RLeak: Hybrid Reinforcement Learning Guided Fuzzing Framework for RISC-V Vulnerability Timing Side-Channel Discovery

Nathanaël Simon[0009−0000−4796−6556], Maria Mushtaq[0000−0003−0046−2582], and Ludovic Apvrille[0000−0002−1167−4639]

Télécom Paris - IP Paris, Palaiseau, France
firstname.lastname@telecom-paris.fr

Abstract. Proactive timing side-channel vulnerability discovery and assessment is essential as microarchitectural complexity continuously expands the attack surface of modern Systems-on-Chip. Existing approaches —formal verification and fuzzing— face a fundamental tension: formal methods offer strong guarantees but do not scale to full processor complexity, while unguided fuzzing lacks the directed coverage needed to surface subtle timing anomalies. In this work, we present *RLeak*, a black-box hybrid framework that resolves this tension by separating concerns across two levels: a Reinforcement Learning agent operates at the *strategy level*, conditioning on the current instruction sequence to select instruction-class patterns correlated with timing anomalies, while a fuzzer operates at the *generation level*, efficiently instantiating concrete sequences within RL-selected categories. This separation enables scalable, directed exploration without requiring white-box processor access. Trained on gem5 RISC-V simulation as a black-box environment, *RLeak* rediscovers five known vulnerability classes and characterizes operand-dependent timing behavior across 19 RV64GC instructions, 11 not previously reported as leaking on open RISC-V cores. To assess simulation-to-silicon transferability, we validate our findings across Register-Transfer Level simulations of three open-source RISC-V cores (Rocket, BOOM, CVA6), confirming two novel vulnerability classes—Store-Buffer pressure and Pipeline-Drain leakage—and a variant of the Spectre-STC FU-Contention pattern. A bare-metal covert-channel proof-of-concept on Rocket demonstrates exploitability of identified vulnerabilities, with bit error rates ranging from 0% to 16.6% across all five oracles. *RLeak* provides a scalable black-box methodology for systematic hardware security assessment, offering early-stage defense coverage against microarchitectural timing side channels.

Keywords: Side-Channel Attacks · Vulnerability Assessment · Reinforcement Learning · Fuzzing · RISC-V · gem5 · Hardware Security

1 Introduction

Performance features such as speculative and out-of-order execution and complex cache hierarchies expand the attack surface for Timing Side-Channel Attacks

(TSCAs) [16,20], and novel TSCAs continue to emerge despite mitigations [15]—motivating proactive assessment at design stage. Yet formal verification suffers state-space explosion [9], while fuzzing is constrained by generation strategy bias and combinatorial scalability [31, 41]. In this work, we explore an hybrid approach guiding fuzzing with Reinforcement Learning (RL). Where existing RL-fuzzing hybrids use RL to *mutate* fuzzer outputs [13, 42, 43], we use RL to *guide generation*: the agent sequentially selects the next instruction category or mutation conditioned on the current sequence and observed timing, delegating concrete instantiation to the fuzzer. This aims to target vulnerabilities emerging from *ordered instruction compositions under accumulated microarchitectural state*, a space combinatorial in sequence length that per-instruction enumeration cannot cover, and over which the agent must allocate a finite fuzzing budget under sparse feedback, without Register-Transfer Level (RTL) access or hardware performance counters.

We present *RLeak*[1], a hybrid RL-guided fuzzing framework for TSCA assessment in RISC-V processors, contributing:

- **Hybrid RL-guided fuzzing:** RL learns sequence-level patterns conditioned on the partial sequence and timing feedback correlating with timing anomalies; fuzzing instantiates concrete sequences within them.
- **Scalable black-box assessment:** via gem5 training, *RLeak* explores the full RISC-V ISA without manual heuristics, domain-specific priors, or source-level instrumentation.
- **RTL validation across three cores:** candidates are confirmed via RTL simulation on Rocket, BOOM, and CVA6, demonstrating generalization from cycle-accurate simulations to microarchitectural designs.
- **Vulnerability discovery and enumeration:** *RLeak* rediscovers five known vulnerability classes and enumerates operand-dependent timing across 19 RV64GC instructions, 11 not previously reported on open cores. It further identifies three timing mechanisms—two novel (Store-Buffer pressure, Pipeline-Drain) and a RISC-V Spectre-STC variant (FU-Contention), the novel ones being interaction-level channels not reported by prior RISC-V fuzzing frameworks (Table 6)—realized as five validated oracles and working covert channels on Rocket.

This paper is organized as follows: Section 2 covers background; Section 3 the *RLeak* framework; Section 4 results; Section 5 limitations and future work; Section 6 concludes.

2 Background & Related Work

2.1 Side-Channel Attacks

Timing side-channel attacks target shared processor resources—caches, branch predictors, execution units—measuring timing variation to deduce memory-access

[1] Source code and Artifacts available at [1] and [2]

patterns and data-dependent operations [16, 19, 20, 26, 44]: secret-dependent behavior creates observable timing differences (e.g. cache hits faster than misses). Architectural side channels are ISA instruction sequences producing unintended information flow from microarchitectural to architectural state [37].

2.2 Automated Vulnerability Assessment

Formal Methods Formal verification [14] has identified cache-based [40] and RISC-V transient-execution [12] vulnerabilities and new attack sequences on commercial processors [46]. It uniquely offers mathematical guarantees, but state-space explosion and the need for domain expertise prevent it from scaling to full processor complexity.

Fuzzing Fuzzers detect timing anomalies via execution monitoring, using random generation [37,41], mutation with timing-variation metrics [31], or contract-based equivalences [27, 28]. On x86, *Osiris* [41] uses an Finite State Machine fuzzer and *Revizor* [27,28] a contract-based approach; *ExfilState* [37] is a timer-free ARM fuzzer. For RISC-V, *SIGFuzz* [31] (CTD clustering) and *WhisperFuzz* [7] (Micro-Event Graphs) operate on white-box RTL, together finding 15 new vulnerabilities; *RISCover* [36] fuzzes differentially across physical boards but needs multiple platforms. A limitation cuts across all: *generation strategy bias*—templates surface only vulnerabilities fitting them, while random fuzzers avoid the bias but are limited to 3–5 instruction sequences.

Reinforcement Learning RL learns a policy through sequential, reward-driven interaction with no labeled data [29], suiting decision-making under uncertainty with quantifiable feedback. *AutoCAT* and *MACTA* [10,23] model cache timing as a victim/attacker game but stay in white-box simulated caches, targeting only memory-access leakage; *μRL* [38] explores the x86 ISA via hardware performance counters but faces the same action-space scalability limits.

