GhostCache: Timer- and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips

Yu Jin
lambda.jinyu@gmail.com
Department of Electronic Engineering, Tsinghua University
Beijing, China

Minghong Sun
smh22@mails.tsinghua.edu.cn
Department of Electronic Engineering, Tsinghua University
Beijing, China

Dongsheng Wang*
wds@tsinghua.edu.cn
Department of Computer Science and Technology, Tsinghua University
Beijing, China

Pengfei Qiu
qpf@bupt.edu.cn
Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education
Beijing, China

Yinqian Zhang
yinqianz@acm.org
Southern University of Science & Technology
Shenzhen, Guangdong, China

Shuwen Deng*†
shuwend@tsinghua.edu.cn
Department of Electronic Engineering, Tsinghua University
Beijing, China

Abstract

Microarchitectural side-channel attacks, which have become increasingly prevalent, often rely on high-resolution timers. Emerging processor architectures have sought to mitigate these vulnerabilities by restricting access to fine-grained timers. In this work, we verify the widespread existence of weak coherence in L1 cache on multiple RISC chips, exploit it to bypass this type of mitigation and propose GhostCache, which constructs timer-free and counter-free instruction cache attacks. It introduces two novel and widely applied attack primitives, Modify+Recall and Call+ModifyCall, which are applicable to both RISC-V and ARM architectures and affect 6 commercial and 3 open-source large RISC processors. To the best of our knowledge, we present the first demonstration of timer-free and counter-free cache attacks on RISC-V processors. We also identify undisclosed features, such as the next-three-line prefetching mechanism and direct forwarding of evicted instructions from data cache to instruction cache. Furthermore, we develop four types of covert channels, achieving up to 1.68 MB/s with a 0.01% error rate. For side-channel attacks, GhostCache enables three types of timer-free real-world attacks. The first is an end-to-end website fingerprinting attack, achieving 92.02% accuracy across 100 website classes. The second is a set of kernel leakage attacks, including the discovery of a new Spectre disclosure gadget via a function pointer to leak arbitrary kernel data at 92.91% accuracy. We also launched an attack to reconstruct cryptographic keys. Lastly, we propose potential countermeasures to address these vulnerabilities in both RISC-V and ARM architectures.

*Also with Zhongguancun Laboratory.
†Shuwen Deng is the corresponding author.

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744833

CCS Concepts

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

Keywords

Security, Micro-architectural side channels, Covert channels

ACM Reference Format:
Yu Jin, Minghong Sun, Dongsheng Wang, Pengfei Qiu, Yinqian Zhang, and Shuwen Deng. 2025. GhostCache: Timer- and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744833

1 Introduction

The cycle-level timer, also known as the high-resolution timer, was originally designed for microbenchmarking and performance optimization [12, 47, 75]. However, it also became a tool for attackers to conduct side-channel attacks such as cache timing attacks [52]. For more than a decade, modern processors have faced increasing microarchitectural side-channel attacks and other vulnerabilities, with timers and counters playing a critical and foundational role [21, 37, 44, 45, 56, 83]. Using these timing or other hardware counters (e.g., performance monitor counter [66]), or counters counted from a sibling thread [42] to build pseudo-timer, attackers can extract sensitive information, such as cryptographic keys, through techniques such as cache timing attacks [30, 83] and power analysis [41].

Emerging processor architectures have increasingly focused on limiting the accessibility of cycle-level high-resolution timer (HRT) to mitigate timing-related information leaks, especially on recent processors [28, 33, 42, 43, 50, 67, 68]. For instance, ARM introduces the `PMUSERENR_EL0` register [47], which allows the configuration of access permissions for performance monitoring counters to reduce the potential for attackers to exploit timing from the processor. Similarly, the RISC-V has implemented `mcyclecfg` and `mcounteren` registers [70], which enable fine-grained control over the privilege modes for cycle and instruction-retirement counters (*mcycle*

and *instret*) [18]. *These HRT mitigation mechanisms are enabled by default in ARM, Apple, and RISC-V chips, thus preventing unauthorized or unprivileged access from leading to potential leaks of timing information.*

To bypass the limitation of timers and counters, several timer-free attacks have been proposed. However, most of the work targets x86-64 and other platforms [17, 36, 79, 81], or specific features and setup of ARM design [27, 77]. In this work, we focus on general instruction cache (denoted as I$) [1] of multiple RISC processors to come up with a timer-free and counter-free attack solution. [2]

We propose *GhostCache*, a class of weak-coherence-based timer-free I$ attacks on 9 RISC-based chips from different ISAs and different vendors. We verify that GhostCache affects at least 6 commercial ARM and RISC-V processors provided by ARM, Apple, and SiFive (ARM Cortex-A53, ARM Cortex A76, Apple Silicon M1 and M4, RISC-V SiFive U74 and P550), as well as 3 open-source large-scale processors including Rocket-Chip [5], SonicBOOM [82], and Xiangshan [73].

To construct such attacks, in addition to the root problem that HRT is crucial but increasingly restricted especially for I$ (C1), we are also faced with the following challenges, where I$ traces (C2) bring even more significant noise than D$, noise may come from branch predictor, instruction prefetcher, also from a newly found data forwarding mechanism from L1D$ to L1I$. Besides, I$ traces (C3) are affected by side-effects of macro-op caches, and (C4) are hard to exploit as Spectre primitives.

As shown in Figure 1, to tackle C1, we develop timer-free and counter-free channels using the *interaction between I$ and D$* and derive the attack primitive. We first invoke a function to be cached in L1I$ and then modify its code. Unless the corresponding cache line is evicted, subsequent calls to the function will execute stale instructions from L1I$. This observation results in an attack primitive called *Modify+Recall* (S1). We also notice that *Modify+Recall* is free from the L1D$ forward mechanism, thus it partially resolves C2. To tackle the remaining issues of C2, we create a new primitive named *Call+ModifyCall* to exclude the influence of the previously undisclosed next-three-line-prefetcher (S2.1), construct a *replacement-policy-orthogonal probing method* to reduce probing noise (S2.2), and develop a *differential mechanism* to filter noise (S2.3). To address C3, we create a *small-overhead misprediction* for macro-op cache to *bypass* its interference with the attack (S3). To address C4, we use the *call from the function pointer as the disclosure gadget* and verify its existence in a real Linux kernel (S4).

Through our exploration, we uncover several previously unknown features in commercial processors. Specifically, we discover a *novel instruction prefetcher that prefetches the next three cache lines of the current cache line* (F1) in SiFive P550, which causes obfuscation of the attacker's observations. We further discover a *forwarding mechanism from L1D$ to L1I$* (F2) in ARM Cortex A76.

We use GhostCache to construct covert channels, in which we show at least four different types of covert channels implemented on ARM and RISC-V processors, including a covert channel that can perform cross-privilege or cross-core attacks on the I$. We achieve a maximum bandwidth of more than 1.68 MB/s with 0.01% error rate in the intra-thread covert channel.

We demonstrate the capabilities of GhostCache through three case studies: 1) real-world end-to-end timer-free cache-based website fingerprinting attacks, 2) cross-privilege kernel-leakage attacks via secret-dependent control flow and instruction-based Spectre disclosure gadgets, 3) cross-context attack on cryptographic code that leaks keys from RSA. Our website fingerprinting attack achieves up to 92.02% accuracy (F_1 score of 91.90%) on 100 websites, surpassing the latest timer-based cache attack [49], which has an F_1 score of 89.3%. For Spectre attacks, we design a novel I$ disclosure gadget using function pointers and identify a real gadget in the Linux kernel. Using our gadget, we achieve kernel secret leakage with 90.840% accuracy.

Besides the attacks, we also propose potential countermeasures in both RISC-V and ARM ISAs. Last but not least, we conduct a systematic analysis and create a taxonomy of the cache state observation methods in the related work section.

This paper makes several contributions, including that we:

- Propose GhostCache weak-coherence-based timer-free I$ attacks, which affect at least 6 commercial ARM and RISC-V processors and 3 open-source large-scale RISC-V processors. To the best of our knowledge, we have developed *the first timer-free L1I$ attacks in RISC-V*.
- Uncover a novel instruction prefetcher that prefetches the next three instruction lines on demand and a forwarding mechanism from the D$ to the I$.
- Develop four types of covert channels that can, for example, perform cross-privilege or cross-core attacks on the I$. Variants can achieve up to 1.68 MB/s bandwidth with a 0.01% error rate.
- Design a timer-free cache-based website fingerprinting attack that, to the best of our knowledge, achieves the highest top-100 classification accuracy (92.02%) among timer-free cache attacks, surpassing even the latest timer-based attack.
- Propose and demonstrates, to the best of our knowledge, the first Spectre Unmask Instruction Cache (IC) gadget in the Linux kernel.

Responsible Disclosure: We have disclosed our work and results to the affected vendors and received responses. ARM PSIRT appreciated our disclosure and accepted the risk of GhostCache on November 6, 2024. SiFive acknowledges the applicability of our attack scenario and plans to disclose it via a security bulletin.

2 Background

2.1 Cache Side-Channel Attacks

Most cache side-channel attacks can be represented using a three-step model [16]. Step 1 sets the cache line into a known state. Step 2 modifies the state of the cache line. Finally, based on the timing, Step 3 observes the change in the state of the cache line. As an example of a Prime+Probe [45, 52, 53] attack, first, the attacker fills all the cache set by memory accesses, during the victim's access, the victim may evict different attacker's cache lines depending on the confidential data (e.g., the different secret values to index AES T-table in AES key attack [52] using Prime+Probe), after that, the attacker re-accesses the memory and measures the latency (which

[1] Accordingly, we denote data cache as D$.
[2] Throughout our work, we use the term "timer-free" to refer to both the timer-free and counter-free features.

