-compilers-to-apple-silicon

[59] OpenBSD Journal, "W^X now mandatory in OpenBSD," 2016. [Online]. Available: https://undeadly.org/cgi?action=article;sid=20160527203200

[60] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.

[61] S. Son, D. Moghimi, and B. Gulmezoglu, "Smack: Efficient instruction cache attacks via self-modifying code conflicts," in *ASPLOS*, 2025.

[62] A. C. Aldaya and B. B. Brumley, "HyperDegrade: From GHz to MHz Effective CPU Frequencies," in *USENIX Security Symposium*, 2022.

[63] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.

[64] F. Thomas, E. G. Arribas, L. Hetterich, D. Weber, L. Gerlach, R. Zhang, and M. Schwarz, "RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs," in *CCS*, 2025.

[65] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[66] T. Ormandy, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

Appendix

Appendix A. Timer Resolution

Table 9 shows the timer resolution on the tested microarchitectures. We list the minimal increments, the resolution, the differences between a cache hit and miss for D- and I-cache Flush+Reload, and bars showing the timer's performance with respect to distinguishing these events.

Appendix B. Used Interfaces

Table 10 summarizes the timing, cache-maintenance, and barrier instructions used in our experiments. On ARM and RISC-V, we use standard unprivileged data-cache flush instructions. On T-Head RISC-V CPUs (C906, C908, C910), we use the `XTHeadCmo` vendor extension [44]. On LoongArch, cache maintenance is privileged, so we use a kernel module. For timing, ARM uses `clock_gettime` because access to `CNTVCT_EL0` is restricted [2], while RISC-V and LoongArch use `rdtime`.

TABLE 9: List of timer resolutions for each microarchitecture, minimal increments, and differences between a cache hit and miss for D- and I-cache Flush+Reload. The bars show the timer's performance with respect to distinguishing these events.

Microarch.	Resolution (ns)	Min inc	ΔFR (ns)	ΔFR/*min inc*	ΔiFR (ns)	ΔiFR/*min inc*
Cortex-A520	41.00	41.0	91.0		169.0	
Cortex-A72	119.00	119.0	166.0		135.0	
Cortex-A73	166.00	166.0	171.0		203.0	
Cortex-A76	290.99	291.0	178.0		202.0	
Cortex-A78	76.00	76.0	148.0		63.0	
Cortex-A720	40.00	40.0	83.0		72.0	
Cortex-A725	76.00	76.0	333.0		181.0	
Cortex-X4	40.00	40.0	65.0		38.0	
Neoverse-N1	80.00	80.0	91.0		117.0	
Oryon	100.00	100.0	104.0		8.0	
Kunpeng Pro	104.00	104.0	172.0		45.0	
Icestorm	41.00	41.0	111.0		92.0	
Firestorm	41.00	41.0	115.0		62.0	
C906	41.67	100.0	385.0		350.0	
C908	37.03	100.0	309.0		277.0	
C910	333.34	100.0	87.0		133.0	
X60	41.66	100.0	385.0		0.0	
3A5000-HV	20.00	200.0	1367.0		1244.0	

TABLE 10: List of the interfaces and instructions used in the work for each architecture. On Apple devices, we enable unprivileged flush instructions. On LoongArch64, we use a kernel module for the experiments and out-of-place L1 eviction for the evaluation.

Architecture	Flush (D-cache)	Flush (I-cache)	Memory Barrier	Timer
ARM64	`DC CIVAC`[a]	`IC IVAU`[a]	`DSB ISH`	`clock_gettime`, `CLOCK_MONOTONIC`
RISC-V	`CBO.FLUSH`	—	`FENCE RW,RW`	`RDTIME`
RISC-V T-Head[b]	`TH.DCACHE.CIVA`	`TH.ICACHE.IVA`	`FENCE rw,rw`	`RDTIME`
LoongArch64 (Experiments)	`CACOP 0b10011`	`CACOP 0b10000`	`DBAR 0`	`rdtime.d`
LoongArch64 (Eval)	L1d eviction[c]	L1i eviction[c]	`DBAR 0`	`RDTIME.D`

[a] On Apple devices, enabled via a kernel module. [b] C906, C908, and C910.
[c] 64 kB, 4 ways. Out-of-place eviction using colliding virtual addresses.

