Spectre on RISC-V Silicon: Attacks and Defenses on Commercial Out-of-Order Processors

Lukas Gerlach[1] Marton Bognar[2] Daniel Weber[1] Michael Schwarz[1] Jo Van Bulck[2]

[1]*CISPA Helmholtz Center for Information Security* [2]*DistriNet, KU Leuven*

Abstract

Speculative execution attacks have been extensively studied on mainstream x86 and ARM architectures. However, on RISC-V, research has mostly concentrated on open-source academic designs. Commercially available RISC-V silicon is widely perceived as too simple to be vulnerable, and as a result, no end-to-end attacks have been demonstrated on real hardware to date and essential software such as the Linux kernel remains unmitigated.

In this paper, we challenge that assumption. We systematically assess all commercially available out-of-order RISC-V processors (SiFive P550 and T-Head Xuantie C910/C920), finding them vulnerable to a range of Spectre attacks, and demonstrate the first Spectre attack leaking arbitrary kernel memory on real RISC-V hardware. Concerningly, our analysis reveals that mitigations in compilers, operating systems, and applications remain largely absent, and that the RISC-V instruction set lacks a dedicated speculation barrier. As a stopgap solution, we empirically characterize which instructions can halt speculation on commercial processors. We additionally audit the Linux kernel for Spectre gadgets and contribute patches, several of which have been accepted upstream. Finally, we evaluate and benchmark software-based Spectre mitigations and derive recommendations for the evolving RISC-V ecosystem, laying the groundwork for securing real hardware as it enters security-critical deployments.

1 Introduction

In 2018, Spectre attacks [55] revealed that speculative execution in modern processors leaves microarchitectural side effects that allow attackers to leak sensitive information. After the initial discovery, extensive research on Spectre attacks was conducted, leading to the identification of multiple variants [1, 9, 15, 39, 56, 70, 103]. Similarly, mitigations against these attacks were developed, ranging from hardware-based [41, 52, 53, 109, 112] to software-based approaches [10, 17, 50, 95, 115]. However, this research has primarily focused on x86 and ARM [10], leaving the emerging RISC-V ecosystem underexplored.

Recently, academic out-of-order RISC-V processors have emerged as platforms for speculative execution research. Openly accessible designs like BOOM [18] allow researchers to modify hardware directly, making RISC-V softcores a primary platform for hardware-level mitigation research [2, 6, 14, 20, 23, 25, 45, 46, 71, 106, 111, 116]. As a byproduct, proof-of-concept attacks demonstrating various Spectre variants on these processors exist [27, 33, 46, 59]. However, this analysis has remained confined to research-grade processors.

In the immediate aftermath of Spectre and Meltdown in 2018, both the RISC-V Foundation [81] and major vendors [88] explicitly asserted that no current or announced RISC-V silicon was susceptible at the time. Consequently, major infrastructure, including the Linux kernel, did not deploy mitigations. However, with the commercial availability of high-performance out-of-order RISC-V processors from SiFive and T-Head, it becomes questionable whether RISC-V silicon is still unaffected by Spectre. These concerns are especially pressing as RISC-V processors enter cloud environments [85] that require multi-tenant isolation. To date, no practical Spectre attacks have been demonstrated on commercially available RISC-V processors. This raises a critical question: *Are commercial RISC-V processors vulnerable to Spectre attacks, and is the software ecosystem prepared with adequate mitigations?*

In this paper, we present the first comprehensive study of Spectre attacks and mitigations on commercial RISC-V silicon and production-grade software. We systematize existing Spectre attack research on RISC-V and find that it is fragmented and mostly targets research processors. To address this gap, we perform a systematic analysis of Spectre attacks on the SiFive P550 and the Xuantie C910/C920, the only commercially available out-of-order RISC-V processors available to us at the time of writing. Our findings indicate that, despite earlier claims [81, 88], today's performant RISC-V silicon is vulnerable to the same Spectre attacks as high-end x86 and ARM processors. We demonstrate proof-of-concept attacks

on both processors using Spectre-PHT, Spectre-BTB, Spectre-RSB, and Spectre-STL, achieving up to 100 % recall with more than 97 % precision. To demonstrate practical impact, we develop a Spectre exploit leaking arbitrary Linux kernel memory on the C910 at 338 B/s.

Given these silicon vulnerabilities, we investigate the adoption of mitigations in the RISC-V ecosystem and find them lacking across the board. At the instruction-set level, RISC-V lacks dedicated speculation-control features available on x86 and ARM, including speculation barriers and predictor management mechanisms such as Indirect Branch Restricted Speculation (IBRS), Indirect Branch Prediction Barrier (IBPB), and Single Thread Indirect Branch Predictor (STIBP). At the compiler level, only prototype implementations exist. Retpolines, which mitigate Spectre attacks on indirect branches, fail in complex programs due to calling convention differences. At the kernel level, mitigations remain unimplemented: Linux's `barrier_nospec()` macro compiles to a no-op. Similarly, the BPF[1] just-in-time (JIT) compiler emits no RISC-V instructions for the critical `BPF_NOSPEC` speculation barriers inserted by the BPF verifier, leaving JIT-compiled BPF programs vulnerable despite passing verification.

To provide a path forward, we systematically evaluate software mitigations on commercial RISC-V silicon. We test which instructions halt speculation on each processor and find that effective barriers vary across implementations. We characterize microarchitectural details such as Return Stack Buffer (RSB) fallback behavior and find that neither processor falls back to the Branch Target Buffer (BTB) on RSB underflow, meaning retpolines can be effective. We benchmark compiler-based mitigations and find that Speculative Load Hardening (SLH) incurs a 1.34× slowdown, while speculation-barrier insertion using `rdtime` achieves a 2.69× slowdown, lower than on x86 (2.5× and 4×, respectively) [98, 115]. Since RISC-V retpolines require tight coupling with function prologue structure that existing compilers do not reliably produce, we propose branchless JIT dispatch as an alternative that relies on neither speculation barriers nor return stack manipulation. Based on these findings, we recommend SLH for userspace protection, processor-specific speculation barriers for kernel hardening, and standardization of a dedicated speculation barrier in the RISC-V instruction set. Since existing binary-level gadget scanners do not support RISC-V, we develop a complementary mitigation-diffing approach that identifies missing protections by comparing hardened code paths on x86 and ARM with their RISC-V equivalents. This reveals 5 vulnerabilities in BPF JIT, syscall table indexing, user copy functions, futex operations, and KVM interrupt handling. We developed patches addressing these gaps; several have been merged into the Linux kernel, with the remainder in ongoing collaboration with maintainers.

[1] In this paper, we use "BPF" to refer to eBPF instead of classic BPF.

Contributions. In summary, our main contributions are:

- We demonstrate, for the first time, all major Spectre variants (PHT, BTB, RSB, STL) on commercial silicon processors (SiFive P550, Xuantie C910/C920) using a unified framework covering 13 attack configurations.
- We demonstrate a Spectre exploit on RISC-V hardware that leaks kernel memory via BPF.
- We audit Spectre mitigations across the RISC-V stack (hardware, compilers, kernel), identify critical gaps, and contribute patches to the Linux kernel.
- We characterize which instructions halt speculation and benchmark software mitigations on commercial silicon.

Responsible Disclosure. We disclosed our findings to the Linux kernel security team on December 3, 2025. The uaccess pointer masking, syscall table, and KVM patches have been merged into mainline Linux, and we are collaborating with maintainers on the remaining patches. We also contacted SiFive and T-Head Xuantie regarding processor-specific findings and they adressed our findings by providing ad-hoc speculation barriers.

2 Background

We introduce the RISC-V instruction set architecture (ISA), speculative execution, and Spectre attacks.

2.1 RISC-V

RISC-V is an open, modular instruction set architecture implemented by multiple independent vendors, each making their own microarchitectural design choices [82, 99]. The specification allows a wide range of implementations, from simple in-order processors to superscalar out-of-order designs. Multiple commercial vendors now ship out-of-order processors (e.g., SiFive P550, T-Head Xuantie C910, and C920).

Beyond the core ISA, the RISC-V extension model permits both optional standard extensions and vendor-specific custom instructions. From a security perspective, extensions can be a double-edged sword: they may enable attacks (e.g., unprivileged cache flushes) or introduce vulnerabilities (e.g., GhostWrite) [29, 93] but could also provide mitigations such as speculation barriers (e.g., the proposed `fence.t` extension [106]). However, custom extensions and their semantics are often undocumented, complicating security analysis.

2.2 Speculative Execution and Spectre Attacks

Out-of-order processors track in-flight instructions in a reorder buffer (ROB). The ROB also limits the number of instructions that can execute speculatively before resolution (we measure the resulting speculation windows in Appendix B).

Speculative execution predicts control and data flow so instructions can run before outcomes are known; correct predictions commit, while mispredictions roll back. Spectre [55] exploits that mispredicted (transient) instructions leave microarchitectural traces before rollback. These traces can be recovered via covert channels: most commonly caches [63, 110], but also execution-port contention [11, 34], or remote timings [87]. Because these effects bypass architectural checks, they undermine isolation between processes or privilege levels. Prior work demonstrated leakage across process boundaries, across sandbox transitions, and from privileged code when suitable gadgets exist [55].

Since the initial disclosure, multiple Spectre variants have been identified [40, 41, 108]. Canella et al. [15] classify Spectre attacks by the exploited predictor. We follow this categorization and summarize the most relevant Spectre variants:

Spectre-PHT (v1) mistrains the Pattern History Table (PHT), allowing controlled mispredictions of direct branches [55]. When targeting array bounds checks, mispredictions allow transient out-of-bounds loads whose results are encoded in the microarchitectural state [55].

Spectre-BTB (v2) poisons the Branch Target Buffer (BTB) to redirect transient control flow to mispredicted targets of indirect branches [55]. Branches in the BTB are indexed by subsets of address bits (and, for indirect branches, often combined with branch history), so colliding entries may let an adversary redirect speculation to chosen code pages. Unlike PHT-style bounds-bypass, BTB poisoning enables ROP-like chaining of existing gadgets in transient execution, constructing covert channels from architecturally unreachable code. Barberis et al. [9] demonstrated that even with BTB isolation between user- and kernelspace, non-isolated prediction structures (concretely, the branch history influencing BTB tag computation) can be exploited to bypass this isolation.

Spectre-RSB (ret2spec) manipulates the Return Stack Buffer (RSB) tracking recent return addresses [56, 70]. When the RSB diverges from the architectural call stack (e.g., due to underflow, cross-domain switches, or transiently executed calls), the processor speculatively returns to attacker-selected gadgets. On underflow, predictions may also fall back to other structures such as the BTB [56].

Spectre-STL (v4, -SSB) exploits mispredicted store-to-load forwarding, exposing stale data transiently [1]. When a load is (incorrectly) deemed independent of a prior store, it may read the old value before the true dependency resolves. As stale load values may contain confidential information, this can lead to information disclosure.

Table 1: Demonstrated Spectre variants on RISC-V in prior work. So far, only research prototypes were analyzed.

Target Processor	PHT	BTB	RSB	STL	References
BOOM [18]	●	●	●	●	[24, 33, 42, 46, 59, 60]
RiscyOO [114]	○	●	●	●	[27]
RSD [72]	●	○	○	●	[3, 19]
Proteus [13]	●	●	○	○	[23]
NaxRiscv [90]	●	○	○	○	[2]
NutShell [76]	●	○	●	○	[42]

● demonstrated, ○ not demonstrated

3 Spectre Attacks on RISC-V Silicon

In this section, we survey prior Spectre research on RISC-V, systematically analyze Spectre variants on commercial silicon, characterize predictor behavior, and demonstrate proof-of-concept attacks across all major variants.

3.1 Prior Work and Research Gap

The original Spectre paper [55] notes that RISC-V includes indirect branches but does not analyze RISC-V processors in detail. An overview of existing Spectre proof-of-concept (PoC) implementations on RISC-V is provided in Table 1. Prior work predominantly targets FPGA-synthesizable research processors. BOOM [18] is the most studied, with demonstrations of all four variants developed incrementally across mitigation [33, 46, 71], detection [24, 42, 60], and benchmarking [59] efforts. Adjacent transient-execution work also targets BOOM and Rocket: IntroSpectre [31] provides a pre-silicon framework for vulnerability discovery, but focuses on Meltdown rather than Spectre. Other research processors received less attention: RiscyOO [114] was used to validate a CHERI-based test suite [27], RSD [72] for detection mechanisms [3] and SSB analysis [19], Proteus [13] for a mitigation for constant-time code [23], and NaxRiscv [90] for custom speculation barrier research [2]. Notably, on most processors, only a subset of variants is tested, and no work systematically analyzes all variants across multiple processors.

Critically, all tested processors are research designs; commercial silicon remains unexplored. This is a significant gap as RISC-V processors reach commercial deployment. Prior work on silicon RISC-V processors [5, 29, 47, 69] has focused exclusively on cache side channels and other microarchitectural attacks, leaving Spectre unexamined. Our work aims to fill this gap by analyzing all Spectre variants from Table 1 on commercial RISC-V CPUs.

3.2 Methodology

We assume the standard Spectre threat model: an attacker with code execution on the same machine as the victim, wanting to read architecturally inaccessible data.