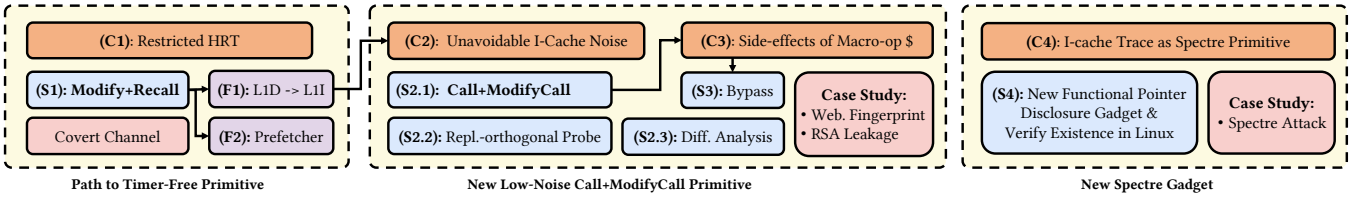

Figure 1: Overview of our research roadmap, including the challenges (C#), the solutions (S#), the findings (F#) and covert-channel and case study applications.

reflects a cache miss or hit) to find out which cache line has been evicted to infer the secret.

Most data-based cache side-channel attacks focus on data cache [53] or unified cache [45] by exploiting the change of the cache state to leak secrets. The instruction cache attacks [1, 2] are relatively less. Researchers have proposed using timing differences in memory-related operations as a method to attack software [3, 7, 9, 28, 53]. These timing-based side-channel attacks often target cryptographic applications, where precise timing measurements can reveal sensitive information. For example, software implementing AES encryption or decryption with table lookup is particularly vulnerable to timing-based attacks [13]. By timing the memory access of specific locations, an attacker can infer the values of the secret keys used.

More recently, timing-based channels have served as a fundamental building block for transient-execution attacks, such as Spectre and Meltdown, which exploit speculative execution vulnerabilities in modern processors. These attacks mostly leverage timing-based channels to extract data that is otherwise protected by isolation boundaries. Notable examples include Spectre [37], Meltdown [44], NetSpectre [60], and various Spectre variants [39].

When the high-resolution timer is restricted, an attacker can design a pseudo-timer using a loop counting from a sibling thread (counter-thread) [42, 59], or amplify the cache latency for the coarse-grained timer [34, 35, 72]. For a restricted threat model that targets a trusted execution environment, the performance counter [21, 42] can also be used to measure the cache state.

2.2 Instruction Cache in Microarchitecture

Modern microarchitecture is combined with three key components, frontend, backend, and memory subsystem, as shown in Figure 2. The instruction cache is commonly within the L1 cache of the memory subsystem. Connecting between the L2 unified cache and frontend to supply the instruction to be fetched, dispatched, and executed in the backend, as we show in Figure 2.

The target address is predicted or obtained from the branch prediction unit or the redirect path, and the instruction at that address is obtained from the instruction cache. Above the L1 instruction cache, there can be another instruction buffer named macro-op cache, which stores the instructions decoded in the pipeline. When the macro-op cache gets a cache hit, the execution stream can avoid fetching data from the L1 instruction cache.

Beyond its basic function, cache often includes mechanisms such as maintaining cache coherence. Unlike x86-64, for ARM and RISC-V ISA, L1$ can be implemented in a weak-coherence-based manner, meaning that hardware cache coherence is optional and may not

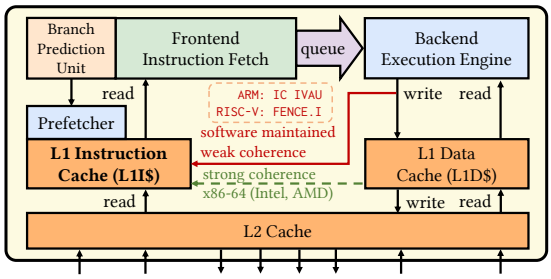

Figure 2: Key components of a modern RISC microarchitecture, including the frontend, the backend, and the memory Subsystem.

be enforced by hardware. For ARM chips with CTR_EL0.DIC == 0 or CTR_EL0.IDC == 0, barriers such as invalidation are needed to ensure L1I$ and L1D$ coherence. In RISC-V, the `FENCE.I` instruction ensures that subsequent instructions fetched reflect previous completed store instructions.

2.3 Self-Modifying Code

Self-modifying code (SMC) presents a notable exception, where code actively alters its own instructions or executable memory during execution. Common applications of SMC include bootloaders and ELF loaders, debugging tools that dynamically modifies program code, as seen with tools such as GNU Debugger, and other just-in-time (JIT) techniques that generate native code at runtime for dynamic optimizations.

3 Attack Primitives

In this section, we discuss four main challenges and their corresponding solutions for building timer-free RISC-based instruction cache attacks. We uncover and analyze new features of RISC architectures that affect attacks. Based on the analysis, we introduce the GhostCache attack, which contains two types of attack primitives, Modify+Recall and Call+ModifyCall, to overcome difficulties brought by the new features and enable real-world attacks.

3.1 Challenge: Restricted HRT

Challenge 1 (C1): High-resolution timers are crucial yet increasingly restricted for instruction cache attacks.

Instruction cache (I$) attacks usually require accurate timers more compared to data cache (D$) attacks. I$ accesses are tightly coupled with speculative execution and branch prediction, which

produce transient and small timing variations that are harder to measure [37]. Furthermore, the optimized and predictable nature of I$ operations, combined with pipeline-level integration, reduces observable timing discrepancies, necessitating high-resolution timers to distinguish cache activity accurately [14, 26] .

Both L1I$ and L1D$ require a high-resolution timer for accurately priming and probing. On P550, using `rdtime` instruction instead of `rdcycle` will result in an error rate of 46.875% when trying to perform Prime+Probe in L1I$. Using `rdtime`, timing differences of L1 hits and misses range from 0 to 2 ticks, and most of the L1 misses will get 0 latency difference (72.2% of 1000 measurements). This indicates that its granularity is insufficient to distinguish between L1 hits and misses.

A common defense against side-channel attacks is limiting high-resolution timers. AMD restricts timer resolution on many CPUs [43], while ARM makes hardware timers privileged [42]. RISC-V introduces `mcounteren` to restrict timer access [18]. Counter-based alternatives, relying on sibling threads, face challenges like scheduling issues and Dynamic Voltage Frequency Scaling (DVFS), making them less accurate and noisier than native timers.

Solution 1 (S1): Create timer-free attack channels using coherence between I$ and D$.

All timer-free methods have the same characteristics, that is, to convert a microarchitecture activity, such as a cache miss/invalid, a memory store (even transient), or some instruction's effect, to an architecturally visible state. Based on the above principle, we conduct an extensive analysis and evaluate in different ISAs and different implementations, shown in Table 1, to find whether there exists an opportunity to create a window that can reflect the status of the microarchitecture. Using operation steps shown in Equation 1, we notice that we can create an incoherent state and observe the cache behavior of the state. As cross-core coherence and MESI protocol (invalidate-based cache coherence protocol) have already been well researched, we focus on the coherence implementation between L1I$ and L1D$.

We design a benchmark using self-modifying code (SMC) to evaluate L1I$ and L1D$ coherence across ISAs. The benchmark modifies executable memory after an initial call and recalls it to observe if the modified instruction executes. Using this, we evaluate across x86-64 (Intel and AMD), ARM, and RISC-V chips, confirming weak L1 cache coherence in tested ARM chips [47]. We further *reverse-engineer* that both the commercial and open-source RISC-V also choose to implement weak L1 cache coherence. [3]

$$Call \Rightarrow Modify \Rightarrow Recall \tag{1}$$

To exploit this incoherence, we notice that if the stale cache line in L1I$ is evicted, the modified instruction data (new value) will be fetched from L1D$ or L2$ and used to execute, even without an invalidation or serializing instruction. Based on this behavior, we can do direct stale/new cache line reads to leak whether there was victim executions mapping to the same set and doing eviction, without using timers. This attack primitive, we name it Modify+Recall (M+R), can be formulated as:

[3]Although our benchmark does not create incoherence on the x86-64's L1 cache, the prefetch queue, as being documented, may still be able to do that. We leave this for future work.

Table 1: Tested environment and systematic evaluation results for Modify+Recall (M+R). W-co. means the weak coherence in the L1$. The ARM Cortex-A53 runs Debian GNU/Linux 11 (bullseye), and the ARM Cortex A76 runs Ubuntu 24.04.1 LTS. M1 Max runs macOS 15.1 (24B83). [4] The SiFive P550 runs Debian GNU/Linux.

CPU	Linux Kernel	W-co.	L1I$ Sync
Intel Xeon 6438Y+	5.15.0-130-generic	✗	(Hardware)
AMD EPYC 9554	5.4.0-196-generic	✗	(Hardware)
ARM Cortex A53	5.10.110-1-rockchip	✓	IC IVAU
ARM Cortex A76	6.8.0-1013-raspi	✓	IC IVAU
Apple M1	6.10.11-linuxkit	✓	IC IVAU
Apple M4	6.8.0-47-generic	✓	IC IVAU
SiFive U74	6.6.20-starfive	✓	FENCE.I
SiFive P550	6.6.18-eic7x	✓	FENCE.I
Rocket Chip & BOOM	N/A	✓	FENCE.I
Xiangshan	N/A	✓	FENCE.I

Attack Primitive 1 (Modify+Recall)

$$Call \Rightarrow Modify \Rightarrow Victim\ Execution \Rightarrow Recall \tag{2}$$

We have formulated the Modify+Recall attack primitive for monitoring L1I$ activity in a timer-free manner. Given a memory page `func`, whose memory protection bits are set as `PROT_READ | PROT_WRITE | PROT_EXEC` via `mprotect` or `mmap` system call, the initial primitive consists of the following steps (demonstrated in Figure 3):

- **Step 1**: Invoke the `func` to ensure that it is cached in L1I$.
- **Step 2**: **Modify** the `func` via a store instruction and write different instructions that have different behavior, e.g., returning a different register value. With that, L1D$ is modified and incoherent with instructions in L1I$.
- **Step 3**: Invoke the `sched_yield` system call to yield the CPU to the victim, or just call the victim's function (e.g., vulnerable kernel system call).
- **Step 4**: **Recall** (invoke) the `func` and observe the execution results to infer whether the stale instruction in L1I$ has been evicted by victim's function.