RL-fuzzing hybrids *GenHuzz* [42], *RLFuzz* [13], and *HFL* [43] use RL as a *mutation engine* to refine fuzzer-generated test cases, maximizing RTL coverage via differential comparison with a golden reference model. All require open-source RTL access, and their coverage-maximization objective prevents a vulnerability-focused exploration. *RLeak* inverts this relationship: rather than using RL to refine fuzzer outputs, RL acts as a *sequential generation strategy*, selecting the next instruction category or mutation conditioned on the current partial sequence and observed timing, and delegating concrete sequence instantiation to the fuzzer.

Gaps and Positioning

Three gaps remain unaddressed. First, **generation strategy independence**: existing approaches rely on hypothesis-driven templates or manually crafted patterns, limiting their ability to generalize; RL-fuzzing hybrids [13, 42, 43] only

partially address this, since they apply RL as a *mutation engine* acting on already-instantiated test cases rather than guiding generation itself. Second, **execution environment balance**: existing work forces a choice between speed (real hardware) and depth (RTL), with no framework exploiting intermediate environments like gem5 that offer microarchitectural timing resolution without RTL's scalability constraints—and the hybrids further couple training to a specific design by requiring open-source RTL in the reward loop. Third, **TSCA-specialized optimization**: coverage-based rewards drive exploration broadly rather than toward timing side-channel candidates specifically; the differential coverage objective of prior hybrids frameworks falls short in regard of the complexity of the attack surface, spending resources on test cases with no side-channel relevance.

In this work, we target leakage-relevant behavior that emerges from *interactions across ordered sequences under accumulated microarchitectural state*, such as functional-unit contention, store-buffer/AMO interplay, and pipeline-drain effects (see Table 6 and explicit threat model definition in Section 3). The space of such ordered compositions is exponential in sequence length, so the problem is not classifying which instructions are slow, but strategically allocating a finite fuzzing budget across a vast compositional space under sparse reward—a sequential learning problem rather than an enumeration one. Consistent with this, the set of new mechanisms RLeak highlights, including the three interaction-level ones, is not reported by SIGFuzz [31] or WhisperFuzz [7] despite their coverage of the same cores, indicating that a state-conditioned sequential strategy reaches regions that existing template- and RTL-structure-driven generation frameworks do not.

RLeak addresses these gaps by splitting concerns on two levels. The RL agent works over a compact, class-level action vocabulary—preserving scalability—but conditions on the current partial sequence and observed timing, learning ordered, state-dependent compositions correlated with timing anomalies. The fuzzer then instantiates concrete sequences within those patterns. This separation allows RL to operate over a compact, high-level action space while retaining fuzzing's efficiency—escaping generation strategy bias without requiring RTL access or hardware performance counters. The key insight is that fuzzing and RL limitations are *mutually compensating*: RL provides strategic guidance that random fuzzing lacks; fuzzing provides execution diversity that abstract RL policies cannot generate alone.

3 Methodology

This section presents *RLeak*, a hybrid RL-guided fuzzing framework for TSCA vulnerability assessment, using gem5 as a black-box training environment and a multi-stage RTL validation pipeline. As illustrated in Figure 1, the framework training loop comprises four components: an RL agent acting as a high-level instruction-class selector (§3.1), a fuzzing-based sequence generator guided by the agent (§3.2), a gem5-based training environment (§3.3), and a timing

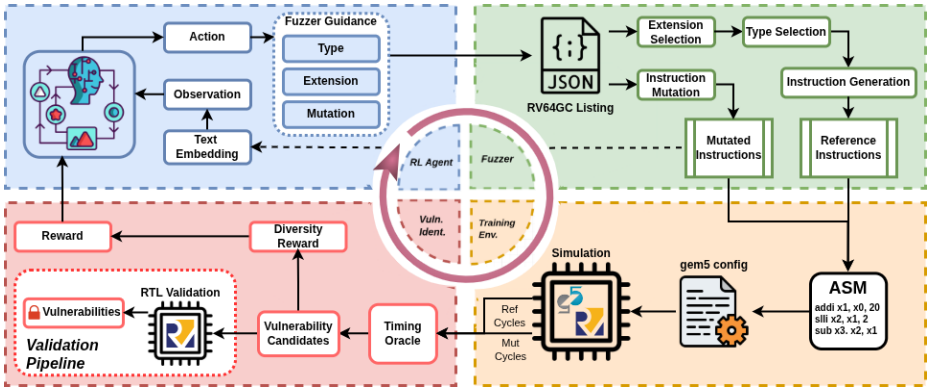


Fig. 1: Overview of *RLeak*: RL agent selects instruction-class patterns; fuzzer instantiates concrete sequences; gem5 evaluates timing; reward steers the agent toward vulnerability candidates, which are then validated through RTL simulation.

oracle computing the reward signal (§3.4). Identified candidates are then validated through RTL simulation on open-source RISC-V cores (§3.5). The three gaps of Section 2 map onto these components: *generation strategy independence* is addressed by the learned agent policy and fuzzer instantiation (§3.1, §3.2); *execution environment balance* by the gem5 training environment (§3.3); and *TSCA-specialized optimization* by the timing-oracle reward (§3.4).

Threat Model: We align with MITRE terminology [24, 25] and distinguish three levels: a **weakness** is a microarchitectural feature (e.g., shared caches, branch predictors) that can enable unintended information flows but does not by itself leak secrets; a **vulnerability** is a specific instruction sequence or execution pattern that exploits a weakness to expose internal processor state (e.g., CVE-2017-5753, CVE-2018-3639); an **attack** is an end-to-end exploitation strategy that leverages one or more vulnerabilities to extract sensitive information. *RLeak* targets vulnerability discovery: generating RV64GC instruction sequences that produce observable, distinguishable timing variations. While we provide PoC of covert channels from our findings, we leave end-to-end attack construction in realistic deployment scenarios to future work [15].

Scope of the black-box design. The agent learns solely from timing observations, without RTL source, design instrumentation, or hardware performance counters of available CPUs—unlike RTL-coupled hybrids [42, 43] or HPC-based RL [38]. RLeak deliberately performs *discovery* under gem5's deterministic, resettable cycle-level timing to isolate operand- and sequence-dependent microarchitectural behavior from system-level noise, then confirms candidates under cycle-accurate RTL. Validating the surfaced mechanisms on physical boards under realistic noise is a complementary measurement problem that we leave to future work (§5).

We target the **RV64GC** instruction set—the de facto standard for available RISC-V boards and cores (CVA6, SiFive U74, XuanTie C906) [32]—combining the base 64-bit integer ISA with compressed and floating-point extensions. Specialized extensions such as Zkt, which mandates data-independent execution latency, are excluded as software mitigations, not removing the leakage root-causes, contradicting our assessment goals. Training within gem5 SE mode excludes privileged and OS-interacting instructions, yielding a working set of **175 instructions** across Immediate, Register, Store, Branch, Upper-Immediate, and Jump-Type categories.