Following the systematization by Canella et al. [15], we categorize Spectre attacks along two axes: *address space* (same-address vs. cross-address) and *branch location* (in-place vs. out-of-place). Same-address attacks operate within one process, while cross-address attacks cross process boundaries; in-place attacks train and exploit the same instruction, while out-of-place attacks mistrain a different instruction at a colliding predictor entry. We further classify attacks by the exploited predictor (PHT, BTB, RSB, STL) and test whether each can be controlled by an attacker, comprehensively characterizing predictor implementations and their exploitability.

We focus on out-of-order processors, though in-order processors may exhibit limited speculation [74]. We test the SiFive P550 and T-Head Xuantie C910/C920, the only commercially available out-of-order RISC-V processors at the time of writing. The C920 updates the C910's vector extension from RVV 0.7.1 to RVV 1.0, while retaining an identical core microarchitecture; we therefore treat them as a single design for security analysis. For completeness, we also test in-order processors (SiFive U74, Xuantie C906, C908), but we do not observe any leakage on these specific processors. Our methodology generalizes to other processors, and we release our framework to enable testing on future hardware.

As shown by previous work [11, 87, 100, 113], different covert channels can be used to transmit data from Spectre gadgets; we use the cache as it offers the most reliable leakage. We validate our cache primitives without speculation: Flush+Reload on C910/C920 and Evict+Reload on P550 (which lacks a flush instruction) both achieve reliable leakage. Our exploits encode full bytes (values 0–255) into the cache state using a probe array of 256 memory pages, where accessing page i encodes byte value i. We measure recall and precision by probing all 256 positions after each attack iteration. For timing, recent RISC-V Linux kernels disable `rdcycle` in userspace [66], but we find that it can be re-enabled via the `perf` utility on permissive configurations (cf. Section 6). Moreover, counter threads [86] can also be used ($\approx$6 cycle resolution on C910).

3.3 Attack Results

We present the results of our systematic Spectre attack evaluation. We develop a unified exploit framework implementing 13 distinct variants across four predictor types, summarized in Table 2. For each attack, we report success rates and discuss relevant microarchitectural observations.

Spectre-PHT. To mount a Spectre-PHT attack, the attacker must mistrain the branch predictor to mispredict a bounds check. For in-place attacks, we repeatedly execute the target branch with in-bounds indices until the predictor learns to predict "taken", then supply an out-of-bounds index. For out-of-place attacks, we use two processes: the attacker process trains a congruent branch (one that aliases in the predictor), then the victim process executes the target branch, which inherits the misprediction. To find congruent entries, we spray a long sequence of fallthrough branches to create collisions that mistrain the victim branch.

Table 2: Spectre attack recall rates for byte leakage (n=30, 95 % confidence interval). Each iteration encodes one of 256 possible byte values. BTB tested with predictor enabled.

Variant	Core	sa-ip	sa-oop	ca-ip	ca-oop
PHT	C910	93.3 %±7.9	87.1 %±4.6	75.0 %±1.3	56.9 %±9.1
	P550	99.8 %±0.1	46.2 %±13.7	18 %±6	2 %±2
BTB	C910	98.9 %±0.5	99.2 %±0.4	96.9 %±0.6	92.2 %±6.3
	P550	30.2 %±14.0	6.2 %±0.9	10.9 %±1.4	4.0 %±0.7
RSB	C910	99.5 %±0.4	99.3 %±0.6	0.0 %±0.1	99.3 %±0.4
	P550	99.9 %±0.1	99.9 %±0.1	0.0 %±0.0	99.9 %±0.2
STL	C910	99.3 %±0.4	–	–	–
	P550	99.9 %±0.1	–	–	–

sa: same-address, ca: cross-address, ip: in-place, oop: out-of-place. Precision is more than 97 % for all variants.

We test four PHT variants: same-address in-place (sa-ip), same-address out-of-place (sa-oop), cross-address in-place (ca-ip), and cross-address out-of-place (ca-oop). Both processors are vulnerable to all PHT variants. The C910/C920 achieves high recall for same-address variants (93.3 % for sa-ip, 87.1 % for sa-oop), while cross-address attacks show lower recall (75.0 % for ca-ip, 56.9 % for ca-oop). The P550 shows lower recall for cross-adress and out-of-place variants with 99.8 % for sa-ip, 46.2 % for sa-oop, 18 % for ca-ip, and 2 % for ca-oop.

Cross-address variants confirm that both processors share PHT state across address spaces. The lower cross-address success rates likely stem from the lack of simultaneous multithreading (SMT) on the tested processors: the RISC-V ISA allows SMT, but neither C910/C920 nor P550 implement it. Without SMT, mistraining requires context switches between the attacker and the victim, during which the predictor state may be partially evicted.

Spectre-BTB. To mount a Spectre-BTB attack, the attacker must mistrain the branch target buffer to redirect indirect branches to attacker-chosen gadgets. For in-place attacks, we repeatedly execute the target indirect branch until it learns our chosen target. For out-of-place attacks, we train a congruent indirect branch in attacker code that aliases in the BTB.

We test all BTB variants on all processors with BTB enabled. Default firmware configurations vary: the C910 and P550 ship with BTB disabled, while the C920 enables it by default. We enable BTB to characterize the full attack surface. With BTB enabled, the C910/C920 achieves 98.9 % for sa-ip, 99.2 % for sa-oop, 96.9 % for ca-ip, and 92.2 % for ca-oop. The P550 achieves 30.2 % for sa-ip, 6.2 % for sa-oop, 10.9 % for ca-ip, and 4.0 % for ca-oop. The P550 BTB is larger, with

a more complex and undocumented update policy and a more complex input hash than the C910. However the precision of the attack stays high on the P550, so an attacker can repeat the attack to compensate for the lower recall.

Both the C910/C920 and P550 implement a user/kernel BTB split that tags entries with privilege level, preventing cross-privilege poisoning. We validated this in T-Head Xuantie's open-source C910 Verilog code [91]; during our disclosure, SiFive confirmed the same for the P550. Same-privilege attacks remain viable, however, including cross-process attacks within userspace and intra-kernel attacks from BPF programs.

Spectre-RSB. Spectre-RSB attacks exploit the return stack buffer, which predicts return addresses based on prior calls. We measure RSB size by JIT-compiling nested call chains at increasing depths (Figure 6 in Appendix A). Consistent with other microarchitectural analyses [57], we find that the C910/C920 implements a 12-entry RSB, while the P550 has a 16-entry RSB.

We test all RSB variants on both processors. Out-of-place and same-address in-place variants achieve more than 99 % recall on both processors. For sa-ip on the C910/C920, we use userspace coroutines that context-switch via an indirect jump register (not classified as a return) to avoid RSB pollution. Cross-address in-place fails on both processors (0 %): the kernel transition flushes the RSB, and privilege-mode tagging prevents cross-domain speculation.

We also test RSB fallback behavior, which is critical for retpoline security (Section 5.2). When the RSB is exhausted by deep call chains, processors may stall, fall back to the BTB for return prediction, or speculatively execute instructions following the return (straight-line speculation). To test BTB fallback, we place an indirect branch and a return at identical offsets within 32 KB-aligned regions, ensuring predictor collisions based on our BTB analysis. We train the indirect branch to jump to a gadget encoding a known value, then execute deep recursion to exhaust the RSB; if the return falls back to BTB prediction, it speculatively executes our trained target. For straight-line speculation, we place speculative loads immediately after a return instruction in deep recursion. Neither the C910/C920 nor the P550 exhibits BTB fallback or straight-line speculation. We confirm this by examining the open-source C910 Verilog code, which shows the RSB operates independently without BTB fallback on underflow. Retpolines would not be undermined by this behavior, though other RISC-V processor implementations may differ.

Spectre-STL. Spectre-STL exploits mispredicted store-to-load forwarding, where a load speculatively reads stale data before a prior store executes. Our in-process setup uses two aliasing pointers for adjacent store and load instructions in the instruction stream; a pointer chase stalls the store address resolution, allowing the load to race ahead and read stale data.

The P550 implements a memory dependence predictor that learns conflicting store-load pairs by program counter and stalls loads that previously conflicted with older stores [89]. We evict the victim's predictor entry by executing store-load gadgets at unique addresses before each attack. The C910/C920 does not appear to implement such a predictor, as the attack works without predictor eviction. Both processors are vulnerable, achieving more than 99 % success rate.

3.4 Summary

Our results demonstrate that commercial RISC-V silicon is vulnerable to multiple Spectre variants. These findings establish that commercial RISC-V processors require similar mitigations as x86 and ARM.

Processor analysis takeaways:

- 12 of 13 Spectre variants succeed on both processors.
- BTB disabled by default on C910/P550, enabled on C920; all vulnerable when enabled.
- Neither processor falls back to BTB on RSB underflow, enabling retpolines as a mitigation mechanism.
- Timer mitigations are bypassable.

4 Exploiting Spectre in the RISC-V Ecosystem

Prior work has demonstrated kernel exploits on x86 and ARM processors [9, 48, 55, 101, 103]. However, developing a reliable Spectre exploit on RISC-V is challenging due to the absence of mature tooling. The following sections detail how we overcome these challenges: finding exploitable gadgets in the Linux kernel, building a reliable kernel memory read primitive, and turning it into an arbitrary file read.

Experimental Setup. We demonstrate a BPF-assisted Spectre-PHT exploit against the vendor-supported Linux `6.6.119-th1520` kernel provided by T-Head for C910 platforms. On this kernel, the BPF verifier rejects programs containing Spectre-PHT gadgets by default. To load our exploit program, we therefore disabled BPF verification during our experiments. We discuss the implications of this setup in Section 4.1.3. Although vendor-supported Linux v6.17+ kernels are not available for our C910 evaluation board, our exploit is in principle applicable to recent mainline kernels with VeriFence [28] support without disabling BPF verification.

4.1 Spectre Gadget Discovery on RISC-V

The first challenge is to identify vulnerable gadgets in the kernel. Existing tools have limited RISC-V support, so we develop a complementary approach that reveals vulnerabilities in the Linux kernel's BPF JIT, syscall table indexing,

Table 3: Spectre-PHT gadget scanning tools by analysis type (source code vs. binary) and support for targeting the Linux kernel and the RISC-V architecture.

	Tool	Kernel	RISC-V
Source	Smatch [16]	●	●
	Coverity† [12]	●	●
	CodeQL [32, 118]	◐	●
	Kästner et al. [49]	○	●
	KLEESpectre [97]	○	●
	Zhu et al. [117]	○	●
	Specognitor [84]	○	●
Binary	Kasper [48]	●	○
	FastSpec [94]	○	○
	SpecFuzz [75]	○	○
	oo7 [98]	○	○
	Spectector [36]	○	○
	Binsec/Haunted [22]	○	○
	Teapot [61]	○	○
	SpecTaint [79]	○	○

● works, ◐ partial, ○ not supported, †proprietary

user copy functions, futex operations, and Kernel-based Virtual Machine (KVM) interrupt handling subsystems. We also develop patches for all identified gadgets (Section 6).

4.1.1 Existing Tools

Automated gadget scanning tools have become the standard approach for identifying Spectre vulnerabilities on x86 and ARM systems. Table 3 summarizes existing gadget detection approaches. The tools span a range of techniques: pattern matching and dataflow analysis [12, 16, 32, 98], symbolic execution [22, 36, 97, 101], dynamic analysis and fuzzing [61, 75, 79], neural embeddings [94], and hybrid approaches [48]. Notably, none of the existing binary-level tools support RISC-V, even though the underlying infrastructure does (e.g., angr [101] and QEMU [79] both support RISC-V analysis).

Source-level tools, on the other hand, are architecture-agnostic by nature, but most lack kernel scanning support, focusing instead on userspace applications. We systematically evaluate available source-level tools supporting kernel analysis on Linux 6.6, the version shipping with the C910. Smatch [16], the static analysis tool bundled with the Linux kernel, requires no modification for RISC-V and identifies 26 candidate gadgets. Coverity is proprietary and therefore excluded from evaluation. CodeQL [32] requires updating the existing Spectre-specific queries [118], which rely on a CodeQL data-flow API that is no longer supported. After porting them to the current API, CodeQL produces 481 candidates. However, both tools are heavily skewed toward architecture-independent code: of CodeQL's 481 warnings, only 10 target the RISC-V kernel subsystem (all valid, in KVM interrupt handling), while Smatch finds none. Manual triage confirms that source-level tools miss architecture-specific gadgets in critical paths, such as user memory access and syscall entry, where vulnerable patterns involve inline assembly that these tools do not analyze.

4.1.2 Mitigation Diffing

To address this tooling gap, we develop a complementary approach: *mitigation diffing*. Rather than searching for gadgets directly, we leverage the observation that x86 and ARM64 kernels have undergone extensive Spectre hardening since 2018. By comparing hardened code paths against RISC-V implementations, we identify locations that lack equivalent protections, such as bounds masking and speculation barriers.

Methodology. We catalogue 44 mitigation sites across x86, ARM64, and PowerPC by analyzing kernel commit history for Spectre-related patches. We then systematically search for RISC-V equivalents by extracting 2187 function bodies from the RISC-V kernel subsystem using ctags (a source code indexing tool). We compute semantic code embeddings using UniXcoder [37], a pre-trained neural model that converts source code into vector representations capturing function behavior independent of variable names or formatting. For each mitigated function, we rank RISC-V candidates by semantic similarity and manually verify high-scoring matches.