To observe the victim's eviction in step 4, the attacker prepares at least four set-aligned `func` to fully evict a 4-way L1I$ cache set.

Fetch from D$ to I$. Furthermore, we observe that even when the L1I$ set is confirmed to be evicted using Modify+Recall, it does not exhibit a larger or distinguishable memory access latency, making the eviction unobservable. Specifically, without invalidating the instruction in L1D$, the function call with L1I$ hit and miss (evicted) incurs the same latency of 92 cycles (measured via `PMCCNTR_EL0`). However, with invalidation, the cycle counts are 92 for hits and approximately 118 to 128 cycles for misses. This suggests that instruction fetch operations prioritize L1I$ but also perform a lookup in L1D$ when necessary. If a cache line in L1I$ is evicted, it is likely

[4]As macOS has limited the use of SMC, for convenience, the experiment has been conducted in its Docker environment with Linux 6.10.11-linuxkit in Apple Virtualization framework.

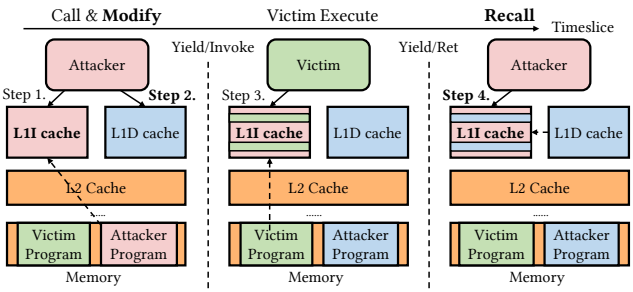

Figure 3: Overview of Modify+Recall.

retrieved and forwarded from the L1D$ block. This is demonstrated in Cortex-A76. The key point of GhostCache is that the stale value in L1I$ is evicted or invalidated. The forward mechanism will not affect the stale value, but it will make it faster to retrieve modified instruction data from L1D$ rather than L2$ during recall for an evicted instruction cache line.

Finding 1: If a cache line in L1I$ is evicted, it is likely fetched and supplied from the corresponding block in L1D$.

3.2 Challenge: Unavoidable I-Cache Noise

Challenge 2 (C2): Unavoidable instruction cache traces create significant noise.

Although we do not exploit timer, cache eviction pollutions can generate noise to impede high-resolution attacks. Furthermore, potential instruction cache prefetchers will bring more cache pollution.

Uncover Next-three-line Prefetcher using Modify+Recall. In attempting to perform a full-set eviction on the P550 processor, as we show in Figure 4, a cache miss triggers prefetching and loads the requested line into the cache while evicting a stale one from the Modify phase. Timing measurements confirm that all three prefetched consecutive cache lines are L1 hits. This behavior suggests the presence of an instruction prefetcher that preemptively fetches the next 3 cache lines. We speculate that this undocumented feature is designed to optimize sequential instruction access by anticipating upcoming lines and loading them into the cache ahead of time. From the evaluation result, we also notice that it does not perform cross-page prefetch, as indicated by the arrow in Figure 4. When the missed cache line is adjacent to the page boundary, the subsequent cache line on the next page will not be prefetched to cause eviction.

Prefetcher obfuscation to the attacks. A prefetcher that fetches lines across multiple sets at once can introduce complex interactions between cache management and prefetching, potentially *impacting the effectiveness of cache-based side-channel attacks.* As shown in Figure 5, when conducting a cache attack on P550, if 4 consecutive set evictions are observed, it can be determined that there is 1 instruction fetched with 3 cache line prefetched, while the 3 prefetched lines may hide later access to these address since executing to prefetched lines will not trigger prefetching. This means, witnessing 4 consecutive set evictions maps to $2^3 = 8$ types of victim's instruction execution patterns. Moreover, the inability to distinguish certain instruction executions increases as the number of sequential evicted traces exceeds 6. Specifically, when n consecutive evictions are observed with $6 \leq n \leq 8$, there are 2^{n-2} indistinguishable victim access patterns, making it unclear whether they result from prefetching or actual accesses. This phenomenon also accounts for the lower classification accuracy of website fingerprinting attack on the P550, as discussed in Section 5, compared to the Cortex A76.

Finding 2: We identify a next-three-line prefetcher, which prefetches the next three cache lines of the current cache line and maps them to the subsequent three cache sets.

Solution 2.1 (S2.1): Generate new attack primitive to avoid prefetching side effects.

To mitigate the obfuscation caused by the next-three-line prefetcher, we analyze the interaction between prefetching and our Modify+Recall primitive. We speculate that prefetcher is triggered only when the stale data held in L1I$ is evicted. This mechanism prioritizes filling potentially useful entries without evicting valid instructions. To avoid triggering prefetching, we re-design the operations to make sure that the "stale" state is not demonstrated before the victim's execution and introduce a new attack primitive, Call+ModifyCall:

$$\textbf{Attack Primitive 2 (Call+ModifyCall)} \quad Call \Rightarrow Victim\ Execution \Rightarrow Modify \Rightarrow Recall \tag{3}$$

The new attack primitive Call+ModifyCall (C+MC) is shown in Figure 7 and consists of the following steps:

- **Step 1**: **Call** the `func` to ensure that it is cached in L1I$.
- **Step 2**: Invoke the `sched_yield` system call to yield the CPU to the victim. Notice that different from Modify+Recall, there is no stale L1I$ at this step.
- **Step 3**: **Modify and Call** the `func` to detect whether there is still a cache line copy within the L1I$.

In 10 tests, the L1I$ eviction of Modify+Recall consistently activates the next-three-line prefetcher (100%), while Call+ModifyCall did not (0%), likely due to differences in stale data states.

Case Study Effect Highlight: Website Fingerprinting Attack. Using Modify+Recall, the website fingerprinting attack can achieve high accuracy (>90%) in Cortex A76. While in P550, because of the effect of next-three-line prefetcher, the accuracy is relatively low. When using the improved Call+ModifyCall, as is demonstrated in Figure 6, the noise caused by the next-three-line prefetcher is eliminated, and the final accuracy is much higher in P550.

Solution 2.2 (S2.2): Construct replacement-policy-orthogonal probing method.

Cache side-channel attacks are also shown to suffer from noises induced by set thrashing [42] to accurately do priming and probing related operations. Different replacement policies such as pesudo-LRU and random replacement policies will require customized attack access pattern and need to overcome indeterminism [42]. In this case, we discover the advantages of our GhostCache attack primitives to accurately control and access arbitrary way and set combination of the L1I$.

Our reverse engineering strategy is similar to previous work [77]. We occupy all the ways of an instruction cache set via function

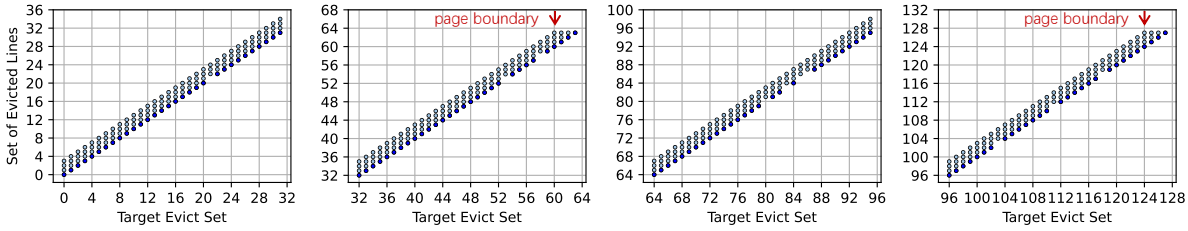

Figure 4: Observation of Cache line eviction across different cache sets in SiFive P550. The deep blue dots represent observed evictions in the corresponding cache sets. Additionally, the next three cache lines, mapped to the subsequent three cache sets (highlighted in light blue), are also observed to be prefetched, evicting the stale lines. This behavior suggests the presence of a next-three-line prefetcher. As shown in the top-right of the second and last sub-figures, this prefetcher does not operate across page boundaries.

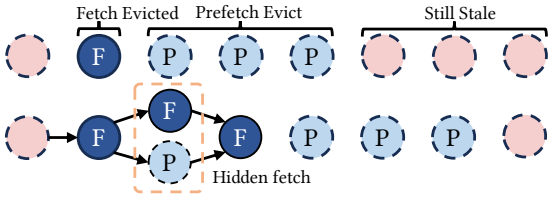

Figure 5: Different scenarios when several consecutive cache lines are observed as evicted. A single cache access triggers the prefetching of the next three lines and evicts original stale lines. When multiple cache lines are evicted, it may reflect additional victim execution scenarios, making it challenging to deduce the exact cache set access sequence from the eviction trace.