3.1 Agent

The RL agent learns high-level patterns correlating with timing vulnerabilities, guiding the fuzzer toward promising instruction classes. We use Proximal Policy Optimization (PPO) via Ray RLlib [18] with OpenAI Gymnasium [39]; Table 1 summarizes the design. We build on mature components (PPO, SIGFuzz oracles) so reliability is not in question; the novelty is the *vulnerability-specialized, diversity-regularized, black-box reward* (§3.4) redirecting a standard policy loop from coverage toward timing-leakage discovery.

At each step the agent emits an action (e, t): if $e \in [1, N_E]$ it selects an extension whose type t the fuzzer appends to the current sequence; if $e = N_E{+}1$, the last added instruction is instead mutated (operand change or NOP insertion). N_E is the number of RISC-V extensions (RV64I, M, F, D) and N_T the instruction types per extension. Semantic embeddings place similar instructions (e.g. `ADD`/`ADDI`) in proximate regions, enabling policy transfer across opcodes without per-instruction symbols.

The reward combines the two oracle violation counts (N_{pr1}, N_{pr2}; §3.4), a diversity term R_{div}, and a step penalty, -0.02 in our experiments. R_{div} discounts repeated similar patterns, avoiding exploitation collapse without discarding genuine anomalies; using this black-box vulnerability-specialized reward rather than white-box coverage [42, 43] keeps exploration vulnerability-first rather than toward benign sequences.

Table 1: RLeak RL Agent Design

Component	**Design**
Algorithm	PPO (Ray RLlib) and Gymnasium environment
Action space	Multi-discrete (e, t): extension $e \in [1, N_E{+}1]$, type $t \in [1, N_T]$; $e{=}N_E{+}1$ triggers mutation
Observation	$N \times 1024$ matrix; each instruction encoded via Qwen3-Embedding-0.6B [47]; $N{=}10$ max sequence length
Reward	Reward $= N_{pr1} + N_{pr2} + R_{div} - 0.1 \times$ step, where $R_{div} = \left(\frac{1}{n}\sum_{i=1}^{n}\frac{1}{1+C_i}\right) \times w_{div}$ penalizes repeated extension/type, timing, transition, and similarity patterns; active only when $N_{pr1}{+}N_{pr2}{>}0$

3.2 Fuzzing-Based Generation

From (e, t), the fuzzer selects a matching RV64GC instruction and generates valid operands (register dependencies, immediate ranges, alignment). "Mutation" has two senses: at *every* step the sequence runs in two forms—a *reference* and a *mutated variant* differing by one controlled change—so the oracles of §3.4 can attribute timing differences to it; separately, the agent's *mutation action* ($e = N_E{+}1$) targets the last added instruction instead of appending. The mutation applies one of two strategies, randomly selected by the fuzzer: **NOP substitution** (`addi x0, x0, 0`, removing computation while preserving length [31]) or **operand mutation** (new valid operands for the same opcode [7]). Table 2 provides examples of the generated sequences.

3.3 Training Environment

Sequences are compiled to bare-metal C (RISC-V GCC, optimizations off). We use **gem5 V24.1.0.0** [22] in SE mode with the O3 CPU: deterministic cycle-level timing with enough throughput for RL convergence, unlike real hardware (noisy, opaque) or RTL (too slow). This determinism deliberately isolates operand- and sequence-dependent timing from real-silicon noise (§5). Reference and mutated sequences run in separate instances for clean state. Per run we record cumulative timestamps c_i after each instruction i via bracketed `rdcycle` pairs, deriving per-instruction times $t_i = c_i - c_{i-1}$.

3.4 Vulnerability Identification

Let $S = \langle I_1, \ldots, I_N \rangle$ be a sequence and m the index of the mutated instruction (where reference and mutated variants differ). With cumulative timestamps c_i and per-instruction times $t_i = c_i - c_{i-1}$ (superscripts ref/mut), we check two constant-time violations adapted from [31]. Property 1: instructions before I_m keep their individual timing (the mutation must not perturb earlier instructions):

$$\forall\, i < m : t_i^{ref} = t_i^{mut} \tag{1}$$

Property 2: instructions after I_m preserve their timing relative to it (equal cumulative offset from I_m in both runs):

$$\forall\, i > m : c_i^{ref} - c_m^{ref} = c_i^{mut} - c_m^{mut} \tag{2}$$

A Property 1 violation indicates the mutation perturbed preceding instructions; a Property 2 violation, that it altered downstream timing—either exposing operand- or data-dependent timing flow. N_{pr1} and N_{pr2} count instructions violating each property in S; those with $N_{pr1} + N_{pr2} > 0$ are flagged. For per-instruction analysis (Section 4) we also compute the Commit Time Difference:

$$\mathrm{CTD} = t_m^{ref} - t_m^{mut_{\mathrm{nop}}} \tag{3}$$

Table 2: Examples of instruction sequences generated by RLeak, and their resepective measurements in gem5. Three mechanisms produced validated RTL oracles (see Table 3); one (BTB state leakage, n=1) was discarded as a gem5 simulation artefact.