While FastSpec [94] uses neural embeddings to classify known gadget patterns in x86 binaries, our approach uses embeddings to match semantically equivalent functions across architectures, identifying where mitigations are absent rather than where gadgets are present. Semantic matching is essential because equivalent functions across architectures often differ in names and file organization while serving the same purpose: for example, x86's KVM debug register handler matches RISC-V's KVM register configuration function, both handling vCPU register operations but sharing neither function names nor file paths that text search would find. Manual cross-referencing of 44 x86/ARM mitigation sites against all RISC-V candidate functions would require substantial effort; our embedding-based approach automates the ranking and reduces verification to top candidates per site. Precomputing embeddings takes 3 minutes; subsequent queries complete in under 2 seconds.

Results. Of the 15 sites with clear RISC-V equivalents, we found that none contain Spectre mitigations in the unmodified kernel. After manual triage, we identify 5 distinct affected subsystems: syscall dispatch, KVM, BPF JIT, user copy functions, and futex operations. In comparison, CodeQL identifies only 1 of these 5 categories (KVM), while Smatch finds no gadgets in RISC-V-specific code. We develop patches for all identified gaps in Section 6.

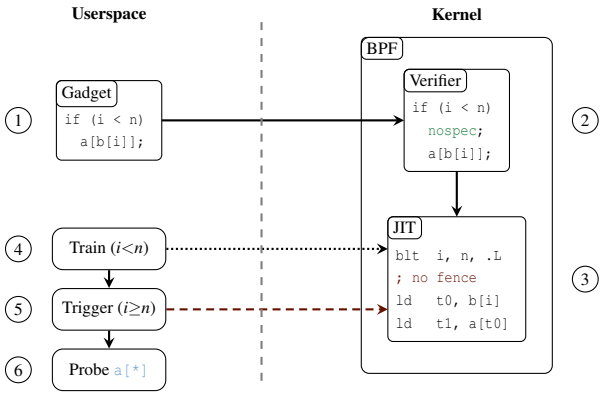

Figure 1: Spectre-PHT exploit via BPF. The attacker injects a gadget ①, the verifier inserts a `nospec` barrier ②, but the JIT emits no fence ③. The attacker mistrains the predictor ④, triggers speculative out-of-bounds access ⑤, and recovers the secret via cache probing ⑥.

4.1.3 Exploitability Assessment

Not all identified gadgets are exploitable on the tested processors. The limitations that render certain gadgets unexploitable on current processors are, however, *not* ISA-mandated but reflect microarchitectural design decisions specific to existing RISC-V implementations. Future RISC-V processors may, therefore, enable exploitation of currently inert gadgets.

First, the `get_user()` and futex gadgets require the kernel to access userspace memory, which on RISC-V involves setting the Supervisor User Memory bit in the privileged `sstatus` control and status register (CSR). While not specified in the ISA, we empirically verify (cf. Section 5.3) that CSR writes halt speculation, thus making the gadget unexploitable on current processors. This was also confirmed by SiFive and T-Head. Second, the syscall dispatch gadget requires redirecting an indirect branch to an encoding gadget, but the current RISC-V processors either disable the BTB entirely or isolate userspace processes from influencing privileged branch predictions (cf. Section 3.3). Finally, KVM gadgets require virtual machine support via the RISC-V H-extension, which has only recently been ratified and is unavailable on the C910/C920, with only a pre-ratified version present on the P550. This leaves the BPF gadget as the only viable attack vector, though it requires unprivileged BPF to be enabled.

BPF Spectre Gadget. The Linux kernel's BPF verifier is hardened against Spectre attacks. On older kernels, the verifier defaults to rejecting BPF programs containing Spectre-PHT gadgets. On recent mainline kernels (6.17+), VeriFence [28] changed this policy: the verifier now accepts such programs and inserts `BPF_NOSPEC` barriers instead. We verified that the x86 JIT backend emits `lfence` for each barrier, whereas the RISC-V backend emits nothing, leaving BPF programs vulnerable to Spectre-PHT. Figure 1 illustrates how this emtpy `BPF_NOSPEC` behavior allows a BPF program with a speculative memory disclosure gadget to pass the verifier unimpeded.

As no vendor-supported v6.17+ kernel is available for our C910 evaluation board, we conduct our evaluation by explicitly disabling BPF verification on an older `6.6.119-th1520` kernel. However, RISC-V systems running newer kernels (Section 6) are vulnerable to BPF-assisted Spectre-PHT exploits when unprivileged BPF is enabled.

4.2 Arbitrary Read Primitive

Constructing a Spectre exploit that provides an arbitrary kernel-memory read primitive on RISC-V presents several challenges. Compared to x86, current RISC-V processors exhibit microarchitectural limitations that complicate end-to-end exploitation.

First, as discussed in Section 3.3, existing RISC-V processors either disable the BTB or prevent userspace applications from influencing kernel-space branch predictions. We did not identify any alternative predictor state shared across privilege levels that could be leveraged to manipulate indirect branch predictions, unlike mechanisms such as Branch History Injection on certain x86 processors [9]. Consequently, rather than targeting indirect branches through Spectre-BTB, we focus on Spectre-PHT exploitation using the previously identified BPF gadgets. Furthermore, contemporary RISC-V processors do not support SMT. Thus, attacks on RISC-V processors cannot spawn a thread on a sibling core to easily mistrain branch predictors in parallel. Finally, we found that certain Linux APIs are less suitable for exploitation on RISC-V. For example, although prior work demonstrated reliable cache-hit and cache-miss discrimination on x86 using the BPF helper function `bpf_ktime_get_ns` [44], we were unable to achieve comparable reliability on the evaluated RISC-V systems.

Exploiting Speculative Type Confusion in BPF. We build a working exploit that uses unprivileged BPF to load a kernel-resident BPF program containing a Spectre-PHT gadget based on conditional branch misprediction, similar to prior work on x86 processors [9,44]. We include the complete BPF program that we exploit in Appendix C. The BPF program reads its arguments from a BPF map populated by userspace. Depending on the input, it either dereferences a valid in-bounds pointer or replaces the pointer register with an attacker-controlled scalar value. Architecturally, the scalar is never dereferenced, as such behavior would be rejected by the BPF verifier.

By repeatedly executing the valid-pointer path, an attacker trains the branch predictor to expect a pointer dereference. When the scalar path is subsequently executed, the processor transiently mispredicts the branch and speculatively dereferences the attacker-controlled scalar. This speculative type confusion enables loading arbitrary kernel memory during

the transient execution. The transiently loaded value is subsequently used as an index into a BPF array map, `cc_map`, to transmit the secret over a cache-based covert channel.

Covert Channel: Memory-Mapped BPF Maps. After the transient access into `cc_map`, the attacker recovers the accessed index via cache timing. Since 2019, BPF array maps support the `BPF_F_MMAPABLE` flag, which allows mapping them into userspace via `mmap`. The attacker uses this mapping to perform Flush+Reload [110] on `cc_map`.

Covert Channel: Prime+Probe. While the previously described covert channel works well for our exploit, it relies on the BPF flag `BPF_F_MMAPABLE`. To demonstrate that removing this feature would not fix the issue, we also develop a unified RISC-V Prime+Probe library that automates eviction set construction using huge pages (providing contiguous physical memory without requiring privileged access). In contrast to advanced x86 microarchitectures, current RISC-V processors do not use cache slices. Thus, we can derive all required indexing bits directly from the number of cache sets, e.g., the C910's 512-set L1 data cache requires 9 bits for indexing. The developer manuals further document the cache replacement policies, thus removing the need for reversing them. The C910 uses deterministic FIFO replacement [92], which enables reliable eviction and Prime+Count [29], an optimized variant of Prime+Probe counting the number of evicted cache lines. The P550 uses random replacement in its last-level cache, where we observe no improvement from more sophisticated eviction strategies (different address ordering and repetition [35]) compared to linear pointer chasing.

Performance. We test our exploit on the Xuantie C910 using the memory-mapped BPF maps as a covert channel. We achieve a median leakage rate of 338 B/s. While slower than on x86, the leakage rate is sufficient for practical exploitation. We next show how targeted exploit engineering overcomes the lower bandwidth by efficiently locating specific file contents in kernel memory.

4.3 Controlled Data Exfiltration

To demonstrate the practical impact of Spectre on RISC-V, we develop an exploit that enables arbitrary file disclosure.

Locating File Contents. Prior x86 exploits locate sensitive files by scanning the physical-memory direct map for recognizable signatures, such as `/etc/shadow` [83, 101]. This approach is less suitable on RISC-V due to lower leakage rates. Moreover, target files may lack known signatures or contain content unknown to the attacker.

Our approach targets files in the kernel's page cache. Linux caches recently accessed file contents in kernel memory to avoid repeated disk accesses. Sensitive files read during boot (e.g., `/etc/shadow` and SSH host keys) or used by long-running services (e.g., TLS private keys) typically remain cached for extended periods. An attacker may also induce caching by triggering operations that access the target file, such as a failed authentication attempt that causes `/etc/shadow` to be loaded.

Linux maintains a hash table that maps cached files to their in-memory page data. The hash key is derived from the file's inode number and a filesystem identifier. The inode number is public metadata and can be obtained through the `stat` interface without requiring read access to the file itself. Since inode numbers are only unique within a filesystem, the kernel also includes the address of a kernel structure representing the mounted filesystem in the hash. We obtain this pointer by traversing kernel data structures starting from `init_task`, whose address is constant on the evaluated kernels because kernel address space layout randomization (KASLR) is not active.[2] From there, we compute the hash, locate the corresponding page-cache entry, follow the associated pointers, and recover the file contents byte by byte. Similar to our exploit, Rain [38] traverses kernel structures but targets cross-VM attacks that require different traversal paths.

Evaluation. We extract `/etc/shadow` (1050 B) in 3.2 s on average (range: 2.65 s to 5.24 s) with 100 % byte accuracy in 77 out of 100 attempts within a 12 s timeout. Attackers can trivially detect stalled attempts via a timeout and restart the exploit, making the remaining timeouts a minor practical limitation. Prior work reports leaking the root password hash after approximately 210 KB of total leakage [101]. Our inode hash lookup reduces the required leakage by three orders of magnitude, locating any cached file by reading 80 B.

These results demonstrate that the absence of Spectre mitigations on current RISC-V systems, deployed on x86 and ARM since 2018, enables practical disclosure of sensitive kernel-resident data. As RISC-V processors enter cloud environments with multi-tenant workloads [85], this mitigation gap presents immediate security risks.

5 Spectre Defenses on RISC-V

This section surveys existing defense proposals for RISC-V (Table 4), systematically identifies which instructions form effective speculation barriers on the C910/C920 and P550 processors, and evaluates compiler-based and kernel-based mitigations. None of the tested processors, kernels, or compilers implements RISC-V-specific Spectre protections. While architecture-independent mitigations already cover some of the attack surface, much of it remains open.

[2] None of the tested C910, C920, or P550 vendor kernels enable KASLR. Our Spectre read primitive could also be used to bypass KASLR if enabled.

Table 4: RISC-V hardware (HW) and software (SW) countermeasures and the Spectre variants they claim to mitigate.

	Mitigation	Target	PHT	BTB	RSB	STL
SW	`fence` insertion [105]	LLVM	◐	◐	◐	◐
	retpoline [95]	LLVM	○	◐	○	○
	SLH [17]	LLVM	●	○	○	○
HW	OISA [111]	BOOM	●	●	●	●
	SbC [45]	BOOM	●	●	●	●
	SEE-RV [71]	BOOM	●	●	●	●
	SpecTerminator [46]	BOOM	●	●	●	●
	ConBOOM [116]	BOOM	●	●	◐	◐
	SpecLFB [20]	BOOM	●	●	●	●
	fence.t [106]	Ariane	◐	◐	◐	◐
	Dome [25]	Salers	◐	◐	◐	◐
	fence.spec [2]	NaxRiscv	●	●	●	●
	ProSpeCT [23]	Proteus	●	●	●	●
	MI6 [14]	RiscyOO	◐	◐	◐	◐
	CURE [6]	Rocket	◐	◐	◐	◐

○ does not mitigate, ◐ partially mitigates, ● fully mitigates

RISC-V has practical advantages for prototyping hardware mitigations due to its mature, synthesizable processor implementations. Unlike x86 or ARM research that typically relies on simulators like gem5, mitigations can be implemented and evaluated on FPGA-synthesizable processors that more closely reflect real hardware behavior. Table 4 summarizes proposed hardware and software mitigations for RISC-V.

5.1 Hardware-Based Defenses in Research

Several hardware mitigation designs have been implemented on RISC-V softcores [20, 23, 46, 71, 106, 116], which may serve as a starting point for commercial vendors to adopt these mitigations. Following Randal [80], we categorize hardware mitigations into cache side-channel elimination, resource partitioning, and selective speculation.

Cache Side-Channel Elimination. These approaches protect against cache-based side channels, reducing the attack surface for Spectre. InvisiSpec [109] prevents speculative loads from modifying cache state by buffering them until commit. DAWG [53] dynamically partitions the cache by protection domain, isolating speculative accesses. Dome [25] proposes hardware-level timing isolation to prevent leakage across security boundaries. SpecLFB [20] uses the line-fill buffer to isolate speculative cache state until commit on BOOM.