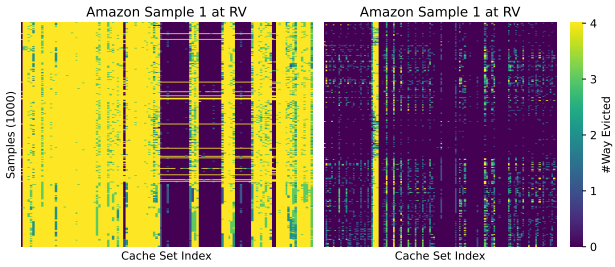

Figure 6: The instruction trace of website fingerprinting in P550. Left uses Modify+Recall; right uses Call+ModifyCall.

calls and record the order of the calls, then traverse an eviction set and observe which cache line is evicted, thus inferring the replacement policy. As illustrated in Figure 8, we sequentially fill each way of L1I$ set 0 with stale instructions and L1D$ set 0 with modified instructions during Modify step.[5] Subsequently, we execute three instructions that map to set 0. Our findings reveal that the stale instructions in the first three ways are evicted, with the execution results corresponding to the modified instructions.

[5]According to ARM's documentation [48], the Cortex-A76 features a 4-way set-associative L1I$.

This behavior is consistently observed for evictions across 1 to 4 ways. Using it, we reverse-engineered that the replacement policy in P550 employs an LRU-like policy. We further confirm that the replacement policy in the L1 instruction cache (L1I$) of the ARM Cortex-A76 is pseudo-LRU as documented. While we mention stale cache in the previous section, we do not observe differences of replacement policy behavior related to cache staleness.

Upon identifying that the replacement policy is pseudo-LRU, we can reverse the recall order, in which the last `func` that was called before `modify` will be the first to be `recalled`, thus reducing the additional eviction of residual `stale` cache lines during probe (reducing pollution by 50.3% under single cache line eviction). Also, compared with timer-based iPrime+iProbe, our approach does not need to invalidate the corresponding D$ block.

Solution 2.3 (S2.3): Design a noise canceler utilizing differential analysis.

To achieve a cache attack, the attacker can use `sched_yield` to give the timeslice to the victim, or a system call to execute the secret-dependent control-flow gadget or trigger speculation gadgets in kernel. The `sched_yield` and system call have inherent side effects that will evict the L1I$ and cause eviction noise. To solve this issue, we run the differential analysis, designing the attack to run twice, with the first run without victim execution or not triggering a vulnerable gadget to collect the background noise. After the second run, we calculate the differential result to even out the noise, and finally get the execution side effects caused by the secret.

Case Study Effect Highlight: Kernel Leakage Attack. We also apply this differential method for kernel leakage attacks. Without it, secret-dependent control flow is hard to distinguish because of too many irrelevant evictions. With eviction traces of known data as background noise, secret trace's additional evictions become clearer, as shown in the example in Figure 9. Here, the background noise is collected independently of victim website fetches, allowing noise from `sched_yield` system calls to be effectively filtered.

3.3 Challenge: Side-effects of Macro-op Cache

Challenge 3 (C3): Interference due to the side-effects of ARM's macro-op cache.

When we conduct an attack using Call+ModifyCall, we notice that after the modification in Cortex A76, without manual `ic ivau`,

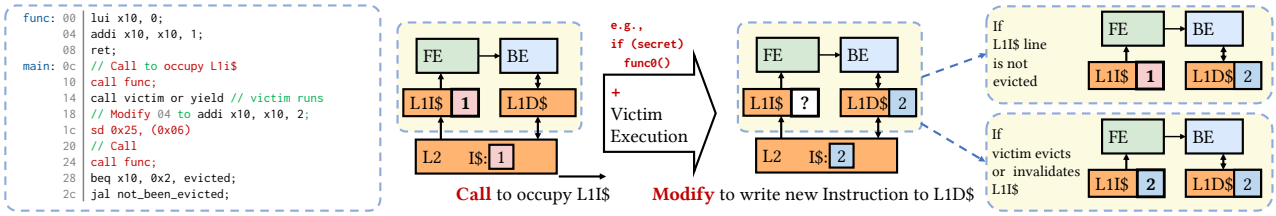

Figure 7: Monitor the eviction of L1I$ using Call+ModifyCall.

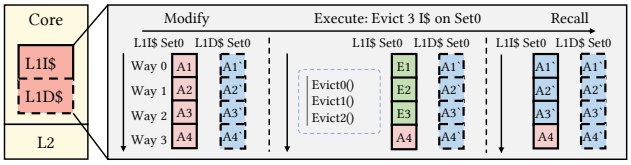

Figure 8: Cache operations to reverse-engineer the replacement policy. A minimum of four accesses is required to evict the youngest entries in the L1I$.

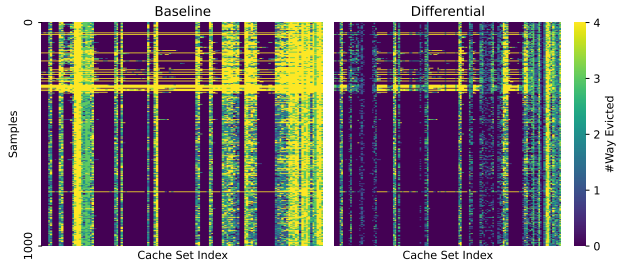

Figure 9: Website fingerprinting traces from google.com in Cortex A76. Left: raw cache eviction result; Right: differential analysis result. Lighter colors means more L1I$ evictions.

even if the L1I$ is evicted, our execution result is still from the stale instruction which reduces the attack effects.

Even though the stale value in L1I$ is evicted (causing L1I$ miss), the executed instruction is still the stale one, which indicates that there is an L0 macro-op cache storing the stale instruction. ARM's manual [48] also supports this.

Solution 3 (S3): Induce a small-overhead misprediction for macro-op cache to bypass its interference.

Building on prior research on L0 cache behavior [14, 57], we address this challenge by inducing a misprediction before recalling the modified instruction. Adding a for loop with an unexpected break appears to bypass the macro-op cache in Cortex A76. Reverse-engineering with `PMCCNTR_EL0` confirms this. Without misprediction before the execution, cycle difference between L1I$ hit (`13-35` cycles) and miss (`22-30` cycles) is small. With misprediction, the cycle difference between L1I$ hit (`14-40` cycles) and miss (`42-59` cycles) arises, which suggests that the macro-op cache is probably bypassed and eviction takes effect in L1I$.

3.4 Challenge: I-Cache Trace for Spectre

Challenge 4 (C4): Instruction cache traces are rarely exploited as Spectre primitives.

Previous work [32] has revealed Spectre disclosure gadgets, which can rely on branch misprediction to be speculatively executed and finally leak the secret of the victim via a covert channel. There are two types of disclosure gadgets [32], masked and unmasked gadgets. They can be further classified as Instruction-signal gadgets (IC gadgets) and Data Cache-signal gadgets (DC gadgets). Existing works [71] have only exploited the masked and unmasked DC gadgets. Besides, finding the real Spectre gadget is also challenging.

Solution 4 (S4): Use the function pointer call as the disclosure gadget.

To transmit the secret via L1I$, intuitively, a secret-dependent function call is needed (IC gadget). We design several gadget patterns and finally propose a real existing kernel gadget using the function pointer.

```
1  // masked DC gadget, use by Spectre V1
2  void masked_gadget(long *secret)
3      { array[(*secret & 0xff) * 4096];  }
4  // unmasked DC gadget, can by Spectre V2
5  void unmasked_dc_gadget(long **secret)
6      { **secret; }
7  // This work: unmasked IC gadget
8  void unmasked_ic_gadget(long **secret) {
9      (*secret)(); } // ldr x0, [sp, #4]; blr x0;
```

Listing 1: Transient leakage gadget, we find real-world unmasked IC gadgets.

Case Study Effect Highlight: Instruction-As-Disclosure-Gadget Spectre Attack. Here we conduct a review of the source code of the Linux kernel and discover that the unmasked IC gadget, formed as `(*secret)()`, exists in the Linux kernel and can be exploited. The `posix_lock_inode` of `fs/locks.c` has an IC gadget gadget: `(*func)();`. This type of gadget will load secret data from `*secret`, and then use the secret value to conduct an indirect branch jump. We demonstrate different types of Spectre gadgets and the corresponding ARM assembly instructions after our `(*secret)()` gadget at Listing 1.

4 Building Timer-Free Covert Channel

In this section, we show how we exploit the GhostCache primitive to construct four distinct covert channels, as shown in Figure 10, leading to the final real-world attack in section 5 and section 6.

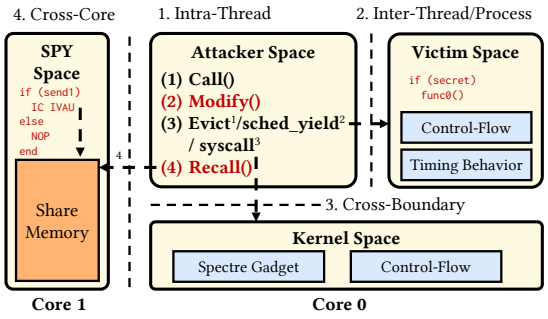

Figure 10: Overview of GhostCache's Covert Channel.

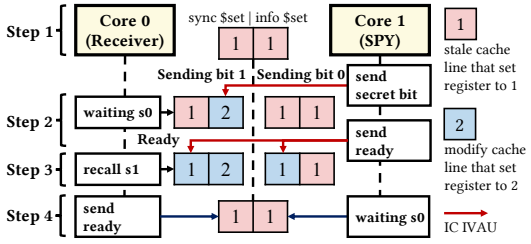

Figure 11: Cross-core timer-free sync covert channel in ARM, including a synchronization channel $ set (left) and an information channel $ set (right). The right cache set is used to transmit the secret, and the left one is used to mark the ready state of this round of transmission.