Mechanism	Idx	Instruction	Ref (cy)	Mutation	Mut (cy)	Δcy	Microarchitectural Root Cause (gem5 O3)	RTL Oracle
Fence.i Pipeline Drain n=22	[0]	`fcvt.lu.s t0, ft0`	4	-	4	-	FP convert — pipeline baseline	Oracle **PD** — Rocket BER=0.100
	[1]	`sraw t1, t2, t3`	2	-	2	-	Integer shift — baseline	
	[2]	`flw ft1, 0(sp)`	4	-	4	-	FP load — baseline	
	[3]	`fcvt.s.l ft2, t4`	6	-	6	-	FP convert — baseline	
	[4]	`fcvt.lu.s t5, ft3`	4	-	4	-	FP convert — baseline	
	[5]	`[lr.d t6, (a0)]`	8	→ `nop`	2	-	**Root cause**: `lr.d` acquires load reservation, dirtying pipeline state	
	[6]	`amoxor.d a1, a2, (a3)`	8	-	8	-	AMO — unaffected downstream	
	[7]	`amoand.d a4, a5, (a6)`	6	-	6	-	AMO — unaffected downstream	
	[8]	`fence.i`	26	-	2	**−24**	**Timing oracle**: Drain cost ∝ pipeline state left by LR/AMO chain	
MUL/DIV FU Contention n=9	[0]	`sraiw x21, x25, 22`	2	-	2	-	Arithmetic shift — baseline	Oracles **FU-3**, **FU-5** — Rocket: MARG. BOOM: BER=0.000
	[1]	`fcvt.d.l f21, x23`	6	-	6	-	FP convert — baseline	
	[2]	`fld f13, 16(x8)`	4	-	4	-	FP load — baseline	
	[3]	`mulw x24, x25, x27`	4	-	4	-	Multiply — occupies shared MUL/DIV functional unit	
	[4]	`sllw x21, x25, x5`	2	-	2	-	Shift — unaffected	
	[5]	`[amoadd.d x16, x5, (x8)]`	8	→ `nop`	2	-	**Root cause**: AMO creates store-buffer pressure stalling subsequent division	
	[6]	`addw x7, x8, x5`	2	-	2	-	Integer add — unaffected	
	[7]	`fcvt.s.lu f5, x8`	6	-	6	-	FP convert — unaffected	
	[8]	`divw x13, x18, x13`	34	-	8	**−26**	**Timing oracle**: Division stalls awaiting MUL/DIV FU vacated by `mulw`	
AMO Store-Buffer Pressure n=73	[0]	`amomax.w x19, x5, (x5)`	8	-	8	-	Atomic max — creates store-buffer entry	Oracles **SB-1**, **SB-5** — Rocket: MARG. BER=0.13–0.17
	[1]	`fcvt.d.lu f21, x23`	6	-	6	-	FP convert — unaffected	
	[2]	`sb x6, 0(x6)`	2	-	2	-	Store byte — unaffected	
	[3]	`[amoor.d x16, x5, (x8)]`	8	→ `nop`	2	-	**Root cause**: Removing AMO drains store-buffer pressure on cache line	
	[4]	`fld f15, 16(x8)`	28	-	22	**−6**	**Timing oracle**: FP load latency reflects residual AMO store-buffer state	
	[5]	`fmadd.s f26, f21, f5, f8`	6	-	6	-	FP multiply-add — unaffected downstream	
Branch Pred. BTB State *discarded* n=1	[0]	`c.jalr x31`	2	-	22	**+20**	**Apparent leak**: BTB/RAS state altered by prior AMO operands (backward causality)	*Not replicated in RTL* — *gem5 artefact*
	[1]	`[amoor.w x19, x5, (x5)]`	47	→ `x18, x8, (x6)`	31	−16	**Root cause**: AMO operand change alters branch-target address state	
	[2]	`mulh x24, x25, x27`	26	-	26	-	Multiply-high — unaffected	
	[3]	`fld f15, 24(x6)`	24	-	24	-	FP load — unaffected	

3.5 Validation Pipeline

Flagged candidates pass through a two-stage validation pipeline.

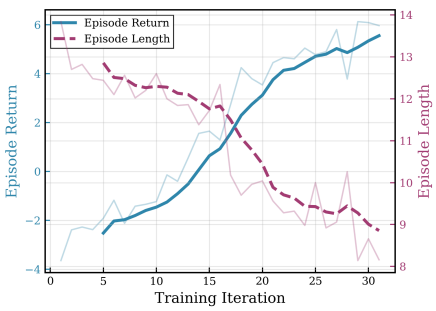
Stage 1 — gem5 Confirmation. Each candidate is (i) reproduced 100 times (90% threshold for determinism), (ii) reduced to its minimal timing-preserving instruction sequence, (iii) evaluated with a blind F-Score trial to quantify attacker distinguishability, and (iv) clustered via K-Means over instruction type, operand pattern, F-Score, and sequence length.

Stage 2 — RTL Validation. Confirmed candidates are re-executed under Chipyard [4] and Verilator [34] on three open-source RISC-V cores: *CVA6* (6-stage in-order) [45], *Rocket* (5-stage in-order) [5], and *BOOM* (out-of-order, multiple variants) [48]. Finally Candidates reproduced in RTL are manually escalated to covert-channel proof-of-concept development, constructing timing oracles from the identified mechanisms to confirm exploitability.

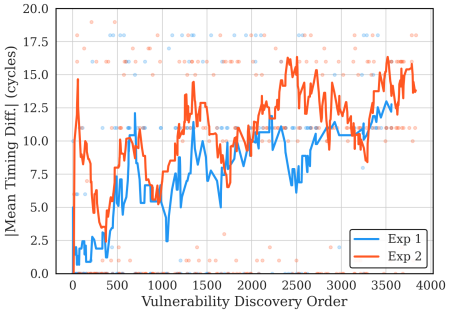
4 Results

We trained *RLeak* in a containerized environment on two Intel Xeon Gold 6130 (16 cores, 2.1 GHz), 64 GB DDR4, running Ubuntu 22.04, with five parallel learners and two environments each (10 concurrent gem5 simulations). Two experiments ran for 24 hours each (48 hours total, 30 training iterations each). Figure 2 shows the training dynamics: episode return increases and episode length decreases over iterations, confirming policy convergence, while growing absolute timing differences among discovered candidates confirm that *RLeak* improves candidate quality over time rather than simply accumulating more violations.

Results are organized in three phases summarized in Table 3: gem5 candidate analysis (Phase 1), RTL validation (Phase 2), and covert-channel PoC (Phase 3).



(a) Mean cumulative reward and episode length over training iterations.



(b) Absolute mean timing difference of discovered candidates over training.

Fig. 2: Training dynamics of *RLeak*: policy convergence (a) and improving candidate quality (b) confirm that the RL agent learns generalizable timing patterns rather than overfitting to a fixed vulnerability class.

4.1 Rediscovery of Known Vulnerabilities

RLeak rediscovers all major vulnerability classes previously reported by SIGFuzz [31] and WhisperFuzz [7] on open RISC-V cores: *ORC*, *Spectre-STC*, *Store Conditional*, *DIV/DIVU*, and `LD` timing channels (Table 6). Prior work characterized DIV timing primarily through division-by-zero and coarse latency differences; the finer value-dependent behavior we attribute to quotient bit-width is treated separately in C1 below. The six compressed-instruction vulnerabilities reported by WhisperFuzz on CVA6 (`C.ADD`, `C.SUB`, `C.AND`, `C.OR`, `C.XOR`, `C.MV`) were not rediscovered, as gem5 does not model CVA6-specific implementation details—an identified limitation of simulation-based training discussed in Section 5.