Resource Partitioning. These approaches partition microarchitectural resources to prevent attackers from encoding secrets in shared state. CURE [6] provides enclave-based isolation for RISC-V, partitioning caches and turning off SMT between enclaves. Wistoff et al. [104, 105] implement temporal partitioning on the CVA6 processor, flushing microarchitectural state on context switches. While partitioning reduces cross-domain leakage, it does not prevent intra-domain attacks in which an attacker and a victim share resources.

Selective Speculation. These approaches restrict speculation to safe scenarios, delaying potentially unsafe speculative operations. STT [112] tracks taint through speculative execution, blocking transmissions of tainted data. SafeSpec [52] uses shadow structures to isolate speculative state until commit. On RISC-V specifically, SpecTerminator [46] blocks speculation based on instruction classes on BOOM. MI6 [14] integrates speculation control into a secure enclave design. ProSpeCT [23] provides provably secure speculation for constant-time code on Proteus. ConBOOM [116] provides a configurable BOOM variant with selectable covert-channel mitigations. Sabbagh et al. [71] develop taint tracking to delay speculative loads until preconditions are verified.

5.2 Compiler-Based Defenses

After the discovery of Spectre attacks, compiler-based mitigations emerged: fence insertion and Speculative Load Hardening (SLH) for Spectre-PHT, and retpolines [17,95] for Spectre-BTB. We evaluate the availability of these mitigations for RISC-V in GCC and LLVM and find that mainstream compiler toolchains do not support them on RISC-V.

Fence Insertion. On certain architectures, memory fence instructions halt speculation and thus enable limiting speculation at strategic locations. Therefore, it has been proposed as a mitigation against Spectre attacks [105]. On x86, all major compilers support inserting fences after conditional branches to prevent Spectre-PHT attacks [54, 77]. This approach rewrites conditional branches by inserting fences at branch targets to prevent speculative execution past security-critical checks. This approach has significant downsides: it drastically limits performance by blocking all speculation, not just vulnerable paths, and requires a speculation barrier instruction. Crucially, the RISC-V ISA lacks a dedicated speculation barrier instruction. We analyze RISC-V candidate instructions on our evaluation platforms in detail in Section 5.3.

Speculative Load Hardening. Speculative Load Hardening (SLH) is a compiler-based mitigation against Spectre-PHT attacks [17]. SLH tracks whether execution is currently on a misspeculated path by maintaining a speculation predicate. After each conditional branch, the predicate is updated using branchless code, creating a dependency to the branch condition. The predicate is then used to mask subsequent load addresses or values, preventing secrets from being accessed speculatively. Unlike manual array bounds masking, SLH is applied automatically by the compiler and protects all loads following conditional branches. The original SLH

has been shown to be bypassable [115], leading to Ultimate SLH [115], which addresses these shortcomings. While mainstream compilers like LLVM and GCC do not implement SLH for RISC-V, an open-source implementation of Ultimate SLH for RISC-V is available [30].

Retpolines. Retpolines [95] prevent Spectre-BTB attacks by replacing indirect branches with a code sequence that uses a return instruction to redirect control flow. As a result, speculative execution follows the software-prepared RSB rather than a potentially attacker-poisoned BTB. Incorrect speculative execution becomes trapped in a benign infinite loop, while architectural execution continues along the intended control-flow path.

Adapting retpolines to RISC-V presents fundamental challenges due to differences in calling conventions. On x86, retpolines redirect control flow by overwriting the return address stored on the stack. In contrast, RISC-V maintains the return address in the `ra` register and only saves it to the stack explicitly in function prologues. Consequently, simply overwriting `ra` is insufficient. The only known software solution is the "skip-prologue" design [8]. Here, the retpoline allocates the callee's stack frame, stores the target address at the stack location where the callee expects to save `ra`, and then transfers control past the callee's prologue. This avoids re-executing the stack-frame setup while ensuring that the target address is restored when the callee returns. Supporting this mechanism requires the compiler to emit function prologues in a strict two-phase structure. The first phase, which is skipped by the retpoline, must contain only stack allocation and the saving of `ra` and `fp`. Any remaining callee-saved registers must be handled in a separate second phase.

This dependency on a specific prologue structure makes RISC-V retpolines inherently fragile. We found that the existing LLVM prototype [8] fails to execute SPEC CPU 2017 correctly because prologue generation does not consistently produce the required two-phase layout, resulting in stack corruption and crashes. Addressing this issue would require modifications throughout the LLVM backend rather than a localized change to the retpoline implementation.

RISC-V retpolines also face additional challenges. The limited range of control-transfer instructions (`jal`: 20-bit offset, `jalr`: 12-bit immediate) prevents the use of a single shared trampoline as on x86 and instead necessitates multiple trampolines. Furthermore, retpolines rely on the RSB not falling back to BTB-based predictions on underflow. Our evaluation confirms this property on the C910, C920, and P550 (Section 3), but other RISC-V microarchitectures may behave differently. Finally, RSB stuffing [56] and related defenses have not yet been investigated on RISC-V.

Branchless JIT Dispatch. Given these challenges, we explore an alternative approach that eliminates indirect branches entirely. Inspired by Switchpoline [10] on ARM, our branchless JIT dispatch dynamically generates direct branch instructions at runtime. This avoids the RSB dependence and prologue splitting that complicate retpolines on RISC-V, and requires no speculation fences. The dispatch path contains no conditional or indirect branches, making it immune to both Spectre-PHT and Spectre-BTB during dispatch.

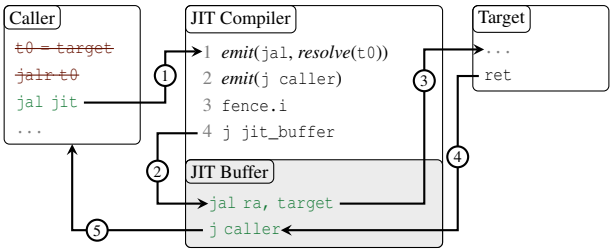

Figure 2: Branchless JIT dispatch replaces unsafe indirect calls with safe direct branches. The caller ① jumps to the JIT compiler, which encodes the resolved target as a direct `jal`, synchronizes the instruction cache, and ② jumps to the JIT buffer. The buffer ③ calls the target, which ④ returns to the buffer. Finally, the buffer ⑤ returns to the caller.

Switchpoline can in principle be applied to RISC-V, but its switch conversion introduces Spectre-PHT gadgets on the dispatch path that must be mitigated with a fence or another form of serialization. This is a problem on RISC-V, which provides no architecturally guaranteed speculation barrier (Section 5.3). Our branchless JIT dispatch instead generates a direct branch at runtime, so it requires no such barrier.

Figure 2 illustrates the mechanism. The core technique computes the PC-relative offset to the target and encodes it into the `jal` instruction's immediate field using shift, mask, and logical or operations. The encoded instruction is written to a pre-allocated executable memory region (the JIT buffer). After a `fence.i` to synchronize the instruction cache, execution transfers to the JIT buffer via a direct jump. The JIT buffer contains two instructions: a `jal` that calls the target directly, and a second `jal` that returns control to the caller. Because all jumps are direct, indirect branch prediction is avoided, eliminating the Spectre-BTB attack surface.

To evaluate practical overhead, we manually modify the uBPF interpreter source code [43] to use our JIT dispatch mechanism. We benchmark with bpf-benchmark [26] on the C910; slowdown ranges from 1.2× for programs with few dispatches to 22× for instruction-intensive workloads.

However, this technique has limitations. The `jal` instruction's ±1 MB offset restricts targets to a limited address range. The `fence.i` incurs overhead on every dispatch, dominating execution time in instruction-intensive workloads. The RISC-V specification acknowledges this limitation and discusses more efficient address-targeted alternatives for future extensions [99]. Our implementation uses a single JIT slot, which is not thread-safe and would require per-thread or per-

core slots in production. Future work could implement this as an LLVM pass to automatically transform indirect calls, handle the full range of calling conventions, and manage JIT slot allocation across threads.

5.3 Speculation Barriers

A critical component of Spectre mitigations is the ability to stop speculative execution. On x86, `lfence` provides this guarantee [64], while ARM offers `CSDB` (speculation barrier) and `ISB` (instruction synchronization barrier) [4]. However, the RISC-V ISA provides no dedicated speculation barrier instruction, a critical gap that leaves software mitigations without architectural support. The RISC-V `fence` instruction only serializes memory operations and does not guarantee that speculation is halted [99]. Previous work has proposed the `fence.t` [104, 105] extension providing stronger guarantees by scrubbing microarchitectural state. However, this extension is not ratified and unavailable on commercial processors.

Instruction Profiling. To identify instructions that may act as speculation barriers on commercial RISC-V processors, we systematically profile candidate instructions on our evaluation platforms. Testing candidate barriers using Spectre exploits directly is problematic: an instruction may prevent a particular side channel without actually halting speculative execution, creating a false impression of protection. Since exhaustively evaluating all possible side channels is infeasible, we instead use processor exceptions as an overapproximation. Exceptions provide a clear distinction between architectural and transient execution. Architecturally, execution is redirected to an exception handler; transiently, instructions already in flight may continue to execute. Our exception triggers are designed to minimize dependencies on subsequent instructions (e.g., loads from address zero, encoded as an immediate target address operand). Consequently, a candidate instruction that prevents transient execution in this setting must do so by halting instruction issue rather than due to incidental data dependencies.

Our framework generates JIT code that triggers an exception, executes a candidate barrier instruction, and then performs memory accesses to a probe array. A cache side channel determines whether speculative accesses occurred after the candidate barrier instruction. One limitation of this approach is that an instruction that suppresses all transient memory accesses, without actually halting instruction issue, would also appear effective. Such a mechanism may still be bypassed through non-memory side channels, such as port contention.

We test 12 candidate instructions, covering all fence variants in the base ISA and other unprivileged instructions that may serialize the pipeline: no barrier (baseline), memory fences that order reads (`fence`, `fence.rw`, `fence.r`), write-only fences (`fence.w`, `fence.tso`), the instruction fence (`fence.i`), `pause`, CSR reads (`rdcycle`, `rdtime`, `rdinstret`), and `lr.d` (atomic load-reserved). We test these across 5 exception categories: access faults (load/store to address 0), page faults (load/store to unmapped memory), instruction page faults (jump to unmapped address), illegal instructions, and breakpoints.

Table 5: Candidate barrier instruction effectiveness across exception types for the evaluated platforms (P550 / C910).

Barrier \ *Fault*	**Access**	**Page**	**Inst Page**	**Illegal Inst**	**Breakpoint**
none	○/○	○/○	●/●	●/○	●/○
`fence`	○/●	○/●	●/●	●/●	●/●
`fence.rw`	○/●	○/●	●/●	●/●	●/●
`fence.r`	○/●	○/●	●/●	●/●	●/●
`fence.w`	○/○	○/○	●/●	●/●	●/○
`fence.tso`	○/○	○/○	●/●	●/○	●/○
`fence.i`	●/●	●/●	●/●	●/●	●/●
`pause`	○/●	○/●	●/●	●/●	●/●
`rdcycle`	○/●	○/●	●/●	●/●	●/●
`rdtime`	●/●	●/●	●/●	●/●	●/●
`rdinstret`	○/●	○/●	●/●	●/●	●/●
`lr.d`	○/○	○/○	●/●	●/●	●/●

● no speculation observed after the tested barrier instruction

Results. Table 5 shows the results. On the C910, most fence variants and CSR reads effectively stop speculation, with `fence.tso` and `lr.d` being ineffective. On the P550, only `fence.i` and `rdtime` reliably stop speculation after load/store faults. Notably, instruction page faults immediately halt speculation on both processors without requiring any barrier, while illegal instructions and breakpoints do so only on the P550. To validate these findings, we also test barriers using the Spectre proof-of-concept attacks from Section 3, confirming that effective barriers reduce attack success rates to 0 %.

However, none of these instructions is architecturally guaranteed to stop speculation. The RISC-V specification leaves microarchitectural behavior explicitly unspecified, so instructions that halt speculation on one processor may not do so on another. For example, `fence.i` stops speculation on both tested processors, but its architectural purpose is instruction stream synchronization. Other implementations could achieve this without affecting speculative execution.

Our methodology enables practical deployment despite this uncertainty: the same fault-based tests can be used at runtime to detect effective barriers for a given processor. System software can probe candidate instructions during boot and select the lowest-overhead barrier that stops speculation on the detected processor. We release our testing framework to enable this approach. We have engaged with the RISC-V Speculation Barrier Task Group, which is working to standardize a dedicated speculation barrier instruction with architectural guarantees. Until such an extension is ratified and available in hardware, per-processor testing remains necessary for reliable Spectre mitigations.