4.1 Intra-Thread Covert Channel

In an intra-thread covert channel, we confirm that *data access does not affect the contents of the L1I$*. This stability allows information to be communicated within a single thread by controlling and observing the L1I$ state. In this case, we also do not have the interference typically caused by data cache interactions, thus we have less noise. As a proof-of-concept covert channel, we demonstrate its widespread presence across ARM and RISC-V chips, verified on at least six commercial processors from various vendors and ISAs, as detailed in Table 1.

4.2 Inter-Thread/Process Covert Channel

In an inter-thread/process covert channel, we exploit the fact that *context switching between threads and processes does not flush the entire L1I$*, allowing the cache state to persist across context switches. This persistence makes it possible to establish a covert communication channel between threads by modulating and detecting changes in specific L1I$ lines, even when threads are isolated from each other in terms of data. This covert channel leads to our cache-based website fingerprinting attacks in section 5.

4.3 Cross-Boundary Covert Channel

We observe that *different system calls lead to evictions in different sets within the L1I$*. This behavior indicates that the state of the L1I$ can persist through privilege mode transitions, such as those between user mode and kernel mode. Using this channel, a kernel-mode spy process can signal information to a user-mode attacker by selectively evicting cache sets based on system calls, creating a bridge across privilege boundaries. This covert channel leads to kernel control flow leakage attacks in section 6.

4.4 Cross-Core Covert Channel with Synchronization: Modify+(Invalid)+Recall

We review and explore ARM's cache management instructions. Through experiments, we find that although not explicitly stated in the manual [47, 48], the `IC IVAU` instruction can invalidate L1I$ lines even cross-core, as is shown in Figure 10. Arm's documentation [48] states IC IVAU is a user-level instruction that invalidates I$ via the point of unification (PoU). Tests show Arm A76 supports cross-core invalidation, while RISC-V P550 does not, but on A76, the invalidation effects stay within a process and do not cross security boundaries. By leveraging this instruction, we can create a cross-core covert channel, allowing processes on different cores to communicate covertly by controlling and observing specific cache blocks across cores.

Cross-core synchronization. Without the timer, it is more challenging to synchronize across different cores for stable transmission. In this case, we carefully design a mechanism to create a cross-core synchronization covert channel of GhostCache attack without relying on a timer (as shown in Figure 11):

- **Step 1**: Prepare two L1I$ cache lines in different cache sets that have stale instruction. The receiver waits the spy on another core to invalidate the left set.
- **Step 2**: Once the spy sends a covert bit by invalidating the right cache set, the spy invalidates the left cache set to notify the receiver that the covert channel is ready to receive.
- **Step 3**: Once the receiver recalls and finds that the left L1I$ cache set holds new instruction, it means the covert data is ready and the receiver recalls the right cache set to infer the secret information.
- **Step 4**: Receiver sends "ready" and go back to step 1 for the following round.

For RISC-V multi-core chips, the `FENCE.I` instruction does not guarantee that other RISC-V harts [6] observe the local hart's stores during their instruction fetches, which means that it is hard to utilize our two-bit synchronization scheme. However, we notice that a ratified but unimplemented RISC-V "CMO" extensions [63] introduce the following three instructions: `cbo.clean`, `cbo.flush`, and `cbo.inval`. These instructions can enable GhostCache's RISC-V cross-core channel.

Finding 3 : ARM's invalidation instructions can invalidate the L1I cache across cores, facilitating attack construction.

4.5 Evaluation of Covert Channels

We evaluate the bandwidth and error rate of the above covert channels (result shown in Table 2). In addition to the Cortex-A76, we also evaluate the intra-thread GhostCache channel on Apple M4

[6] Harts in RISC-V refer to hardware threads [70], or an execution environment within a logical core.

Table 2: Covert channel evaluation using GhostCache on ARM Cortex A76, including bandwidth and error rates (ERs).

Channel Model	Scenario	Bandwidth	ERs
Intra-Thread (I$ PP)	PoC	837.6 kB/s	1.90%
Intra-Thread (D$ PP)	PoC	730.8 kB/s	4.00%
Inter-Thread/Process (D$ PP)	Context Switch	3.08 kB/s	0.6%
Cross-Boundary (D$ PP)	User-Kernel	41.7 kB/s	0.3%
Intra-Thread (M+R)	PoC	1685.7 kB/s	0.01%
Inter-Thread/Process	Context Switch	1280.2 B/s	1.30%
Cross-Boundary	User-Kernel	31.6 kB/s	0.00%
Cross-Core (Sync)	Spy's `IC IVAU`	11.2kB/s	0.00%

and SiFive P550, achieving 104.242 kB/s with a 0.004% error rate and 1013.407 kB/s with 0% error, respectively.

Comparison with Prime+Probe. In covert-channel scenarios, we can systematically analyze finer-grained timing and spatial characteristics, measuring bandwidth and error rates to define the upper granularity limit of corresponding side channels. We implemented a similar 1-bit covert channel using state-of-the-art Prime+Probe [58] on L1I$ and L1D$, named iPrime+iProbe and dPrime+dProbe, respectively. We evaluate three types of timers in ARM chips, including POSIX function `clock_gettime`, `CNTVCT_EL0` that are commonly readable for user programs at EL0, and `PMCCNTR_EL0` that requires to be enabled from kernel space. In our experiment, at Cortex-A76, the `CNTVCT_EL0`'s resolution is not enough to detect whether the L1 instruction cache set is evicted. The `clock_gettime` and `PMCCNTR_EL0` have enough resolution, while the former is prone to noise [77] and has limited resolution in the operating system as it relies on the system call, and the userspace (EL0) access to later needs to be enabled in privilege mode.

In Prime+Probe covert channels, sender and receiver use an L1 eviction set and timer-based latency to detect cache evictions, with thresholds pre-profiled using an HRT. The experiment result of iPrime+iProbe and dPrime+dProbe to transfer 64K random bytes has been shown in Table 2. For dPrime+dProbe, the channel speed is mainly affected by the barrier instruction (e.g. `ISB; DSB`).

As shown in Table 2, GhostCache achieves approximately twice the speed of comparable intra-thread covert channel attacks. GhostCache outperforms iPrime+iProbe on A76. In A76, GhostCache can also achieve 18.1% higher bandwidth in boundary-crossing and almost quadruple the bandwidth in context-switching compared with Prime+Probe. This advantage arises because iPrime+iProbe requires handling duplicate cache line copies in D$ and I$ due to our newly discovered forwarding mechanism (F2). Therefore, after the prime, `DC CIVA` instructions are needed to invalidate all D$ copies in order to enable observable timing differences from I$ interference caused by the victim. In contrast, GhostCache bypasses this extra invalidation step, resulting in improved efficiency. For dPrime+dProbe, barrier instructions such as `ISB` and `DSB` are needed to reveal the latency of different memory data accesses, while GhostCache does not rely on the timer, thus we get rid of barrier instructions and are faster.

Table 3: Evaluation of our timer-free cache-based website fingerprinting attacks. Call+ModifyCall can achieve distinctly higher accuracy in P550 than Modify+Recall since it eliminates noise from our discovered prefetcher (F1).

Primitive	Platform	#Class	Accuracy	Precision	Recall
M+R	A76 (ARM)	15	97.98%	98.03%	97.98%
	A76 (ARM)	100	92.02%	91.98%	92.00%
	P550 (RISC-V)	15	95.06%	95.22%	95.07%
	P550 (RISC-V)	100	87.60%	87.71%	87.63%
C+MC	P550 (RISC-V)	15	97.50%	97.53%	97.52%
	P550 (RISC-V)	100	90.05%	90.23%	90.07%

5 Case study 1: Timer-Free Cache-based Website Fingerprinting Attack

In this section, we construct an end-to-end cache-based website fingerprinting attack utilizing the GhostCache.

5.1 Threat Model

We follow previous work [79]'s threat model, considering an unprivileged attacker without access to architectural timing primitives. Additionally, the attacker is assumed to have the ability to set the processor core affinity of its threads, which does not require elevated privileges on Linux and even Windows.

5.2 Attack Implementation: Spying L1I$

The form of website fingerprinting [61, 62] poses significant privacy risks, particularly in scenarios requiring strong user anonymity. Web browsing generates distinctive cache access patterns influenced by factors such as network request, content structure, page layout, and script execution.

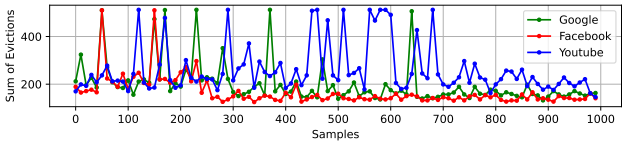

Figure 12: Processed L1I$ eviction patterns on the Cortex-A76.

In this case study, by observing the state of the L1I$, an attacker can use GhostCache to construct fingerprints corresponding to specific websites visited by a user. The attacker gathers L1I$ eviction sequence patterns, which can be classified to identify specific websites. This approach enables the inference of browsing behavior or the identification of the victim's visited website without relying on conventional timing-based techniques.

In our real-world attack, the attacker binds its thread to the same CPU core as the victim's browser and performs GhostCache. The attack involves alternating between monitoring L1I$ evictions and deliberate busy-waiting, using one or more calls to `sched_yield` to relinquish the time slice to the victim. By tracking the number and location of evicted cache sets across these iterations, the attacker constructs a sequence of eviction sets without relying on a timer.

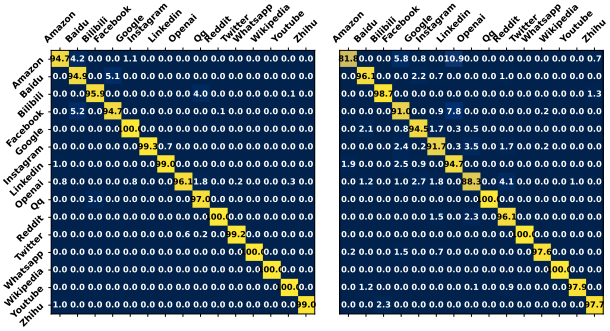

Figure 13: Confusion matrixs using GhostCache for the top-15 website classification on ARM (Left) and RISC-V (Right).