4.2 Newly Discovered Vulnerabilities

Operand-Dependent Timing (CTD Analysis) The three timing families below—value-dependent division, cache-address load timing, and write-allocate store timing—are well-established microarchitectural mechanisms; cache hit/miss latency and operand-dependent divider latency are precisely what motivates constant-time programming and the Zkt extension [32]. Our contribution here is to demonstrate that an oracle-driven, RL-guided search rediscovers them automatically, without manual heuristics or templates, and *generalizes* them into a concrete per-instruction, per-core map: several specific instructions are, to our knowledge, not previously reported as leaking on open RISC-V cores. *RLeak* identifies three classes of operand-dependent timing behavior, validated across all 8 RTL cores, detailed in Table 4.

C1 — DIV/REM value-dependent timing. All eight division and remainder instructions exhibit latency proportional to quotient bit-width, due to

Table 3: Summary of timing side-channel vulnerabilities discovered by the RL agent through the different validation phases.

Category	Count	% of clean	Notes
Phase 1 — gem5 Candidates Analysis			
Raw candidate sequences	7,520	—	Exp. 1: 3,684 + Exp. 2: 3,836
Clean vulnerabilities (filtered)	**545**	**100%**	Mean F-score 0.980; 7.3% retention rate
Known patterns	102	18.7%	ORC, Spectre-STC, Store Conditional
ORC (Observable Register Corruption)	24	4.4%	
Spectre-STC (MUL/DIV FU contention)	57	10.5%	
Store Conditional (`sc.w`/`sc.d`)	21	3.8%	
Novel patterns	443	81.3%	11 major clusters; 3 microarch. mechanisms
Phase 2 — RTL Validation			
2-1 CTD Operand Analysis (8 cores)			
Instruction classes tested	17	—	Rocket, 6×BOOM (V3/V4), CVA6
Confirmed in cycle-accurate RTL	**15**	**88.2%**	Verilator simulation
C1 DIV/REM value-dep. timing	8/8	100%	Δ up to 65 cy (BOOM), 64 cy (CVA6)
C2 cache-address load timing	5/5	100%	Δ 22–55 cy; all 7 Chipyard cores
C3 store write-allocate timing	4/4	100%	Chipyard only; CVA6 timing-flat
2-2 Oracle Validation (multiple cores) — Details in Table 5			
Validated oracles (timing separation)	**5**	—	Up to 5 cores per oracle (see Table 5)
From *known* patterns	2	40%	FU-Contention (Spectre-STC Variant)
From *novel* patterns	3	60%	Store-Buffer, Pipeline-Drain
Phase 3 — Covert-Channel PoC on Rocket (intra-process, bare-metal, 1 000-bit run)			
Oracles achieving BER < 20%	**5**	—	`rdcycle` timing
FU-3 (3 × `mulw`→`divw`)	—	—	BER = 9.3%
FU-5 (5 × `mulw`→`divw`)	—	—	BER = 0%
SB-1 (1 × `[lr.d+sc.d]`→`flw`)	—	—	BER = 12.5%
SB-5 (5 × `[lr.d+sc.d]`→`flw`)	—	—	BER = 16.6%
PD (`lr.d+sc.d`→`fence.i` drain)	—	—	BER = 10.0%

iterative non-restoring division. `divu(1,1)` completes in 33 cycles on BOOM-Small V3 versus 93 cycles for `divu(MAX,1)`—a 60-cycle span. *RLeak* showcased a fine-grained quotient-bit-width characterization across cores and the enumeration of three instructions (`divw`, `remu`, `remuw`) not previously reported as leaking, distinct from the division-by-zero behavior of prior work [7, 31].

C2 — Cache-address load timing. Five integer and floating-point load instructions (`flw`, `fld`, `lh`, `lwu`, `ld`) expose a three-tier timing channel: L1 hit, L2 hit, or SRAM miss, with spans of 22–55 cycles across cores. This is a cache hit/miss side-channel. We enumerate four load instructions not previously reported in this role on open cores, with the address operand directly encoding cache tier.

C3 — Store write-allocate timing. Four store instructions (`sd`, `sh`, `sw`, `sb`) exhibit timing variation between warm and cold cache-line writes (up to 50 cycles on `sd`). CVA6's write-through cache absorbs the penalty. Write-allocate timing is a known effect; here all four store instructions are newly enumerated as exhibiting it on the evaluated open cores.

Figure 3 showcases timing variations for each category.

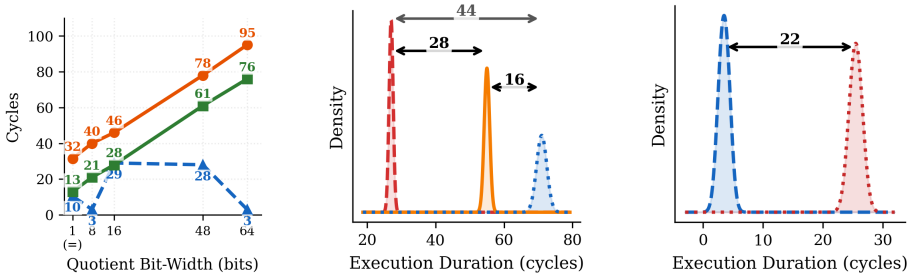
Table 4: CTD Operand-Dependent Timing Channels. **New**: independently discovered by the RL agent in gem5. ✓ = confirmed timing variation; blank = no variation ($\Delta = 0$). [a]Rocket C2 timing is non-monotonic (large-value operands can be faster); [b]confirmed on 5 of 6 BOOM variants (S/M/L/Mg-V3, S-V4); M-V4 RTL simulation crashes (LSU assertion) for div/rem; [c]CVA6 stores are timing-flat.

Instr	New	Timing trigger	Rocket	BOOM (×5)	CVA6
C1 — DIV/REM Value-Dependent Timing		*Quotient bit-width drives iterative-divider latency*			
`div`	—	1-bit quotient vs 63-bit (MAX/1)	✓[a]	✓[b]	✓
`divu`	—	1-bit quotient vs 64-bit (MAX/1)	✓[a]	✓[b]	✓
`divw`	✔	1-bit quotient vs 31-bit (MAX32/1)	✓[a]	✓[b]	✓
`divuw`	—	1-bit quotient vs 32-bit (MAX32/1)	✓[a]	✓[b]	✓
`rem`	—	0-remainder vs large dividend	✓[a]	✓[b]	✓
`remu`	✔	0-remainder vs large dividend	✓[a]	✓[b]	✓
`remw`	—	0-remainder vs large (32-bit signed)		✓[b]	✓
`remuw`	✔	0-remainder vs large (32-bit unsigned)	✓[a]	✓[b]	✓
C2 — Cache-Address Load Timing		*Address operand selects L1-cached vs L2/SRAM region*			
`flw`	✔	L1 hit (buffer) vs SRAM miss	✓	✓	✓
`fld`	✔	L1 hit vs L2/SRAM miss (64-bit FP)	✓	✓	✓
`lh`	✔	L1 hit vs L2/SRAM miss (16-bit int)	✓	✓	✓
`lwu`	✔	L1 hit vs SRAM miss (32-bit unsigned)	✓	✓	✓
`ld`	—	L1 hit vs L2/SRAM miss (64-bit int)	✓	✓	✓
C3 — Store Write-Allocate Timing		*Cache state (not data value) determines write-allocate miss penalty*			
`sd`	✔	L1-warm write vs cold write-allocate (64-bit)	✓	✓	—[c]
`sh`	✔	L1-warm write vs cold write-allocate (16-bit)	✓	✓	—[c]
`sw`	✔	L1-warm write vs cold write-allocate (32-bit)	✓	✓	—[c]
`sb`	✔	L1-warm write vs cold write-allocate (8-bit)	✓	✓	—[c]

Timing Side-Channel Oracles Five timing oracles are validated via RTL simulation across three cores. We group them by microarchitectural mechanism.