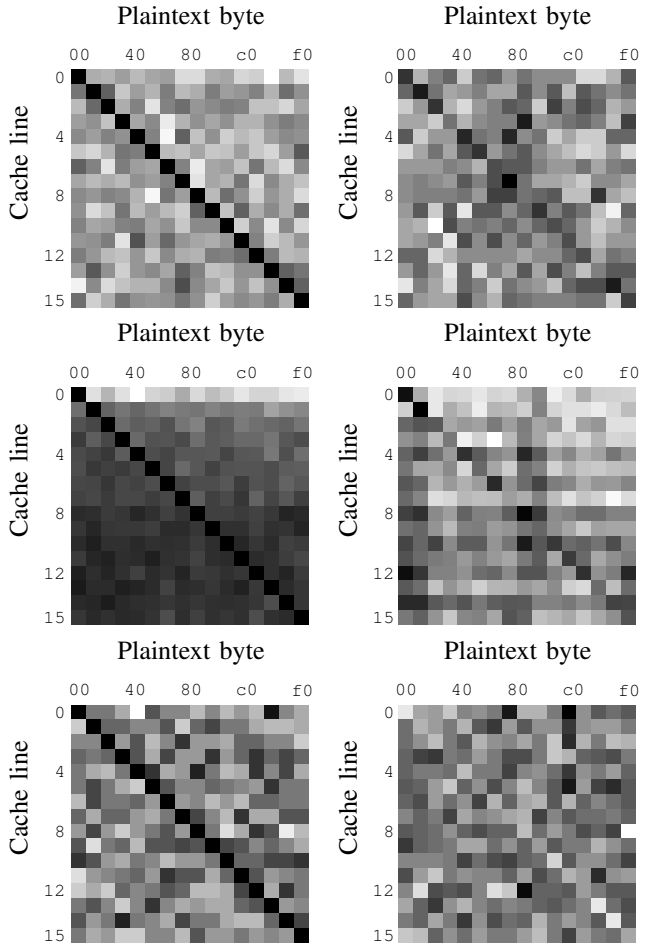

Figure 10: AES T-table cache-access pattern for T-Head XuanTie C910 (top), Qualcomm Oryon (middle), Loongson 3A5000-HV (bottom) with I^{2}SC (left) and Flush+Reload (right).

Appendix C.
AES T-Table Cache-Access Patterns

Figure 10 shows the AES T-table cache-access patterns for the reference microarchitectures.

Appendix D.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper explores both architectural and microarchitectural speculative side channels on 3 different RISC architectures: ARM, RISC-V, and LoongArch. Architecturally visible inconsistencies between the L1 instruction and data caches are demonstrated on multiple processors across multiple architectures. Speculative execution attacks are used to move targeted data into the L1i cache where it can be extracted architecturally. This allows for successful cache attacks without a dependency on an accurate timer, showing that mitigations which focus only on making accurate timers inaccessible are not sufficient.

D.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

D.3. Reasons for Acceptance

1) This paper identifies an impactful vulnerability. The paper shows that side-channel defenses based on eliminating accurate time sources are insufficient.
2) The paper provides a valuable step forward in an established field. Architecturally visible inconsistencies between the instruction and data cache are shown to be a powerful side channel.
3) The paper establishes a new research direction. The paper makes clear that new defense mechanisms will need to be developed.

D.4. Noteworthy Concerns

The PC found the leakage of touch-event timing to be convincing. While this appears to establish the preconditions for exploitability as demonstrated in prior work, the end-to-end attack is not actually shown.