Although this sequence does not follow fixed-time intervals or directly map to real-time events, further analysis demonstrates its ability to effectively capture cache activity patterns, enabling highly accurate inference of the victim's browser behavior.

5.3 Website Classification and Investigation

For the visited websites, we follow established practices [79], selecting the 100 most-visited websites from the Alexa Top list. For each website, 100 traces are collected. The dataset is split into training and testing sets (4:1 ratio). Training involves repeated splits: 15-class classification is conducted 100 times, and 100-class classification 10 times. Final accuracy is averaged across iterations.

We observed that invoking `sched_yield` evicts a fixed set of cache sets, regardless of the victim program's state, as shown in Figure 9 and Figure 6. However, when the victim accesses the L1I cache, evictions span nearly all sets. Thus, we focus on counting total evicted sets, ignoring specific patterns. For classification, we used `TimeSeriesForestClassifier` from `sktime` [46], which segments time series into intervals and extracts features like mean, standard deviation, and slope, as shown in Figure 12. These features train a Random Forest classifier with `n_estimators=500`.

5.4 Experimental Setup and Evaluation

The experiments were conducted using the Chromium browser [54], which is an open-source browser, as the official Chrome browser does not support RISC-V architectures yet. We perform evaluations on two platforms: ARM-based Raspberry Pi 5 (Cortex A76) and a RISC-V development board (P550). On the P550, the browser (Chromium 125.0.6422.112) is operated in headless mode (rendering and generating screenshots without a physical display), while on the Raspberry Pi, the browser (Chromium 131.0.6778.139 snap) runs with a standard display setup with a real physical monitor. We summarize and show the classification evaluation results in Table 3 and Figure 13. Using GhostCache, we achieve high-accuracy website classification across ARM and RISC-V architectures, with ARM reaching a top-100 classification accuracy of 92%. Furthermore, our results demonstrate that I$ traces from different websites remain highly distinguishable. Even as the number of websites increases from 15 to 100, the classification accuracy exhibits minimal decline.

Regarding the degradation of classification accuracy under irrelevant operations, we simulate noise by playing online videos in another browser, using a simple setting of 15 website classifications. As a result, we find that it causes cache pollution that lowers classification accuracy from 98.03% to 94.4% on A76. Cache pollution also affects the accuracy of timing-based attacks, while a naive timing-based iPrime+iProbe attack under the same settings on A76 only results in 55% accuracy.

6 Case Study 2: Timer-Free Kernel leakage Attack

In this section, we show how we use GhostCache to attack the secret-dependent control flow and the IC gadget, bypassing the user-kernel boundary.

6.1 Threat Model

The attacker aims to track the execution of a specific control flow or conditional function call within a victim process or kernel context to extract sensitive data or enable further exploits.

For Control Flow Leakage. We assume that the attacker and the victim are on the same logical core, the same as standard user-kernel leakage model [10, 76], thus sharing the L1I$. Only user-level access and system call capabilities are required—no superuser privileges or direct communication.

For Spectre Attack. This work adopts the user-to-kernel Spectre threat model [37, 71, 79], where an adversary controls an unprivileged user process to leak kernel data. High-resolution timers are assumed disabled as a mitigation, and alternative timer methods are considered too noisy or limited for effective use. Following prior work [37, 79], the presence of an exploitable instruction gadget in the kernel is assumed or discovered by the attacker.

6.2 Leak Kernel Control-Flow

Kernel control-flow leakage serves as the first step on the road to conduct a user-kernel Spectre attack, which shows that we can do cross-boundary leakage. The next step is to obtain a secret using the Spectre gadget and transmit it via control flow, which we will introduce later.

Following the methodology of prior work [10], we implement a custom kernel function embedded within a user-accessible system call, incorporating a switch statement as shown in Listing 2. This function contains 16 unique switch cases, each determined by a secret value and invoking a function aligned with a distinct cache set. The attacker utilizes this system call and employs GhostCache to infer the secret value by monitoring cache eviction patterns. Our evaluation is conducted on a Cortex-A76 device running Ubuntu 24.04.1 LTS (6.8.0-1013-raspi). Across 4096 iterations, the victim function updates the secret value randomly after each system call. Using GhostCache, the attacker identifies the evicted cache set to deduce the secret value, achieving a leakage accuracy of 96.729%.

```
1  void vulnerable_syscall(void* param) {
2    int value = get_secret_state();
3    switch (value) {
4    case STATE_ONE: do_something(); break;
5    case STATE_TWO: do_other(); break;
6    default: do_another(); } }
```

Listing 2: The customized kernel function leaking control flow using GhostCache.

6.3 Use Function Pointers as Spectre Gadgets

Next, we examine Spectre attacks. Within the cache domain, there are two types of Spectre disclosure gadgets [71]: Instruction Cache-signal gadgets (IC gadgets) and Data Cache-signal gadgets (DC gadgets). An IC gadget involves two dependent memory operations: the first retrieves the secret, and the second executes secret-dependent control flow, transmitting the secret to L1I$. Examples of IC gadgets are shown in Listing 3. The first, an unmask gadget from `posix_test_lock` (a real Linux Kernel case), is `(*func)()`. The second, a mask gadget, demonstrates a common proof-of-concept. The transient execution window (several-hundred-cycle cache misses) allows sub-speculative execution (around 10 cycles) to resteer from BTB entries to secret-encoded function calls [65].

```
1  void posix_test_lock(...) { // fs/locks.c
2    ... (*func)(); ... }
3  void conditional_call_function(size_t x) {
4    if (x < array1_size) {
5      uint8_t secret = array1[x];
6      func[(secret && 0xff) * offset](); } }
```

Listing 3: Two types of Spectre IC gadgets.

In this work, we focus on exposing IC gadgets and utilize GhostCache to transmit secrets in Spectre attacks. As listed in Listing 3, following established practices [10, 37, 60], we inject a SpectreRSB [6, 39, 71] misprediction gadget [7] into a user-accessible system call. This misprediction gadget triggers a misprediction, retrieves the secret address, and uses it as the value of `*func` to speculatively execute the disclosure gadget `(*func)()`. During transient execution, the secret is loaded into a register and used as a target jump address. Prior research [32] demonstrates that attackers can manipulate this process to make the secret part of a valid transient address. The final secret-dependent control flow (jump) transmits the secret's value to the eviction set of L1I$. Finally, the attacker employs GhostCache to retrieve the transmitted secret.

For a real-world end-to-end attack, we can use SpectreBHB [6] for branch target injection, use ROP-gadgets [8] for valid function pointers, then identify and execute kernel disclosure gadgets with the stated success rate.

The attacker leverages the vulnerable system call to trigger the above transient execution attack, as we show in Figure 14. Finally, through cache state observation using GhostCache, the attacker manipulates cache evictions to infer secret data without explicit timers. The noise can be filled using the differential method (S2.3) introduced in section 3.

6.4 Spectre Disclosure Gadget Evaluation

Using the Spectre unmask IC gadget, in ARM's Cortex-A76, our GhostCache achieves 90.840% accuracy when leaking 1024 random secrets from kernel address space. The value of secret ranges from 0 to 255, with 256 possible values. The error rate comes from whether the speculative execution can be triggered and whether the transient windows are enough to finish the transmission. Lastly, we perform a real-world exploit on the Spectre IC gadgets discovered in the Linux kernel (the gadget in the `posix_test_lock` function

[7] Identifying misprediction gadgets in the kernel is outside the scope of this work.

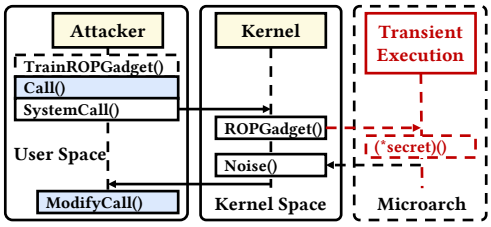

Figure 14: Kernel leakage procedure that exploits the IC gadget we proposed.

is listed in Listing 4), with an 82.910% success rate in recovering 1024 bits from kernel address space.

```
1  0xffffa000844bf3cc: ldr x1,[x1,#8]
2  0xffffa000844bf3d0: cbz x1,0xffffa000844bf3dc
3  0xffffa000844bf3d4: mov x0,sp
4  0xffffa000844bf3d8: blr x1
```

Listing 4: The real Linux Kernel IC gadget we exploited.

7 Case Study 3: RSA Leakage

In this case study, we investigate a vulnerability in RSA implementation of the widely used MbedTLS library [64].

7.1 Threat Model

In this attack scenario, we follow the assumptions of previous work [10]. The adversary is assumed to execute an unprivileged program on the same physical core as the victim. The victim thread is an RSA decryption routine using a cryptographic library such as MbedTLS [64]. The attacker is able to attack the scheduler of the operating system [29] by injecting a malicious context switch, thus synchronizing with the execution of the victim's secret-dependent branch, with the goal of leaking confidential information from the victim.

7.2 Attack the Vulnerable Code Gadget

```
1  // Sliding-window exponentiation: X = A^E mod N ...
2  int mbedtls_mpi_exp_mod(/* ... */) { // ...
3      // ...  Secret-dependent branch
4      if (ei == 0 && state == 1) {
5        MBEDTLS_MPI_CHK(mpi_select(&WW, W,
         w_table_used_size, x_index));
6        mpi_montmul(&W[x_index], &WW, N, mm, &T);
7        continue; } /* ... */ }
```

Listing 5: The RSA implementation of MbedTLS.