FU-Contention (FU-3, FU-5). Rocket and BOOM share a single non-pipelined MUL/DIV functional unit. A `mulw` chain in-flight occupies the FU; a subsequent `divw` stalls until the unit is released, producing a secret-dependent timing gap. FU-5 (five data-dependent `mulw`) achieves clean distribution separation on BOOM-Small and BOOM-Medium V3/V4 (BER = 0%) and overlapping distributions on Rocket and CVA6 (BER = 9.3%). This shares a microarchitectural root cause with Spectre-STC [11]—fixed-priority arbitration on a shared FU—but is exploitable in *committed, non-speculative execution* from unprivileged user space, without gadget construction.

Store-Buffer (SB-1, SB-5). `lr.d+sc.d` sequences leave dirty cache-line state in the store buffer; a subsequent `flw` must wait for buffer drain, adding ≈4 cycles. SB-5 amplifies the signal with five AMO pairs and three chained `flw` loads, achieving BER of 12.5%–16.6% on Rocket and confirmed on up to five cores. Unlike C1–C3, this is an *interaction* channel—it arises only from the composition of AMO and load instructions, not from any single opcode—and is, to our knowledge, not previously reported on open-source RISC-V cores (CWE-1303).



(a) C1: Value-Dependent `div` Latency for: ▲ Rocket, ● BOOM-S V3, ■ CVA6

(b) C2: Cache-Address Load Timing of `flw` for BOOM-Mega V3

(c) C3: Store Write-Allocate of `sb` Timing for Rocket

Fig. 3: Three operand-dependent timing hypotheses validated on RTL. **(a)** Execution latency of `div` vs. quotient bit-width for three cores: ▲ Rocket, ● BOOM-S V3, ■ CVA6 **(b)** Density curves for `flw` under three cache states: *red dashed* = D0 (L1 hit), *orange solid* = D1 (L2 hit), *blue dotted* = D2 (SRAM miss); annotated arrows indicate the cycle-count gap between consecutive tiers. **(c)** Density curves for `sb` under two cache states: *blue dashed* = L1-warm (pre-cached), *red dotted* = cold (write-allocate miss); the arrow indicates the resulting cycle gap.

Pipeline-Drain (PD). `fence.i` serializes the pipeline; its drain cost scales with in-flight micro-ops. A baseline `fence.i` drains in ≈2 cycles; preceded by `lr.d+sc.d`, the cost reaches 26 cycles. Oracle PD achieves BER = 10% on Rocket and timing separation on three BOOM variants. This too is an interaction channel, arising from AMO–`fence.i` interaction semantics (CWE-1342), and exemplifies the compositional patterns that per-instruction characterization or microbenchmarking cannot surface.

4.3 Covert-Channel PoC on Rocket

All five oracles are evaluated as covert channels in a controlled bare-metal RTL simulation on Rocket (5-stage in-order, Verilator, M-mode, interrupts disabled). Sender and receiver are co-located in a single binary; timing is measured with back-to-back `rdcycle` reads. Each binary begins with a warmup pass; for short-gap oracles (FU, SB), per-bit minimum over n measurements suppresses I-cache spikes; a three-round majority vote is applied before decoding. BER is evaluated over 1,000 pseudorandom bits.

All five oracles achieve BER < 20% (Table 3). FU-5 achieves BER = 0% after majority voting. SB errors are deterministic and positionally stable across rounds—correctable with simple forward error correction. Following the terminology of our threat model, this PoC establishes *exploitability in principle*: it confirms that the discovered timing differences carry a recoverable signal through a working covert channel under controlled co-location (single binary, interrupts

Table 5: Detailed validated timing oracles in RTL. The *Timing Separation* column shows which cores exhibit a measurable cycle-count difference between the two code paths: ✓ = clean distribution separation (no overlap); △ = overlapping distributions with distinct mean.

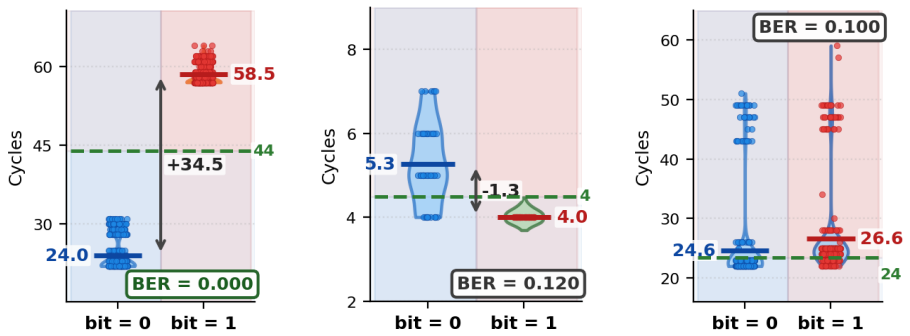
Oracle	Mechanism	CWE	Bit 0	Bit 1	Timing Separation
FU-Contention — shared MUL/DIV functional unit					**Variant: Spectre-STC**
FU-3	3×`mulw` chain occupies the shared integer MUL/DIV FU; subsequent `divw` is delayed until FU is free.	1303	`divw` (FU idle)	`3×mulw` + `divw` (FU busy)	R$^{\triangle}$
FU-5	5× data-dependent `mulw` chain saturates the FU; strengthened version of FU-3, confirms mechanism on OoO cores.		`divw` (FU idle)	`5×mulw`$_{\text{dep}}$ + `divw` (FU stalled)	R$^{\triangle}$, BS3$^{\checkmark}$, BM3$^{\checkmark}$, BS4$^{\checkmark}$, CVA6$^{\triangle}$
Store-Buffer — AMO store-buffer pressure and cache-line state					**Novel**
SB-1	1×`lr.d+sc.d` leaves dirty cache-line state in the store buffer; subsequent `flw` latency reveals buffer pressure.	1303	`flw` (buffer clean)	`lr.d+sc.d` + `flw` (buffer dirty)	R$^{\triangle}$, BS3$^{\triangle}$, BM3$^{\triangle}$, BS4$^{\triangle}$, BM4$^{\triangle}$
SB-5	5×`[lr.d+sc.d]` chain accumulates store-buffer pressure; 3×`flw` chain amplifies the timing shift.		`3×flw` (buffer clean)	`5×[lr.d+sc.d]` + `3×flw` (buffer dirty)	R$^{\triangle}$, BS3$^{\triangle}$, BM3$^{\triangle}$, BS4$^{\triangle}$
Pipeline-Drain — `fence.i` serialization reveals persistent pipeline state					**Novel**
PD	`lr.d+sc.d` loads the pipeline with in-flight micro-ops; `fence.i` serializes all stages and its drain cost is proportional to the pipeline fullness left by the AMO.	1342	`fence.i` (baseline drain)	`lr.d+sc.d` + `fence.i` (dirty drain)	R$^{\triangle}$, BS3$^{\triangle}$, BM3$^{\triangle}$, BS4$^{\triangle}$