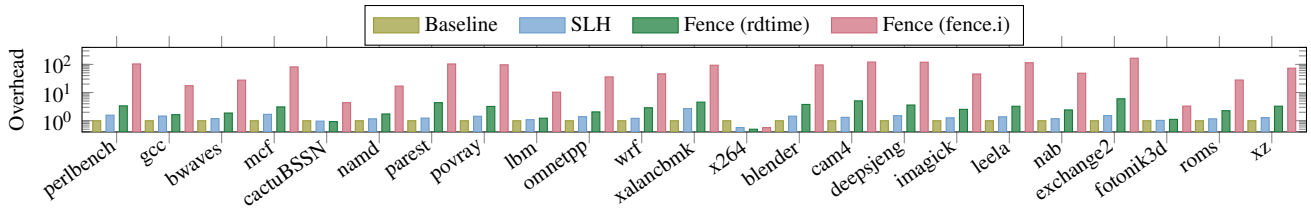

Figure 3: Performance overhead of compiler mitigations on RISC-V for SPEC CPU 2017 rate (test workload), normalized to baseline on the C910.

This has concrete security implications: upstream software already assumes reliable speculation barriers exist. The kernel's `barrier_nospec()` macro, which emits `lfence` on x86, has no equivalent on RISC-V and compiles to a no-op. Similarly, VeriFence [28], a Spectre mitigation for BPF that is upstream in Linux, relies on speculation fences to provide security guarantees.

5.4 Evaluation of Compiler Mitigations

To test the robustness and performance impact of compiler-based mitigations on RISC-V, we use the SPEC CPU 2017 rate benchmarks [21], used in previous work [96, 98, 115]. Apart from measuring performance overhead, the benchmarks also stress test the correctness of compiler-based mitigations. Particularly, we find that the robustness of compiler-based mitigations on RISC-V is not at a production-ready level. The LLVM-based retpoline prototype [8] fails to produce correct binaries for all benchmarks in SPEC CPU 2017. The generated binaries do not correctly restore the stack pointer offset in some situations, leading to segmentation faults at runtime. Therefore, we cannot evaluate the performance impact of retpolines on RISC-V. Fence insertion and SLH both produce correct binaries for all benchmarks.

Experimental Setup. We benchmark all our mitigations on the C910, as our P550 board was too unstable to complete a full benchmark run. For fence insertion, we use the `fence.i` and `rdtime` instructions, both of which stop speculation on the tested processors (cf. Table 5).

We evaluate all SPEC CPU 2017 23 rate benchmarks (10 intrate, 13 fprate). We use the test workload (ref inputs would require multiple days per benchmark with fence insertion, making a full evaluation infeasible). A few benchmarks (525.x264_r, 502.gcc_r) have baseline runtimes below 100 ms on test inputs, where per-benchmark ratios are within measurement noise; we report them, but they do not meaningfully affect the geometric means.

Results. Figure 3 shows per-benchmark slowdowns for all 23 SPEC CPU 2017 rate benchmarks under SLH, `rdtime`-based fence insertion, and `fence.i`-based fence insertion on the C910, normalized to baseline.

The `fence.i` instruction-based fence shows a geometric mean slowdown of 46.6× across all benchmarks, with the highest slowdown of 165× for 548.exchange2_r. Across all 23 benchmarks, the compiler inserts roughly two million `fence.i` instructions, with the 502.gcc_r benchmark alone requiring 312,000 insertions. This slowdown occurs because `fence.i` flushes the entire instruction cache on the tested RISC-V processors. Floating-point benchmarks see substantially lower overhead (geometric mean 30.9×) than integer benchmarks. As an alternative, we evaluate `rdtime` writing to the `x0` register (discarding the result) as the speculation barrier. The `rdtime`-based fence achieves a geometric mean slowdown of only 2.69×, compared to 46.6× for `fence.i`. This dramatic improvement occurs because `rdtime` serializes the pipeline without flushing the instruction cache or clobbering registers.

SLH has an even lower geometric mean slowdown of 1.34×, ranging from 0.97× (507.cactuBSSN_r, within noise) to 2.69× (523.xalancbmk_r). For context, x86 implementations of these mitigations report comparable to slightly higher overheads on SPEC CPU, with SLH around 2.5× and conservative fence insertion around 4× [98, 115]. Retpolines are excluded from the plot as they fail to produce correct binaries.

Software mitigation takeaways:

- RISC-V compilers lack Spectre mitigations.
- LLVM retpoline prototype fails on SPEC CPU 2017.
- RISC-V lacks a standardized speculation barrier; ad-hoc substitutes incur substantial overheads (on the C910: 46.6× for `fence.i`; 2.69× for `rdtime`).
- Branchless JIT dispatch can mitigate Spectre-BTB.

6 Protecting the Linux Kernel

The Linux kernel is a prime target for Spectre attacks, and multiple prior works have demonstrated exploitation through kernel-resident gadgets [7, 56, 103]. Architectures such as x86 and ARM provide several layers of mitigations. These mitigations include code rewriting to remove gadgets, compiler-aided hardening, hardware-assisted features such as IBRS and STIBP, and per-process controls [51]. Nevertheless, new attacks continue to emerge [83, 102], highlighting the diffi-

Table 6: Status of kernel-based mitigation techniques for RISC-V in the Linux kernel and the Spectre variants they address. The C910 supports Linux v5.10 and v6.6, while the P550 supports v6.6. Linux v7.1 is the latest released version at the time of writing. Highlighted rows denote patches contributed in this work: merged upstream (blue; ★) or pending further review (orange; ☆).

Technique	Availability			Mitigates Variant			
	5.10	6.6	7.1	PHT	BTB	RSB	STL
★ User ptr masking	□	□	◪	●	○	○	○
★ Spec. barriers	□	□	◪	◐	◐	◐	◐
☆ Index masking	◪	◪	◪	●	○	○	○
☆ BPF hardening	■	■	□	●	○	○	●
Low-res timers	□	◪	◪	◐	◐	◐	◐
KASLR	◪	◪	◪	◐	◐	◐	◐

□ : not implemented, ◪ : partially implemented, ■ : effective
○ : does not mitigate, ◐ : partially mitigates, ● : fully mitigates

culty of comprehensively securing a code base as large and complex as the Linux kernel. In this section, we examine the current state of Spectre mitigations for RISC-V in the Linux kernel and present the patches we contributed to address the gaps identified in our analysis.

6.1 Kernel Mitigation Status

Table 6 overviews the implementation status of the investigated kernel-level mitigation techniques across kernel versions, along with the Spectre variants each technique addresses or partially mitigates.

User Pointer Masking. User copy functions such as `copy_from_user` must prevent speculative bypass of the `access_ok()` bounds check. x86 uses `lfence` barriers for this; ARM64 uses pointer masking instead. We found that, on RISC-V, neither is implemented: `barrier_nospec()` degenerates to a no-op and no masking exists.

Following ARM64's approach, we implement pointer masking using a guard page. RISC-V kernels maintain an unmapped guard region between user and kernel address spaces. Our patch uses branchless arithmetic to clamp any address at or above the boundary into this guard page, ensuring speculative accesses with invalid pointers fault. This works with all paging modes (Sv39/48/57) and is compatible with pointer masking extensions.

Index Masking. Index masking prevents speculative out-of-bounds array accesses by arithmetically clamping user-controlled indices. The `array_index_nospec` macro computes `size-1-idx` and uses an arithmetic right shift to propagate the sign bit: valid indices produce an all-ones mask,

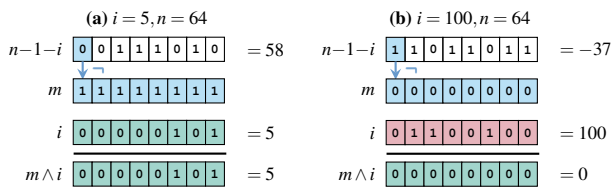

Figure 4: Index masking with `array_index_nospec`, where i is the index, n the array size, and m the mask. The mask is derived by computing $n-1-i$, negating, then spreading the sign bit to all positions via arithmetic right shift. (a) Valid indices produce a positive difference and an all-ones mask. (b) Out-of-bounds indices produce a negative difference and an all-zeros mask that clamps the index to zero.

while out-of-bounds indices produce an all-zeros mask (Figure 4). A logical `and` operation of the mask and the index preserves valid values and clamps invalid ones to zero [50].

On x86 and ARM64, this macro is applied throughout the kernel wherever user-controlled values index into arrays. While the macro is available and functions correctly on RISC-V, we found that it is not used consistently in security-critical code paths in the `arch/riscv` subsystem. In particular, the syscall dispatcher indexes the syscall table using a user-controlled syscall number after a bounds check. Under speculative execution, an attacker may bypass this check and transiently perform an out-of-bounds table access, potentially disclosing adjacent kernel memory. We address this issue by applying `array_index_nospec`, consistent with the existing x86 and ARM64 implementations.

BPF Hardening. BPF programs execute dynamically compiled code in kernel space and can potentially mount multiple Spectre variants. The Linux kernel offers multiple mitigations: disabling unprivileged BPF loading (`CONFIG_BPF_UNPRIV`), protecting indirect branches in JIT-generated code, and inserting `BPF_NOSPEC` speculation barriers.

The BPF verifier inserts `BPF_NOSPEC` instructions at potentially dangerous code paths. On x86, the JIT lowers `BPF_NOSPEC` to `lfence`. On ARM64, it emits `SB` (speculation barrier) when available and otherwise falls back to `DSB+ISB`. VeriFence [28] further extends the use of `BPF_NOSPEC` to mitigate Spectre-PHT by inserting barriers instead of rejecting programs containing potential gadgets. Crucially, as exploited in Section 4.1.3, we identify that the RISC-V BPF JIT emits no instruction for `BPF_NOSPEC`. As a result, verifier-inserted speculation barriers provide no protection on RISC-V. Since `BPF_NOSPEC` is also used to mitigate Spectre-STL attacks, RISC-V likewise lacks protection against those attacks. Figure 5 illustrates this issue using the Spectre-PHT type-confusion gadget from Section 4: the verifier inserts a barrier, but the RISC-V JIT emits no corresponding instruction, leav-

```
reg = is_ptr ? ptr
: scalar;
if (!is_ptr) goto exit;
ST_NOSPEC;
val = *reg;
leak[val];
```

```
mv    a0, is_ptr ? ptr
: scalar
beqz  is_ptr, exit
fence.i
lbu   a1, 0(a0)
lbu   a2, leak(a1)
```

Figure 5: Speculative type confusion gadget. Left: BPF with `ST_NOSPEC` barrier inserted by the verifier (highlighted). Right: RISC-V JIT output with `fence.i` emitted by our patch (highlighted); without the patch, nothing is emitted.

ing the speculative load unprotected. Our patch adds support for `BPF_NOSPEC` in the RISC-V BPF JIT. As discussed in Section 5.3, effective barrier instructions exist but currently lack architectural guarantees. We are therefore collaborating with the Linux BPF maintainers on a solution that can be updated once a dedicated speculation barrier is standardized.

For Spectre-BTB, indirect jumps used by BPF tail calls must be protected against BTB poisoning. On x86 and ARM64, the JIT implements architecture-specific mitigations (retpolines and BTI, respectively). Retpolines are not currently implemented by the kernel for RISC-V. Nevertheless, the practical impact of this omission is limited on the evaluated processors. The C910 and P550 ship with the BTB disabled via OpenSBI, whereas the C920 enables BTB predictions by default. All three processors additionally partition BTB state between user and kernel privilege levels, preventing cross-privilege mistraining and thereby limiting Spectre-BTB attacks against the kernel.

Low-Resolution Timers. The kernel disables the RISC-V `rdcycle` instruction in userspace to impede high-resolution timing attacks [66]. However, prior work has shown that attackers can construct their own timers or amplify leakage [62, 78, 86]. We experimentally verify that a counter-thread timer indeed achieves ≈6 cycle resolution on the C910, sufficient for all attacks presented. Additionally, on kernels with legacy `perf_user_access` settings, opening a perf event re-enables `rdcycle`.

Kernel Address Space Layout Randomization. KASLR randomizes the base address of the kernel. KASLR slightly raises the bar for Spectre attackers, as they have to leak kernel pointers first to break it. However, other variants to break KASLR via timing side channels on RISC-V exist [29]. KASLR support is implemented in the Linux kernel and available on RISC-V, but the vendor kernels on all tested C910/C920 and P550 platforms disable KASLR.

6.2 Kernel Patches

We developed patches to address the vulnerabilities identified in Section 4.1: the uaccess pointer masking, syscall table indexing, and KVM index masking patches have been accepted into the Linux kernel, while the BPF speculation barrier and futex pointer masking patches are currently under review. However, we stress that these patches are only a first step towards hardening the RISC-V Linux kernel against Spectre attacks. The lack of a dedicated speculation barrier instruction in the RISC-V ISA makes comprehensive mitigation difficult.

Mitigation Selection. The Linux kernel exposes processor vulnerability status via the `sysfs` interface. Architectures must implement handlers for each vulnerability; otherwise, the default reports "not affected". On RISC-V, this interface exists since version 6.12 [67], and version 6.14 added a handler for the RISC-V-specific GhostWrite vulnerability [68]. Notably, no Spectre-specific handlers are currently implemented, causing the kernel to incorrectly report RISC-V processors as unaffected.

On x86 and ARM64, the Linux kernel exposes speculation control via `prctl` to let processes enable hardware mitigations (e.g., STIBP, IBPB, SSBD) on a per-process basis. Current RISC-V processors do not implement such hardware mitigation mechanisms.