A real-world code segment from MbedTLS (v3.5.2) [64] demonstrates a branch whose execution depends on the value of `ei`, as we can see from line 4 to line 6 of Listing 5, which is derived from the private key during the modular exponentiation process (ei corresponds to the E in RSA encryption operation: $c = m^E \bmod N$). The execution target such as `mpi_montmul` depends on the secret key bits, making the function vulnerable to control-flow leakage via the L1 instruction cache, thus we can achieve RSA leakage.

The attacker monitors the cache status before and after the secret-dependent branch, and observes whether the cache set that corresponds to `mpi_select` or `mpi_montmul` is evicted, and further

infers the value of `ei`. Specifically, after preparing the L1I$, the attacker thread calls a `sched_yield` system call to give the CPU to the victim thread to run the `exp_mod` function. To avoid trivial engineering configurations, we simplify the code snippet of the victim and focus on the key secret-dependent branch. Finally, using GhostCache, without relying on any timer or shared memory, we attack the RSA successfully.

7.3 Evaluation Result

In our experiments on Cortex A76, it took no more than 1 second to leak a 1-bit secret key in the RSA leakage attack. We repeated the experiment 10 times, each time the attacker leaked a 32-bit secret from the victim and the average accuracy was 91.876% with a standard deviation of 5.625. Finally, by monitoring the instruction trace to identify patterns in secret-dependent control flow and using multiple-round attacks, we are able to reconstruct the full key.

8 Discussion

In this section, we propose potential countermeasures and discuss the generality of GhostCache.

8.1 Potential Countermeasures

Hardware defenses. Implementing hardware-enforced coherence protocols to synchronize L1 instruction and data caches in ARM and RISC-V can mitigate GhostCache, eliminating incoherent data. However, this approach may introduce hardware complexity.

Operating System Defenses. Operating systems can mitigate GhostCache by ensuring no observable cache side effects during transitions such as process switches or system calls. This involves using coherence maintenance instructions, such as invalidating or cleaning cache entries, to obscure cache states containing sensitive information. However, sanitizing the instruction cache may incur large performance overhead due to increased cache misses.

ARM defenses. On ARM systems where barriers enforce coherence (`CTR_EL0.DIC == 0` or `CTR_EL0.IDC == 0`), `IC IALLU` can invalidate all instruction cache entries, ensuring modified data is reflected in the cache and reducing stale instruction risks.

RISC-V defenses. For RISC-V systems with weak cache coherence, using `FENCE.I` during process switches or privilege transitions enforces synchronization between instruction and data caches.

Using CoreMark [19], we evaluate the performance loss of mitigation that uses `FENCE.I` on SiFive P550, inserting it before each iteration to clean the L1I$ trace, resulting in a 9% benchmark slowdown. This exceeds a simple downgrade from SonicBoomV3 (2020, CoreMark/MHz: 6.2) to AWS Graviton (2018, CoreMark/MHz: 5.8), which incurs a 6.9% loss.

Software-Application-Level Defenses. *Mitigating Spectre vulnerabilities.* Kernel and software developers can use speculation barriers, memory fencing, or restricting speculative execution on sensitive code to rework or remove speculative execution paths

Constant-time programming. Avoid secret-dependent control flow by ensuring execution paths do not depend on sensitive data, as this can create side-channel vulnerabilities. Adopting constant-time programming techniques and minimizing secret-influenced branches can help mitigate such risks, though these measures may largely increase development complexity.

8.2 GhostCache Attack on other Chips or ISAs

GhostCache is an ISA-agnostic primitive to reveal the microarchitectural state. The root cause is the weak-coherence-based implementation of L1$, which means that the software needs to manually maintain the coherence. Since the use of a weak-coherence-based L1$ can reduce hardware complexity and overhead, most RISC-based chips such as Cortex A76 (ARM), SiFive P550 (RISC-V), and even the high-performance Apple M4 core currently choose to adopt this design.

The ARM ISA [47] allows "Software instruction cache maintenance", and the RISC-V ISA [69] allows "incoherent instruction cache". For processors from other ISAs or other manufacturers, as long as they also use similar caches that require manual maintenance by software, the same security issue will arise.

9 Related Work

In this section, we provide a systematic analysis of state-of-the-art cache state observation techniques for side-channel attacks and create a taxonomy (see Table 4), including timer-based, counter-based, amplification-based, and timer-and-counter-free techniques. To the best of our knowledge, this is the first systematic analysis of cache monitoring methods.

9.1 Taxonomy of Cache State Observation Techniques

Timer-based method: Observing cache state changes is key to cache side-channel attacks (SCAs). Most attacks rely on HRT [26], such as `RDTSC` (x86-64), `PMCCNTR` (ARM), and `RDCYCLE` (RISC-V). These timers measure memory access latencies to infer cache states. However, hardware timers in ARM and RISC-V often have strict restrictions [18, 42], and OS-level syscalls like `clock_gettime` can be easily limited for security.

Counter-based method: PMU counters provide accurate cache state monitoring, offering events like L1 hits or misses, but they are often restricted to privileged modes [21, 66]. User-level attackers design implicit timers using sibling threads [42, 59], requiring uninterrupted concurrent execution and shared memory. These methods are susceptible to noise and instability, limiting their reliability.

Amplification-based method: Methods like Hacky Racers [72] use out-of-order execution to amplify timing difference for coarse-grained timers. By leveraging sophisticated manipulation of pLRU-state [35, 55] or cache side-channel topologies [34], the latency of a single cache miss can also be amplified by several orders of magnitude. However, this amplification significantly increases the access time, making each operation much slower than normal cache access. Moreover, since the method relies on additional resources such as eviction sets to perform cache attacks, the overall monitoring capacity is reduced.

Timer-Free and Counter-Free method: Timer-free and counter-free attacks exploit specific instructions (e.g., `MWAIT`, TSX) or vendor-specific implementations (e.g., NVIDIA, Apple). The first timer-free cache attack, Cache Storage Channels (CSC) [27], exploits virtual aliases to create intentional incoherence, similar to our

[8] We leave the exploration of its prefetch queue to future work.

Table 4: A taxonomy of the cache side-channel attack observation techniques, with a notation system that addresses limitations such as N for suffering from noise, M for the one that can be mitigated or disabled using corresponding ways, S for too slow, P for privilege required, R for resolution limited, V for vendor limited, A for ability constraints.

Taxo.	Exploit Feat.	x86-64	ARM	RISC-V	Vendor	Typical Attack Example(s)	Limitation
Timer-based	RDTSC	✓	✗	✗	Intel, AMD	Flush+Reload/Flush [25, 74], Evict+Reload [26], Prime+Probe [52, 58], Prime+Scope [56]	N (Frequency), V
	PMCCNTR	✗	✓	✗	ARM	Evict+Reload (Armageddon) [42], Prime+Probe (TruSpy) [38], PRIME+COUNT [11]	P (Privileged)
	RDCYCLE	✗	✗	✓	SiFive, etc	Cache+Time, Flush+Fault [22]	M (mcounteren)
	clock_gettime	N/A	✓	✓	N/A	Evict+Time, Prime+Probe, Evict+Reload [15]	R
Counter-based	counter-thread	N/A	✓	✓	N/A	Armageddon [42]	N (Schedule), R
	PMU	✓	✓	✓	Intel, AMD	Exploiting HPC [66], CounterSEVeillance [21]	P (Privileged)
Amplif.-based	Parallelism-based	✓	N/A	N/A	Intel, AMD	Hacky Racers [72]	N, S
	Cache Gate/pLRU-based	✓	✓	N/A	Intel, Apple	Gates of Time [34], ShowTime [55], iLeakage [35]	S
Free from Timer and Counter	Cache Storage	✗	✓	N/A	ARM	Cache Storage Channels [27]	P (Privilege)
	TSX	✓	✗	✗	Intel only	Prime+Abort [17], DPrime+Dabort [36]	V (Intel only)
	MWAIT Inst.	✓	✗	N/A	Intel, AMD	(M)WAIT for it [79]	A (Only monitors write)
	LL/SC Inst.	✗	✓	N/A	Apple only	Synchronization Storage Channels (S^2C) [77]	A (Up to 11 sets)
	Invalid Inst.	✗	✗	✗	NVIDIA	INVALIDATE+COMPARE [81]	V (NVIDIA GPU only)
	Weak Coherence	N/A [8]	✓	✓	ARM, SiFive, etc.	**GhostCache (This work)**	

Modify+Recall. However, CSC assumes an attacker (untrusted kernel) running as privileged software outside TrustZone, while we target a weaker userspace attacker. CSC also relies on a custom scheduler and excludes general-purpose OS attacks, whereas we demonstrate our attack in real-world OS environments and identify a new functional pointer Spectre gadget. Additionally, we propose new primitive Call+ModifyCall to mitigate interference from prefetchers and bypass macro-op caches. Prime+Abort [17] and DPrime+DAbort [36] rely on Intel TSX, limiting their applicability. Other methods, such as INVALIDATE+COMPARE [81], target NVIDIA GPUs, exploiting GPU-specific cache behavior.

To summarize, existing timer-free and counter-Free methods are constrained by specific instructions or vendor implementations, limiting adaptability and effectiveness. Some approaches, like S^2C [77], monitor only a limited number of cache sets (up to 11), further restricting their scope.

9.2 Cache Side-Channel Attack on Emerging Architectures: RISC-V and ARM

Early RISC-V security research focused on simulated or prototype cores. Gonzalez et al. [23] simulated Spectre on BOOM cores in FireSim, while Mata et al. [51] and Ahmadi et al. [4] demonstrated Flush+Reload and Prime+Probe attacks on FPGA-based RISC-V systems. Fuchs et al. [20] developed a transient attack test suite for RISC-V. Research on commercial chips remains limited; Gerlach et al. [22] identified three timer-based attacks, including Cache+Time and Flush+Reload, on commercial RISC-V CPUs.