disabled). It is deliberately not an end-to-end attack—there is no separate victim, cross-privilege boundary, or synchronization under realistic noise—which we leave to future work (§5). Figure 4 shows representative cycle-count distributions for each oracle class; Figure 5 illustrates a 16-bit SB-5 transmission trace.

4.4 Relevance to Known Attacks and Exploitability

These mechanisms map onto well-established attack primitives. *RLeak* demonstrates that an oracle-based detection on instruction-level timing differences *generalizes* known families automatically—recovering them without manual heuristics and enumerating which instructions and cores exhibit them. This per-instruction map complements developer-side mitigations such as constant-time programming and the Zkt extension, indicating precisely which opcodes cannot be used in sensitive code without mitigation.

The operand-dependent channels (C1–C3) instantiate primitives in committed, non-speculative execution. Value-dependent division (C1) mirrors the variable-time modular reduction exploited in RSA [17] and ECC timing attacks; the 60–64-cycle span on BOOM/CVA6 distinguishes single-bit secrets without amplification. Cache-address load timing (C2) is a direct cache channel in the FLUSH+RELOAD [44]/PRIME+PROBE [21] tradition, its L1/L2/SRAM tiers exposing residency of attacker-chosen addresses (22–55-cycle separation, above bare-metal noise floors). Store write-allocate timing (C3) shares DRAMA's root cause [30]: cold-line penalties reveal cache-line residency independent of load latency.



(a) FU-5 · BOOM-Medium V3 (b) SB-5 · Rocket (c) PD · CVA6

Fig. 4: Cycle-count distributions for three oracle classes. Blue = bit 0, red = bit 1; thick bars = mean; green dashed = optimal decision threshold. This illustrates that the timing oracles showcases distinct timing distribution for bit encoding enabling signal extraction for an exploitation, i.e. through a covert channel.

FU-Contention (FU-3, FU-5) is a RISC-V Spectre-STC variant that can substitute for x86 port-contention channels on Rocket and BOOM; Store-Buffer (SB-1, SB-5) and Pipeline-Drain (PD) are novel oracles extending the attack surface on in-order cores. The 0% BER of FU-5 and the large PD span confirm bit-perfect transmission under noise-free conditions—an exploitability prerequisite for a full exploit chain. These map onto commodity attacks: port contention underlies PortSmash [3] and SMoTherSpectre [6], store-buffer pressure ZombieLoad [33], and fence serialization transient-execution attacks [8].

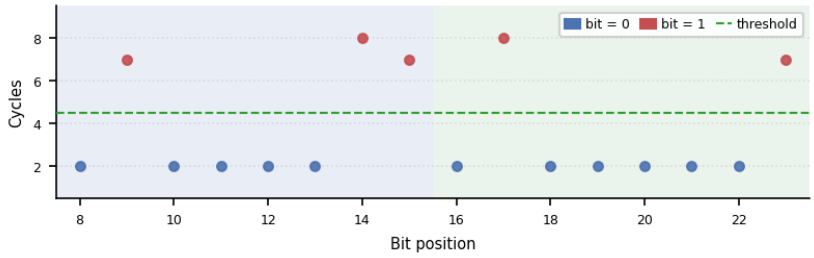


Fig. 5: Store-Buffer covert channel (SB-5) on Rocket — receiver timing for 16 bits. Each dot is one `flw` latency measurement: 2cy when the store buffer is clean (bit0), 7–8,cy when pressurised by a preceding `5,×,[lr.d+sc.d]` chain (bit1).

5 Discussion, Limitations and Future Work

5.1 Methodology Scope and Limitations

We chose gem5 to isolate RL's exploration from hardware variability, measurement noise, and non-determinism, gaining deterministic cycle-level timing and fast iteration for RL convergence. gem5 abstracts away silicon details—hence the six compressed CVA6 instructions we could not re-identify—so we confirm all findings in RTL, validating our *methodological claim*: RL-discovered oracles correspond to real microarchitectural behaviors, not simulator artifacts. Two boundaries follow. First, discovery runs under gem5's deterministic timing; we do not claim robustness on production silicon, where OS scheduling, frequency scaling, cache noise, thermal effects, and interrupts perturb measurements—real-board validation is future work. Second, the PoC of Section 4.3 is a controlled exploitability demonstration (co-located sender/receiver, interrupts disabled), not an end-to-end cross-privilege attack, which we likewise defer.

Table 6: RISC-V timing side-channel vulnerability comparison. **Detection**: framework independently discovered the pattern. Detection from other works on: BOOM (✔^{B}), CVA6 (✔^{C}) and Rocket (✔^{R}). **RTL Validation**: cycle-accurate RTL confirmation. †Covert-channel PoC (BER $< 20\%$; Rocket via `rdcycle`).