Kernel Mitigation Slowdown. We benchmark our kernel patches using LMBench [73], which measures latency and bandwidth of core kernel operations including syscalls, process creation, context switches, and memory operations. On the P550, with all patches applied, the null syscall shows a 65.51× slowdown, as `array_index_nospec` masking dominates the near-zero 0.25 µs baseline latency. However, real-world syscalls incur negligible overhead: `open/close` slows by 1.08×, `stat` by 1.07×, and IPC, process creation, context switching, networking, and bandwidth benchmarks remain within measurement noise (<1.06×). On the C910, overhead is lower overall: the null syscall shows only 1.01× overhead, with `fork+exit` at 1.29× and pipe bandwidth at 1.27× as the notable exceptions.

Linux kernel analysis takeaways:

- `barrier_nospec()`, `BPF_NOSPEC` compile to no-ops.
- Standard architecture-independent hardening (KASLR, index masking) is missing from critical code paths.
- Sysfs reports "not affected" despite exploitable gadgets in syscall dispatch, `copy_from_user`, and BPF JIT.

7 Discussion

Our findings highlight the challenges of transferring mature Spectre mitigations to RISC-V, arising from inherited software assumptions, limited hardware transparency, implementation diversity, and the relative immaturity of the ecosystem.

Inherited Assumptions. Upstream software assumes guarantees that are not provided by RISC-V, most notably a dedicated instruction for halting speculative execution. The kernel's `barrier_nospec()` macro, which emits `lfence` on x86, therefore degenerates to a no-op on RISC-V. Similarly, the BPF verifier inserts `BPF_NOSPEC` barriers for speculative safety, yet the RISC-V JIT emits nothing for them. Consequently, the upstream VeriFence [28] Spectre mitigation, relying on these barriers, is ineffective on RISC-V. Retpolines likewise assume properties that do not readily transfer to RISC-V, including predictable RSB behavior and sufficient branch range to support a shared trampoline design.

Together, these examples illustrate that Spectre mitigations cannot be assumed to transfer across ISAs unchanged; their security guarantees must be carefully re-evaluated in the context of the target architecture.

Hardware Opacity. Current RISC-V processors do not sufficiently expose security-relevant microarchitectural state. On the P550, the BTB configuration register is write-only. The C910 and C920 share identical microarchitecture, yet the C910 (TH1520) disables BTB while the C920 (SG2042) enables it, determined solely by OpenSBI configuration. Without vendor-exposed state, defenders cannot reliably detect which microarchitectural features are active, let alone reason about them; meaningful system mitigations on RISC-V will require hardware introspection interfaces that today's processors do not offer.

Hardware Diversity. Though often praised as a strength, the heterogeneity of competing RISC-V implementations fragments the security landscape. Defenses built for research processors such as BOOM [18] are not adopted on commercial silicon, and microarchitectural behavior differs across processors; the C910 provides unprivileged cache-flush instructions, for instance, while the P550 does not. No single mitigation strategy can serve this fragmented landscape: robust operating-system defenses will likely need to probe hardware capabilities at runtime and select mitigation primitives accordingly, e.g., via the `mvendorid`, `marchid`, and `mimpid` CSRs [65].

Ecosystem Catch-Up. Notably, the RISC-V ecosystem lacks hardening that has been standard on x86 and ARM since at least 2018. Vendor kernels ship with KASLR disabled and code paths unhardened, and compiler-based mitigations are missing from mainstream RISC-V toolchains.

The challenges above share a root cause: RISC-V inherits the software and threat model of mature architectures without their accumulated hardening. Closing this gap is not a matter of porting individual mitigations, but of building the architectural primitives, hardware transparency, and ecosystem-wide tooling that effective Spectre defense presupposes. As RISC-V processors enter cloud and multi-tenant deployments, closing these gaps becomes increasingly urgent.

8 Conclusion

Speculative execution attacks have been extensively studied on x86 and ARM, but RISC-V research has concentrated on open-source academic designs. We systematized Spectre research on RISC-V and developed a unified framework to test 13 attack configurations on commercial out-of-order processors. Our analysis shows that the SiFive P550 and Xuantie C910/C920 are vulnerable to all major Spectre variants, and we demonstrated a Spectre exploit on RISC-V hardware that leaks kernel memory via BPF. We identified critical gaps across the software stack (missing speculation barriers, incomplete kernel mitigations, non-functional retpolines) and contributed patches to the Linux kernel to address them. Our work provides a foundation for securing the RISC-V ecosystem as it matures into security-critical deployments.

Acknowledgments

This research is partially funded by the Internal Funds KU Leuven and the Cybersecurity Research Program Flanders. This work has benefited from the participation of Michael Schwarz and Jo Van Bulck in Dagstuhl Seminar #26042 "Trustworthy System Architectures for the Age of Custom Silicon".

Ethical Considerations

This work demonstrates Spectre attacks on commercial RISC-V processors, revealing that the SiFive P550 and T-Head Xuantie C910/C920 allow unprivileged leakage of kernel memory through speculative execution. Publishing attack techniques on commercially deployed hardware carries dual-use risks: while our findings enable vendors and developers to remediate vulnerabilities, they also provide adversaries with concrete attack primitives. We carefully considered these tradeoffs throughout the research process.

Responsible Disclosure. We reported the issue to the Linux kernel security team on December 3, 2025, and shared draft patches with them privately on December 5, 2025. After the maintainers confirmed that public submission was appropriate, we publicly submitted the patches to the linux-riscv mailing list on December 18, 2025. Three of our patches (uaccess pointer masking, syscall table indexing, and KVM index masking) have since been merged into mainline Linux. The remaining two (BPF speculation barrier and futex pointer masking) are under review. SiFive joined the resulting kernel discussion on December 19, 2025, and we coordinated the P550-specific findings with them directly. We subsequently

disclosed the C910/C920-specific findings to T-Head through the Alibaba Security Response Center on January 24, 2026, which acknowledged the report on January 27, 2026. We provided both vendors with the test tooling from Section 3. Both vendors confirmed that they will publish ad-hoc speculation barriers for their processors. We chose not to delay publication because Spectre has been publicly known since 2018, and our contribution lies in demonstrating that RISC-V is equally affected. This accelerates remediation and raises awareness of the issue, hopefully positively impacting the development of future commercial processors. All experiments were conducted on hardware owned and controlled by the authors in a laboratory setting. No experiments targeted third-party systems, production infrastructure, or workloads belonging to others. The study does not involve human subjects, nor does it collect or process personal data; no institutional ethics board review was required.

Artifact Release. We release our unified PoC framework, measurement tools, gadget scanning pipeline, and kernel patches to enable reproducibility. We deliberately do not release a turnkey end-to-end exploit package: the individual components are provided for validation, but assembling them into a weaponized tool requires significant domain expertise. We believe this strikes a reasonable balance between scientific transparency, enabling future work, and limiting immediate misuse.

Stakeholder Analysis.

- **RISC-V deployers**: Our work reveals that existing deployments lack Spectre mitigations and provides concrete patches. Deployers can adjust their configuration to be more secure (e.g., disabling predictors) if required.
- **CPU vendors**: Our findings inform the affected vendors (SiFive, T-Head) about their vulnerable processors, and enable more informed microarchitectural design decisions.
- **Linux kernel community**: The community benefits from our contributed patches, speculation barrier characterization, and gadget scanning methodology.
- **RISC-V International**: Our work provides concrete motivation for standardizing speculation control extensions in the ISA.
- **Potential adversaries**: While there is a risk that adversaries could use our techniques against unpatched systems, we minimize this risk via our responsible disclosure.

Open Science

Our artifacts are available at: https://doi.org/10.5281/zenodo.18452278. We provide the following components:

- **Spectre PoC Framework**: Unified framework for all 13 Spectre attack configurations (PHT, BTB, RSB, STL) across same-address/cross-address and in-place/out-of-place variants (Section 3).
- **Microarchitectural Measurements**: Measurement code for speculation window and RSB depth analysis (Appendix B).
- **Speculation Barrier Testing**: Testing framework for candidate speculation barrier instructions (Section 5.3).
- **Gadget Scanning Pipeline**: Smatch and CodeQL configurations for RISC-V kernel analysis (Section 4.1).
- **Cross-Architecture Mitigation Diffing**: Semantic embedding-based tool to identify missing RISC-V mitigations (Section 4.1).
- **Mitigation Benchmarks**: Setup to run SLH and fence-based mitigation benchmarks on SPEC CPU 2017 (Section 5.2). SPEC CPU 2017 is proprietary and cannot be redistributed.
- **Branchless JIT Dispatch**: Modified uBPF interpreter with branchless dispatch implementation (Section 5.2).
- **Cache Side-Channel Primitives**: Cache side-channel library including Flush+Reload, Evict+Reload, Prime+Probe with eviction set construction, and a counter-thread timer for C910 and P550 (Section 3.2).
- **BPF Gadget Emitter**: BPF bytecode to reliably emit the type confusion gadget used in the end-to-end exploit (Section 4). We deliberately do not release a turnkey end-to-end exploit; assembling the individual components requires significant domain expertise.
- **Linux Kernel Patches**: All contributed patches to the Linux kernel (Section 6).

References

[1] Spectre variant 4, 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[2] H. Andrianatrehina, R. Lashermes, J. Paturel, S. Rokicki, and T. Rubiano. Exploring speculation barriers for RISC-V selective speculation. In *ARES*, August 2025.

[3] K. Arıkan, A. Palumbo, L. Cassano, P. Reviriego, S. Pontarelli, G. Bianchi, O. Ergin, and M. Ottavi. Processor security: Detecting microarchitectural attacks via count-min sketches. *VLSI*, 2022.

[4] ARM. ARM: Whitepaper Cache Speculation Side-channels, 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper.

[5] C. Austa, J. T. Mühlberg, and J.-M. Dricot. Systematic assessment of cache timing vulnerabilities on risc-v processors. In *ESORICS*, 2025.

[6] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf. CURE: A security architecture with CUstomizable and resilient enclaves. In *USENIX Security*, August 2021.

[7] M. Balliu, M. Dam, and R. Guanciale. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. *arXiv:1911.00868*, 2019.

[8] R. Bălucea and P. Irofti. Software mitigation of risc-v spectre attacks. In *SECITC*, 2023.

[9] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, 2022.

[10] M. Bauer, L. Hetterich, M. Schwarz, and C. Rossow. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *AsiaCCS*, 2024.

[11] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[12] Black Duck Editorial Staff. Detecting spectre vulnerability exploits with static analysis, 2018. URL: https://www.blackduck.com/blog/detecting-spectre-vulnerability-exploits-with-static-analysis.html.

[13] M. Bognar, J. Noorman, and F. Piessens. Proteus: An extensible risc-v core for hardware extensions. In *RISC-V Summit Europe*, 2023.

[14] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[15] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[16] D. Carpenter. Smatch check for Spectre stuff, 2018. URL: https://lwn.net/ml/linux-kernel/20180419051510.GA21898@mwanda/.

[17] C. Carruth. Speculative load hardening, 2018. URL: https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6.

[18] C. Celio, D. A. Patterson, and K. Asanović. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. Technical report, EECS, UC Berkeley, 2015.

[19] T. Chen, R. Matsuo, R. Shioya, and K. Suzaki[1]. Analyzing and mitigating the ssb vulnerability in an mdp-equipped. In *IWSEC 2025*, 2025.

[20] X. Cheng, F. Tong, H. Wang, Z. Zhou, F. Jiang, and Y. Mao. SpecLFB: Eliminating Cache Side Channels in Speculative Executions. In *USENIX Security*, 2024.

[21] S. P. E. Corporation. SPEC CPU 2017, 2017. URL: https://www.spec.org/cpu2017/.

[22] L.-A. Daniel, S. Bardin, and T. Rezk. Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In *NDSS*, 2021.

[23] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens. ProSpeCT: Provably secure speculation for the constant-time policy. In *USENIX Security*, 2023.

[24] A. de Faveri Tron, R. Isemann, H. Ragab, C. Giuffrida, K. von Gleissenthall, and H. Bos. Phantom trails: Practical pre-silicon discovery of transient data leaks. In *USENIX Security*, 2025.

[25] M. Escouteloup, R. Lashermes, J. Fournier, and J.-L. Lanet. Under the dome: Preventing hardware timing information leakage. In *CARDIS*, 2022.

[26] eunomia-bpf. bpf-benchmark: Userspace eBPF Runtime Benchmarking Suite, 2023. URL: https://github.com/eunomia-bpf/bpf-benchmark.

[27] F. A. Fuchs, J. Woodruff, S. W. Moore, P. G. Neumann, and R. N. Watson. Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V. Technical report, University of Cambridge, 2021.

[28] L. Gerhorst, H. Herzog, P. Wägemann, M. Ott, R. Kapitza, and T. Hönig. VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions. In *RAID*, 2024.

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

[30] M. Ghaniyoun. LLVM-SLH-RISCV, 2023. URL: https://github.com/MoeinGhaniyoun/LLVM-SLH-RISCV.

[31] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ISCA*, 2021.

[32] GitHub. CodeQL, 2024. URL: https://codeql.github.com/.

[33] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanovic. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In *CARRV*, 2019.

[34] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, 2018.

[35] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[36] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In *S&P*, 2020.

[37] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv:2203.03850*, 2022.

[38] M. Hertogh, D. Quakkelaar, T. Raymakers, M. H. Sarma, M. Muench, H. Bos, and E. van der Kouwe. Rain: Transiently leaking data from public clouds using old vulnerabilities. In *S&P*, 2026.