ARM processors have seen extensive cache attack studies. AutoLock [24] (2017) examines timing attack defenses on ARM but shows bypass techniques. ARMageddon [42] (2016) demonstrates cross-core timing attacks using multiple techniques. TruSpy [78] (2016) exploits TrustZone cache contention with Prime+Probe, while Zhang et al. [80] (2016) and Lee et al. [40] (2021) focus on Flush+Reload in ARM processors. Deng et al. [15] provide comparative ARM cache attack evaluations, and iTimed [31] (2021) targets Apple A10 Fusion. Recent work, S^2C (2023), demonstrates a timer-free attack exploiting Apple's LL-SC implementation.

10 Conclusion

In this paper, we introduce GhostCache, a novel timer-free cache attack. By leveraging self-modifying code to disrupt coherence between L1I and L1D caches, GhostCache establishes two new side-channel attack primitives that detect L1I cache evictions without the need of timing mechanisms. Our three case studies show that this method achieves competitive performance compared to state-of-the-art Prime+Probe attacks while operating effectively on both ARM and RISC-V chips.

Acknowledgments

This work was generously supported by NSFC (No. U24A6009, 62361166633), the National Key Research and Development Program of China under Grant (2024YFB4405402), Beijing Municipal Science and Technology Project (Nos.Z241100004224028), Beijing Natural Science Foundation (L247013, 4242026), BNRist. We thank the anonymous reviewers and our shepherd for their valuable feedback. We further would like to thank XiangShan security team (Yungang Bao, Miaomiao Yuan, etc.), for helping us analyze GhostCache on XiangShan and discuss the mitigation.

Open Science

The artifacts implementing proof-of-concept of GhostCache are publicly available at https://doi.org/10.5281/zenodo.15559504.

References

[1] Onur Aciiçmez. 2007. Yet another MicroArchitectural Attack: exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (Fairfax, Virginia, USA) *(CSAW '07)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/1314466.1314469

[2] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems* (Santa Barbara, CA) *(CHES'10)*. Springer-Verlag, Berlin, Heidelberg, 110–124.

[3] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-Driven Cache Attacks on AES (short paper). In *International Conference on Information and Communications Security*. 112–121.

[4] Mahya Morid Ahmadi, Faiq Khalid, and Muhammad Shafique. 2021. Side-Channel Attacks on RISC-V Processors: Current Progress, Challenges, and Opportunities. *CoRR* abs/2106.08877 (2021). arXiv:2106.08877 https://arxiv.org/abs/2106.08877

[5] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, , and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 971–988.

[7] Daniel J Bernstein. 2005. Cache-Timing Attacks on AES. (2005).

[8] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of ROP Chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 1–16.

[9] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 201–215.

[10] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2023. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 16–32. https://doi.org/10.1145/3575693.3575719

[11] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3274694.3274704

[12] Intel Corporation. 2024. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[13] Joan Daemen and Vincent Rijmen. 1999. AES Proposal: Rijndael. (1999).

[14] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends . In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 53–66. https://doi.org/10.1109/HPCA53966.2022.00013

[15] Shuwen Deng, Nikolay Matyunin, Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. 2022. Evaluation of Cache Attacks on Arm Processors and Secure Caches . *IEEE Trans. Comput.* 71, 09 (Sept. 2022), 2248–2262. https://doi.org/10.1109/TC.2021.3126150

[16] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2020. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 683–697. https://doi.org/10.1145/3373376.3378510

[17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 51–67.

[18] Ben Dooks. 2024. RFC: extern illegal instruction trap and trap RDCYCLE - Ben Dooks. https://lore.kernel.org/linux-riscv/20240917130853.18657-1-ben.dooks@codethink.co.uk/ Last accessed 28 September 2024.

[19] EEMBC. 2009. CoreMark. https://github.com/eembc/coremark

[20] Franz A Fuchs, Jonathan Woodruff, Simon W Moore, Peter G Neumann, and RN Watson. 2021. Developing a test suite for transient-execution attacks on RISC-V and CHERI-RISC-V. In *Workshop on Computer Architecture Research with RISC-V*.

[21] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *Network and Distributed System Security Symposium 2025: NDSS 2025*.

[22] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2321–2338. https://doi.org/10.1109/SP46215.2023.10179399

[23] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanovic. 2019. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[24] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1075–1091.

[25] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 279–299.

[26] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium (USENIX)*. 897–912.

[27] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *Symposium on Security and Privacy (S&P)*. 38–55.

[28] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

[29] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

[30] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1458–1473. https://doi.org/10.1109/SP46214.2022.9833692

[31] Gregor Haas, Seetal Potluri, and Aydin Aysu. 2021. iTimed: Cache Attacks on the Apple A10 Fusion SoC. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 80–90. https://doi.org/10.1109/HOST49136.2021.9702290

[32] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. 2024. Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 3773–3788. https://doi.org/10.1109/SP54263.2024.00158

[33] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. 2022. HyperDbg: Reinventing Hardware-Assisted Debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1709–1723. https://doi.org/10.1145/3548606.3560649

[34] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1955–1972.

[35] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2038–2052. https://doi.org/10.1145/3576915.3616611

[36] Sowoong Kim, Myeonggyun Han, and Woongki Baek. 2022. DPrime+DAbort: A High-Precision and Timer-Free Directory-Based Side-Channel Attack in Non-Inclusive Cache Hierarchies using Intel TSX. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 67–81. https://doi.org/10.1109/HPCA53966.2022.00014

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[38] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 463–480.

[39] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[40] Heemin Lee, Sungyeong Jang, Han-Yee Kim, and Taeweon Suh. 2021. Hardware-Based FLUSH+RELOAD Attack on Armv8 System via ACP. In *2021 International Conference on Information Networking (ICOIN)*. 32–35. https://doi.org/10.1109/ICOIN50884.2021.9334005

[41] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 643–660.

[42] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564.

[43] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 813–825.

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

[46] Markus Löning, Anthony J. Bagnall, Sajaysurya Ganesh, Viktor Kazakov, Jason Lines, and Franz J. Király. 2019. sktime: A Unified Interface for Machine Learning with Time Series. *CoRR* abs/1909.07872 (2019). arXiv:1909.07872 http://arxiv.org/abs/1909.07872

[47] ARM Ltd. 2024. *ARM Architecture Reference Manual.* Available at https://developer.arm.com/documentation.

[48] Arm Ltd. 2024. Cortex-A76. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a76

[49] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. 2025. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In *Network and Distributed System Security Symposium (NDSS) 2025.* https://doi.org/10.14722/ndss.2025.240223 Network and Distributed System Security Symposium 2025 : NDSS 2025, NDSS 2025 ; Conference date: 23-02-2025 Through 28-02-2025.

[50] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 118–129. https://doi.org/10.1109/ISCA.2012.6237011

[51] Prashant Mata and Nanditha Rao. 2021. Flush-Reload Attack and its Mitigation on an FPGA Based Compressed Cache Design. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 535–541. https://doi.org/10.1109/ISQED51717.2021.9424252

[52] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[53] Colin Percival. 2005. Cache Missing for Fun and Profit.

[54] The Chromium Project. 2024. Chrome Browser. https://www.chromium.org/chromium-projects/ Accessed on 2015-02-10.

[55] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. 2023. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3579856.3590332

[56] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2906–2920. https://doi.org/10.1145/3460120.3484816

[57] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 361–374. https://doi.org/10.1109/ISCA52012.2021.00036

[58] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. 2021. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1077–1090. https://doi.org/10.1145/3445814.3446742

[59] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2023. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3–7, 2017, Revised Selected Papers* (Sliema, Malta). Springer-Verlag, Berlin, Heidelberg, 247–267. https://doi.org/10.1007/978-3-319-70972-7_13

[60] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *European Symposium on Research in Computer Security (ESORICS)*. 279–299.

[61] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2021. Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2021), 2042–2060. https://doi.org/10.1109/TDSC.2020.2988369

[62] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 639–656.

[63] RISC-V Cache Management Operation Task Group (CMO TG). 2022. Cache Management Operations for RISC-V. https://github.com/riscv/riscv-CMOs

[64] Mbed TLS. 2024. Mbed TLS. https://github.com/Mbed-TLS/mbedtls/tree/v3.5.2.

[65] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7303–7320.

[66] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. 2008. Exploiting Hardware Performance Counters. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. 59–67. https://doi.org/10.1109/FDTC.2008.19

[67] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[68] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. Association for Computing Machinery, New York, NY, USA, 41–46.

[69] Andrew Waterman, Krste Asanovic, et al. 2019. The risc-v instruction set manual volume i: Unprivileged isa. *Document Version* 20191213 (2019), 1–4.

[70] Andrew Waterman, Krste Asanovic, and John Hauser. 2021. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203.*

[71] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 89–89. https://doi.org/10.1109/SP61157.2025.00089

[72] Haocheng Xiao and Sam Ainsworth. 2023. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 354–369. https://doi.org/10.1145/3575693.3575700

[73] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199. https://doi.org/10.1109/MICRO56248.2022.00080

[74] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732.

[75] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[76] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. 2024. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 770–784. https://doi.org/10.1145/3620666.3651382

[77] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. 2023. Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1973–1990.

[78] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Paper 2016/980. https://eprint.iacr.org/2016/980

[79] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7267–7284.

[80] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 858–870. https://doi.org/10.1145/2976749.2978360

[81] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2101–2118.

[82] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).

[83] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 582–600. https://doi.org/10.1145/3620665.3640403