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

[40] N. R. Holtryd, M. Manivannan, and P. Stenström. Sok: Analysis of root causes and defense strategies for attacks on microarchitectural optimizations. In *EuroS&P*, 2023.

[41] G. Hu, Z. He, and R. B. Lee. Sok: Hardware defenses against speculative execution attacks. In *SEED*, 2021.

[42] J. Hur, S. Song, S. Kim, and B. Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *CCS*, 2022.

[43] IOVisor Project. uBPF: Userspace eBPF VM, 2015. URL: https://github.com/iovisor/ubpf.

[44] Jann Horn. Linux: eBPF Spectre v1 mitigation is insufficient, 2018. URL: https://project-zero.issues.chromium.org/issues/42450782.

[45] T. Jauch, A. Wezel, M. R. Fadiheh, P. Schmitz, S. Ray, J. M. Fung, C. W. Fletcher, D. Stoffel, and W. Kunz. Secure-by-construction design methodology for cpus: Implementing secure speculation on the RTL. In *ICCAD*, 2023.

[46] H. Jin, Z. He, and W. Qiang. Specterminator: Blocking speculative side channels based on instruction classes on risc-v. *TACO*, 2023.

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

[48] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*, 2022.

[49] D. Kästner, L. Mauborgne, and C. Ferdinand. Detecting spectre vulnerabilities by sound static analysis. 2019.

[50] kernel.org. Documentation: Document array_index_nospec - kernel version v4.16-rc1, 2018. URL: https://www.kernel.org/doc/Documentation/speculation.txt.

[51] kernel.org. Spectre Side Channels — The Linux Kernel documentation, 2025. URL: https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

[52] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC*, 2019.

[53] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[54] P. Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler, 2018.

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

[56] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[57] C. Lam. Alibaba/T-HEAD's Xuantie C910, February 2025. URL: https://chipsandcheese.com/p/alibabat-heads-xuantie-c910.

[58] C. Lam. Inside SiFive's P550 Microarchitecture, January 2025. URL: https://chipsandcheese.com/p/inside-sifives-p550-microarchitecture.

[59] E. Lazzeri, M. Colella, G. Furano, and L. Cassano. S4v: A benchmark suite of transient execution attacks for risc-v processors. In *DDECS*, 2025.

[60] A.-T. Le, T.-T. Hoang, B.-A. Dao, A. Tsukamoto, K. Suzaki, and C.-K. Pham. A real-time cache side-channel attack detection system on risc-v out-of-order processor. *IEEE Access*, 2021.

[61] F. Lin, W. Zhongfa, and H. Sasaki. Teapot: Efficiently Uncovering Spectre Gadgets in COTS Binaries. In *CGO*, 2025.

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

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

[64] LKML. x86/cpu/AMD: Make LFENCE a serializing instruction, 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e4d0e84e490790798691aaa0f2e598637f1867ec.

[65] LKML. RISC-V: Add mvendorid, marchid, and mimpid to /proc/cpuinfo output, 2022. URL: https://lore.kernel.org/lkml/20220620115549.1529597-1-apatel@ventanamicro.com/T/.

[66] LKML. Implement perf event mmap support in the SBI backend, 2023. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cc4c07c89aada16229084eeb93895c95b7eabaa3.

[67] LKML. riscv: Enable generic CPU vulnerabilites support, 2024. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0e3f3649d44bf1b388a7613ade14c29cbdedf075.

[68] LKML. riscv: Add ghostwrite vulnerability, 2025. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=4bf97069239bcfca9840936313c7ac35a6e04488.

[69] A. Louka, J. De Meulemeester, S. Keuchel, I. Verbauwhede, and J. Van Bulck. Physical memory please: Practical memory-aliasing attacks on RISC-V PMP. In *uASC*, 2026.

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

[71] Y. F. Majid Sabbagh. Secure speculative execution via risc-v open hardware design. In *CARRV*, 2021.

[72] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, et al. An open source fpga-optimized out-of-order risc-v soft processor. In *ICFPT*, 2019.

[73] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[74] H. Nemati, R. Guanciale, P. Buiras, and A. Lindner. Speculative leakage in arm cortex-a53. *arXiv:2007.06865*, 2020.

[75] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security*, 2020.

[76] OSCPU Team. NutShell, 2019. URL: https://github.com/OSCPU/NutShell.

[77] M. Patrignani and M. Guarnieri. Exorcising spectres with secure compilers. In *CCS*, 2021.

[78] A. Purnal, M. Bognar, F. Piessens, and I. Verbauwhede. Showtime: Amplifying arbitrary cpu timing side channels. In *AsiaCCS*, 2023.

[79] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. 2021.

[80] A. Randal. This is how you lose the transient execution war. *arXiv:2309.03376*, 2023.

[81] RISC-V Foundation. A more secure world with the RISC-V ISA, January 2018. URL: https://riscv.org/2018/01/more-secure-world-risc-v-isa/.

[82] RISC-V Foundation. *The RISC-V Instruction Set Manual, Vol. II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified*, 2019.

[83] S. Rüegge, J. Wikner, and K. Razavi. Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions. In *USENIX Security*, 2025.

[84] A. Sahraee. Specognitor: Identifying spectre vulnerabilities via prediction-aware symbolic execution. *arXiv*, 2022.

[85] Scaleway. Elastic Metal RV1 - The world's first RISC-V servers available in the cloud., 2023. URL: https://labs.scaleway.com/en/em-rv1/.

[86] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*, 2017.

[87] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[88] SiFive. SiFive statement on meltdown and spectre, January 2018. URL: https://web.archive.org/web/20251207154543/https://www.sifive.com/blog/sifive-statement-on-meltdown-and-spectre.

[89] SiFive, Inc. *SiFive P550 Core Complex Manual*, 2021. Version 21G2.01.00.

[90] SpinalHDL. NaxRiscv, 2024. URL: https://github.com/SpinalHDL/NaxRiscv.

[91] T-Head. openC910, 2021. URL: https://github.com/T-head-Semi/openc910.

[92] T-Head Semiconductor Co., Ltd. *XuanTie OpenC910 User Manual*, 2022.

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

[94] M. C. Tol, B. Gulmezoglu, K. Yurtseven, and B. Sunar. FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings. In *EuroS&P*, 2021.

[95] P. Turner. Retpoline: a software construct for preventing branch-target-injection, 2018. URL: https://support.google.com/faqs/answer/7625886.

[96] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[97] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *TOSEM*, 29(3):1–31, 2020.

[98] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *TSE*, 2019.

[99] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual, Vol. I: Unprivileged ISA, Version 20191213, 2019.

[100] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security*, 2021.

[101] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida. Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *USENIX Security*, 2024.

[102] S. Wiebing and C. Giuffrida. Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks. In *S&P*, 2025.

[103] J. Wikner and K. Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[104] N. Wistoff, G. Heiser, and L. Benini. fence. ts: Closing timing channels in high-performance out-of-order cores through isa-supported temporal partitioning. In *ApplePies*, 2024.

[105] N. Wistoff, M. Schneider, F. K. Gürkaynak, G. Heiser, and L. Benini. Systematic prevention of on-core timing channels by full temporal partitioning. *IEEE TC*, 2022.

[106] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser. Microarchitectural timing channels and their prevention on an open-source 64-bit risc-v core. In *DATE*, 2021.

[107] Y. Xiao, Y. Zhang, and R. Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

[108] W. Xiong and J. Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 2021.

[109] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[110] Y. Yarom and K. Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[111] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS*, 2019.

[112] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

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

[114] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind. Composable building blocks to open up processor design. In *MICRO*, 2018.

[115] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *USENIX Security*, 2023.

[116] Z. Zhang, Y. Liu, Y. She, A. I. Sanka, P. S. Y. Hung, and R. C. C. Cheung. Conboom: A configurable cpu microarchitecture for speculative covert channel mitigation. *Electronics*, 2025.

[117] Y. Zhu, W. Huang, and Y. Xiong. Place protections at the right place: Targeted hardening for cryptographic code against spectre v1. *USENIX Security*, 2025.

[118] J. Zomer and A. Sandulescu. Finding gadgets for cpu side-channels with static analysis tools, 2023. URL: https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md.

A RSB Size Measurement

Figure 6 shows the RSB size measurement for the tested processors, obtained by JIT-compiling nested call chains at increasing depths (Section 3.3). Once the RSB is exhausted, returns can no longer be predicted, slowing down execution.

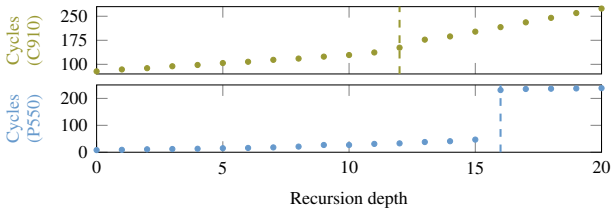

Figure 6: RSB size measurement for the C910/C920 and P550. After the RSB is exhausted, returns can no longer be predicted, slowing down execution.

B Speculation Window Analysis

The *speculation window* is the number of instructions that can be executed speculatively before the processor resolves the speculation, squashing transient instructions. This window is bounded by the reorder buffer size: 192 instructions on the C910/C920 and 92 instructions on the P550 [57, 58]. The effective speculation window depends on how quickly the mispredicted condition resolves, making window extension techniques necessary.

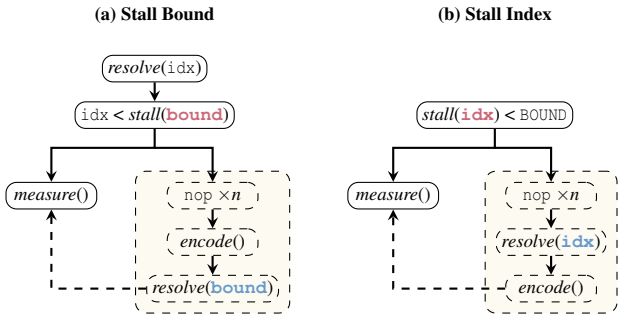

Figure 7: Speculation window measurement setup. The transient path executes n `nop`s followed by an encoding load. Variables are shown in red when stalled and in blue when resolved. (a) Prior work stalls the bound; since `idx` resolves before the branch, encoding occurs before speculation ends. (b) Our approach stalls the index instead; `idx` must resolve before encoding, enabling exploitation of immediate bounds.

Windowing gadgets [107] are instructions that retire slowly, stalling subsequent instructions and the time until speculation can be resolved. Prior work has shown that the speculation window for Spectre-PHT can be extended by flushing branch bound operands from the cache [107] (cf. Figure 7a).

However, our gadget analysis (Section 4.1) shows that many candidate Spectre-PHT gadgets use non-memory immediate bounds (e.g., `idx < IMM`). Therefore, we also stall the index operand instead (Figure 7b). By stalling the index, it resolves during speculation rather than before the branch, extending the window equivalently. This enables exploiting gadgets with immediate bounds, where stalling the bound is not possible.

To measure the speculation window, we JIT-compile tests that execute a trigger instruction followed by *n* `nop` instructions and a memory load (Figure 7). If the load executes before speculation is squashed, it leaves a cache timing side effect; by varying *n*, we determine the window size [107]. We test four configurations: a fault-based baseline using a null-pointer access, a branch with immediate operands where both index and bound resolve immediately, stalling the bound operand (prior work), and stalling the index operand (our approach). We use cache-based stalling, where the operand is loaded from uncached memory. For branch-based experiments, we mistrain the PHT to induce misprediction.

The fault-based baseline measures the speculation window when a null-pointer dereference triggers speculation. On the C910/C920, faults are detected within 60 instructions, while on the P550, the window extends to 93 instructions before the fault resolves. With immediate operands where both index and bound are cached, the branch resolves almost immediately, leaving only 7 instructions on C910/C920 and 1 instruction on P550 for speculative execution. Stalling the bound operand by placing it in uncached memory extends the window to 61 instructions on the C910/C920 and 87 instructions on the P550, confirming prior results on RISC-V. Crucially, stalling the index achieves an equivalent window (61 and 85 instructions respectively).

C Spectre BPF Gadget

Listing 1 shows the Spectre gadget that we exploit in Section 4. At a high level, the gadget tricks BPF code into speculatively dereferencing a BPF scalar. The code gets its arguments by reading them from BPF maps and is crafted to behave as follows: For the training phase, the value in `r7` is set to 1. Hence, the branch in line 12 evaluates to false and line 13 overwrites the attacker-controlled scalar (`r8`) with a valid pointer to a memory map (`r0`). To prevent the branch outcome from influencing the branch prediction of the target branch in line 40, we execute a chain of branches. To further increase the exploit's stability in case any of these chained branches collide with the target branch's PHT entry, these branches are crafted to follow the same branch direction as the mispredicted target branch. Lines 27 to 32 execute a slow operation with a data dependency on `r7` to slow down the retirement of the target branch; while we use division, any slow operation works [107]. During the training phase (`r7 != 3`), the target branch in line 40 loads the valid pointer (41), selects a bit offset and aligns it to a multiple of 4096 (lines 42 and 43), and eventually uses the result as an index into the map pointer stored in `r4` (line 44 and 45). If the target branch is mispredicted, which the attacker can reliably trigger using training runs, `r8` still contains the attacker-controlled scalar for the load in line 41. Thus, the code transiently uses an attacker-controlled value as index to `r9`.

```
1  // arguments
2  // r0: Valid pointer (BPF map pointer)
3  // r2: Target Pointer Bit Offset
4  // r7: Attack or Training
5  // r8: Target pointer (BPF scalar)
6  // r9: Pointer to covert channel bpf map
7  READ_ARGS_FROM_MEMORY_MAPS
8
9  // switch between training and attack phase
10 // if r7 != 3
11 //   r8 = r0
12 BPF_JMP_IMM(BPF_JEQ, BPF_REG_7, 3, 1),
13 BPF_MOV64_REG(BPF_REG_8, BPF_REG_0),
14
15 // spam branches to poison the history
16 // buffer.
17 // isolates the previous branch
18 // from the target branch
19 BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 1, 1),
20 BPF_MOV64_REG(BPF_REG_6, BPF_REG_6),
21 [...]
22 BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 1, 1),
23 BPF_MOV64_REG(BPF_REG_6, BPF_REG_6),
24
25 // slow-executing windowing gadget
26 // with data dependency on R7
27 BPF_MOV64_IMM(BPF_REG_5, 0x13371337),
28 BPF_MOV64_IMM(BPF_REG_6, 0x3),
29 BPF_ALU64_REG(BPF_DIV, BPF_REG_5, BPF_REG_6),
30 BPF_ALU64_IMM(BPF_AND, BPF_REG_5, 0x1),
31 BPF_ALU64_IMM(BPF_AND, BPF_REG_5, 0x2),
32 BPF_ALU64_REG(BPF_ADD, BPF_REG_7, BPF_REG_5),
33
34
35 // target branch to mistrain
36 // if r7 != 3:
37 //  r4 = *r8
38 //  r4 = (r4 << r2) & 0x1000
39 //  r9 = *(r9 + r4)
40 BPF_JMP_IMM(BPF_JEQ, BPF_REG_7, 3, 5),
41 BPF_LDX_MEM(BPF_DW, BPF_REG_4, BPF_REG_8, 0),
42 BPF_ALU64_REG(BPF_LSH, BPF_REG_4, BPF_REG_2),
43 BPF_ALU64_IMM(BPF_AND, BPF_REG_4, 0x1000),
44 BPF_ALU64_REG(BPF_ADD, BPF_REG_9, BPF_REG_4),
45 BPF_LDX_MEM(BPF_B, BPF_REG_9, BPF_REG_9, 0),
46
47 // exit(0)
48 BPF_MOV64_IMM(BPF_REG_0, 0),
49 BPF_EXIT_INSN(),
```

Listing 1: The BPF gadget exploited during our end-to-end exploit. The target branch is mistrained when `r7 != 3`. After training, a branch misprediction leads to a speculative type confusion in which an attacker-controlled BPF scalar is dereferenced.

D Artifact Appendix

D.1 Abstract

This artifact supports the Open Science scope stated in the paper. It contains the unified Spectre proof-of-concept framework, microarchitectural measurement tools, speculation-barrier tests, Smatch/CodeQL gadget-scanning scripts, cross-architecture mitigation-diffing code, branchless uBPF dispatch prototype and benchmark, optional SPEC CPU 2017 build setup, cache/timer primitives, BPF gadget-emitter source, and Linux kernel patches.

D.2 Description & Requirements

Full reproduction requires the evaluated RISC-V hardware.

D.2.1 Security, privacy, and ethical concerns

Several experiments show Spectre-style leakage, load kernel modules and change BPF settings. Run them only on isolated systems owned or explicitly authorized by the evaluator. The artifact does not process personal data or contact third-party systems.

D.2.2 How to access

Download the latest Zenodo release at `https://doi.org/10.5281/zenodo.18452278`. The relevant directories are:

- `spectre-poc-framework/`: 13 Spectre variants plus runner.
- `microarch-measurements/`: RSB-size and speculation-window tests.
- `barrier-test/`: candidate speculation-barrier tests.
- `gadet-scanning/`: Smatch and CodeQL kernel gadget scanning.
- `gadet-scanning/mitigation_diffing/`: semantic mitigation-diffing tool.
- `specbuild/`: SPEC CPU 2017 cross-build setup for mitigation benchmarks.
- `branchless-jit-dispatch/`: branchless dispatch prototype and modified uBPF.
- `cacheutils.h`, `counter-thread-timer/`: cache primitives and timer benchmark.
- `bpf-spectre-exploit/`: BPF Gadget Emitter: BPF bytecode to reliably emit the type confusion gadget and leak from a controlled kernel address.
- `kernel-patches/`: contributed RISC-V Spectre-v1 patches.

D.2.3 Hardware dependencies

The main hardware targets are T-Head Xuantie C910/C920 and SiFive P550 systems with root access. BTB experiments require BTB prediction to be enabled in firmware, which is not the case in the stock configuration of either board. P550 cache-eviction experiments require huge pages:

```
sudo sysctl -w vm.nr_hugepages=64
```

On some cores, userspace access to the cycle counter has to be enabled explicitly, otherwise the timed experiments abort with an illegal-instruction fault:

```
sudo sysctl -w kernel.perf_user_access=2
```

BPF experiments require a Linux target with BPF maps and JIT support.

D.2.4 Software dependencies

The target-board experiments (Basic Test, E1–E3, and E5) require a C/C++ toolchain, `make`, `cmake`, `bc`, Python 3, and the kernel/BPF settings listed per experiment. All experiments that are run on non RISC-V hardware (gadget scanning, cross-compiling SPEC CPU) are bundled in a Docker container that installs all the required dependencies.

D.2.5 Benchmarks

We use the `bpf-benchmark` to evaluate performance of our mitigated BPF interpreter. The proprietary SPEC CPU 2017 benchmark is optional. The `specbuild/` helper includes tooling to cross-compile the benchmarks on a fast machine and then run them on the target.

D.3 Set-up

D.3.1 Installation

Obtain the artifact from the provided Zenodo link and unpack it.

D.3.2 Basic Test

On a RISC-V target, run:

```
cd spectre-poc-framework
python3 run_all_tests.py --group baseline \
  --runs 1 --iterations 100
```

The runner should detect the platform, build the baseline binary, and print precision/recall metrics (compare to Table 2).

D.4 Evaluation workflow

D.4.1 Major Claims

(C1) C910/C920 and P550 are vulnerable to major Spectre variants; reproduced by E1.

(C2) The BPF source emits the paper's speculative type-confusion gadget and shows that the leakage can be reproduced; reproduced by E2.

(C3) Speculation barriers, RSB size, speculation windows, cache channels, and counter-thread timing are processor-dependent; reproduced by E3.

(C4) Gadget scanning and mitigation diffing reveal missing RISC-V kernel hardening; reproduced by E4.

(C5) The branchless uBPF dispatch mitigation can be built and benchmarked with bundled BPF programs; reproduced by E5.

(C6, optional) The SPEC CPU 2017 mitigation setup reproduces the reported mitigation overheads given a licensed SPEC tree, a target sysroot, and dedicated RISC-V hardware; reproduced by optional E6.

D.4.2 Experiments

(E1) Spectre variant framework [15 human-minutes + 1-2 compute hours per board].

```
cd spectre-poc-framework
python3 run_all_tests.py --group all --runs 30 \
  --csv results.csv
```

Compare the CSV to Table 2 in the paper. The test runner also outputs human-readable logs with all the information. C910/C920 should show high recall for most PHT, RSB, and STL variants. P550 should show strong PHT sa-ip, RSB, and STL results with lower cross-address recall. The BTB variants only leak once BTB prediction is enabled in firmware, so they show no leakage on stock boards.

(E2) BPF proof-of-concept [20 human-minutes].

```
cd bpf-spectre-exploit
make clean && make exploit-complete
sudo sysctl -w kernel.perf_event_paranoid=0 \
  kernel.perf_user_access=2 \
  net.core.bpf_jit_enable=1 vm.nr_hugepages=1000
sudo ./exploit-complete
```

Expected result on the C910/C920: the program leaks the kernel banner. Additionally one can inspect the emitted RISC-V bytecode from the BPF JIT by setting:

```
echo 2 | sudo tee /proc/sys/net/core/bpf_jit_enable
```

This command dumps a lot of data to the kernel log.

(E3) Microarchitectural measurements [20 human-minutes + 1 compute-hour per board].

```
cd barrier-test && make
sudo ./barrier_test && sudo ./barrier_test_rsb
cd ../microarch-measurements/rsbsize-test
make && ./test
cd ../speculation-window
make && sudo ./main > spec-window.csv
cd ../../counter-thread-timer
sudo sysctl -w kernel.perf_event_paranoid=0 \
  kernel.perf_user_access=2
make && COUNTER_CORE=1 taskset -c 0 ./bench
```

Expected results: barrier tests report low hit rates for effective barriers. The RSB-size test shows a cliff near 12 entries on C910/C920 and 16 on P550. The counter-thread timer reports approximately cycle-scale resolution on C910/C920.

(E4, local) Kernel gadget scanning, mitigation diffing, and patches [1 human-hour + 1-2 compute-hours].

```
# Build the analysis image.
cd gadet-scanning
docker build -t riscv-gadget-scan .

# Get the kernel source.
git clone --depth 1 --branch v6.6 \
  https://github.com/torvalds/linux.git ~/linux-6.6
CACHE=mitigation_diffing/.embedding_cache
mkdir -p results .hf-cache "$CACHE"

# Run Smatch.
docker run --rm \
  -v ~/linux-6.6:/kernel \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./run-analysis.py linux-6.6 /kernel --tool smatch

# Run CodeQL. Creates a reusable database in
# results/linux-6.6/codeql-db/.
docker run --rm \
  -v ~/linux-6.6:/kernel \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./run-analysis.py linux-6.6 /kernel --tool codeql

# Run mitigation diffing.
docker run --rm \
  -v ~/linux-6.6:/kernel:ro \
  -v "$PWD/.hf-cache:/root/.cache" \
  -v "$PWD/$CACHE:/opt/gadet-scanning/$CACHE" \
  riscv-gadget-scan bash -lc "cd mitigation_diffing \
    && mitigation-diffing /kernel -m mitigations.csv"

# Compare tools (Smatch vs CodeQL).
docker run --rm \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./compare-tools.py linux-6.6
```

Expected results: the scanner emits kernel gadget reports; `mitigation-diffing` reports missing RISC-V mitigation counterparts for the listed cross-architecture mitigation sites. This should reproduce the findings in Section 4.1, where CodeQL and Smatch do not find the RISC-V specific gadgets that our mitigation diffing approach found.

(E5) Branchless uBPF dispatch benchmark [30 human-minutes + benchmark compute time].

```
cd branchless-jit-dispatch
make && ./dispatch
cd ubpf
cmake -S . -B build_baseline -DUBPF_ENABLE_TESTS=ON \
  -DCMAKE_BUILD_TYPE=Release
cmake --build build_baseline --target ubpf_test --parallel
cmake -S . -B build_mitigated -DUBPF_ENABLE_TESTS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_C_FLAGS="-DUBPF_JIT_DISPATCH"
cmake --build build_mitigated --target ubpf_test --parallel
./benchmark.sh build_baseline/bin/ubpf_test \
  build_mitigated/bin/ubpf_test \
  ../bpf-benchmark/bpf_progs 50
```

Expected results: `dispatch` runs the branchless direct-`jal` prototype. By inspecting the binary with `objdump -d` it can be confirmed that no indirect branches are used for the dispatch. The uBPF benchmark prints CSV rows `program,baseline_ms,mitigated_ms,overhead` which should reproduce the numbers in Section 5.2.

(E6, optional) SPEC CPU 2017 mitigation setup [2 human-hour + benchmark compute time]. Run it with a licensed SPEC CPU 2017 installation and a target sysroot.

```
cd specbuild
```

```
python3 -m venv .venv
. .venv/bin/activate
python3 -m pip install --upgrade pip pyyaml
docker build -t specbuild docker/
./fetch-sysroot.py <riscv-host> \
  ./sysroots/<riscv-host>-sysroot
# Place SPEC CPU 2017 at cpu2017/spec_cpu/
# or edit spec-build.yaml.
python3 build.py -l
python3 build.py -t <target> --size test \
  --benchmarks=505.mcf_r
scp packages/<target>-*.tar.gz <riscv-host>:~
# On <riscv-host>: unpack one package and run:
#   ./run.sh 505.mcf_r --csv spec.csv
```

Expected results: `specbuild` lists configured targets and fetches a sysroot from the target machine. The sysroot allows to cross compile the SPEC benchmarks for the target machine. The LLVM mitigation profiles additionally require the compiler bundle described in `specbuild/compilers/README.md`. The SPEC benchmarks should reproduce the overheads listed in Figure 3 of the paper.

D.5 Notes on Reusability

The artifact can be reused by testing all or a subset of the spectre variants on new RISC-V hardware. This requires adapting timers and the cache side channel to the new hardware, but should be much simpler than building a proof-of-concept from scratch. Our mitigation diffing can be used to map code sites from a kernel of one architecture to another. We believe that this is a useful tool not only for Spectre mitigations, but working with cross-architecture kernel code in general. The branchless JIT dispatch prototype could be extended into a more general mitigation by implementing it as part of a compiler pass.

D.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.