Vulnerability	**Detection**			**RTL Validation**		
	SIGFuzz [31]	**WhisperFuzz** [7]	***RLeak***	**BOOM**	**CVA6**	**Rocket**
Known						
ORC	✔^{B}		✔	✓		
Spectre-STC	✔^{B}		✔	✓		✓
`LD`	✔^{B}		✔	✓	✓	✓
`SC.D`/`SC.W`	✔^{B,R}	✔^{B,R}	✔	✓		✓
`DIV`/`DIVU`	✔^{B}	✔^{B}	✔	✓	✓	✓
`DIVUW`		✔^{C}	✔	✓	✓	✓
`REM`		✔^{B}	✔	✓	✓	✓
`REMW`		✔^{C}	✔	✓	✓	
6 C Instructions		✔^{C}				
New — 11 operand-dependent instructions + 3 timing mechanisms						
DIV/REM Value-Dep. (3 instrs)			✔	✓	✓	✓
Cache-Address Load (4 instrs)			✔	✓	✓	✓
Store Write-Allocate (4 instrs)			✔	✓		✓
FU-Contention (STC variant)			✔	✓	✓	✓†
Store-Buffer — AMO			✔	✓		✓†
Pipeline Drain			✔	✓		✓†

5.2 Future Work

Quantitative comparison. The training dynamics of Figure 2 show the agent optimizes its reward, but this is optimization under our objective, not

demonstrated superiority over random or coverage-guided fuzzing. A fair comparison is hard because no single metric isolates strategy from substrate: time-to-bug favors raw fuzzing (lower per-execution cost, but conflated with gem5 overhead), steps-to-bug favors RL, and a matched wall-clock test demands identical environment, oracle, and compute—distinct work in itself. Moreover, the closest RL hardware fuzzers [13, 42, 43] optimize RTL coverage against a golden model, not timing leakage, and run white/gray-box, so their metrics do not transfer to a timing-oracle objective. We therefore compare by scope and findings (Table 6) and leave a sound matched baseline as priority future work.

Tighter training loop. Promoting candidates to RTL mid-training and feeding RTL timing into the reward would ground the policy in microarchitectural reality, narrowing the simulation-reality gap.

Richer representation. The agent observes the partial sequence and selects the *instruction class* of the next instruction—reasoning over ordering and context—while delegating opcode and operands to the fuzzer, trading fine-grained control for a compact action space. Opcode-level, microarchitectural-event-driven, or program-sketch actions would each shift this tradeoff toward finer operand- and opcode-level reasoning.

Scaling and generalization. Scaling beyond our 10-instruction sequences [7, 31] to realistic control flow [35], via Hierarchical RL, is the natural next step. The architecture-agnostic reward and text-embedding observation are designed for cross-ISA transfer; evaluating transfer without retraining, and extending the oracle to other classes (transient execution, memory aliasing, cross-privilege channels), would broaden scope without changing the framework.

6 Conclusion

We introduced *RLeak*, a hybrid RL-guided fuzzing methodology that separates concerns: the agent observes the partial sequence and selects high-level instruction-class patterns correlated with timing anomalies, while the fuzzer instantiates concrete sequences within them. This escapes the generation-strategy bias of template-driven fuzzers and scales over the full RV64GC space without RTL access or hardware counters. Trained black-box in gem5, the agent rediscovered five known vulnerability classes (ORC, Spectre-STC, Store Conditional, DIV/DIVU, `LD`) and surfaced operand-dependent timing across 19 RV64GC instructions, 11 not previously reported on open RISC-V cores—spanning findings prior frameworks reached only through separate generation strategies. RTL validation on Rocket, BOOM, and CVA6 confirmed two novel oracle classes (Store-Buffer pressure, Pipeline-Drain) and a RISC-V variant of Spectre-STC FU-Contention, with a bare-metal covert channel on Rocket reaching bit error rates of 0%–16.6%. These confirm our core claim: oracles found in simulation correspond to real RTL behaviors, not artifacts. RLeak establishes hybrid RL-guided fuzzing as a scalable first line of defense for proactive hardware security assessment, with future work extending it toward a fully automated testbed across modern SoC architectures.

Acknowledgment

This work has been supported by the French National Research Agency in the frame of PEPR-ARSENE (ANR-22-PECY-0004). We thank Jean-Roch Coulon, Thales Group and the Eclipse/OpenHW Foundation for CVA6, and the UC Berkeley Architecture Research (BAR) group for Rocket, for their engagement during the coordinated-disclosure process and for discussions that helped refine our findings.

Ethics and Disclosures

This work is intended strictly for defensive security research, with the goal of improving the resilience of processor designs against microarchitectural timing side-channel attacks. The framework is designed to assist hardware designers and security researchers in identifying vulnerabilities during design and verification, before deployment.

All vulnerabilities discovered through RLeak were identified in simulation (gem5, RTL) and responsibly disclosed to the upstream maintainers of CVA6, BOOM, and Rocket prior to publication. The CVA6 and Rocket maintainers acknowledged the reported behaviors. For Rocket, the BAR group acknowledged that the underlying mechanisms are consistent with the design intent of the core; given that Rocket was not designed for multi-tenant deployment, no upstream hardening is planned. For the CVA6, the findings have been disclosed to a weekly meeting of the OpenHW Foundation.

References

1. RLeak: Source code. `https://github.com/nthsmn/RLeak` (2026)
2. RLeak: Validation artifacts. `https://github.com/nthsmn/RLeak-Artifacts` (2026)
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., Pereida García, C., Tuveri, N.: Port contention for fun and profit. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 870–887 (2019). https://doi.org/10.1109/SP.2019.00066
4. Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., et al.: Chipyard: Integrated design, simulation, and implementation framework for custom socs. Ieee Micro **40**(4), 10–21 (2020)
5. Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al.: The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 **4**, 6–2 (2016)
6. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: Smotherspectre: exploiting speculative execution through port contention. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 785–800 (2019)

7. Borkar, P., Chen, C., Rostami, M., Singh, N., Kande, R., Sadeghi, A.R., Rebeiro, C., Rajendran, J.: {WhisperFuzz}:{White-Box} fuzzing for detecting and locating timing vulnerabilities in processors. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 5377–5394 (2024)
8. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 249–266. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/canella
9. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School on Software Engineering, pp. 1–30. Springer (2011)
10. Cui, J., Yang, X., Luo, M., Lee, G., Stone, P., Lee, H.H.S., Lee, B., Suh, G.E., Xiong, W., Tian, Y.: MACTA: A multi-agent reinforcement learning approach for cache timing attacks and detection. In: The Eleventh International Conference on Learning Representations (2023), https://openreview.net/forum?id=CDlHZ78-Xzi
11. Fadiheh, M.R., Müller, J., Brinkmann, R., Mitra, S., Stoffel, D., Kunz, W.: A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In: Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference. DAC '20, IEEE Press (2020)
12. Fadiheh, M.R., Wezel, A., Müller, J., Bormann, J., Ray, S., Fung, J.M., Mitra, S., Stoffel, D., Kunz, W.: An exhaustive approach to detecting transient execution side channels in rtl designs of processors. IEEE Transactions on Computers **72**(1), 222–235 (2022)
13. Götz, R., Sendner, C., Ruck, N., Rostami, M., Dmitrienko, A., Sadeghi, A.R.: Rlfuzz: Accelerating hardware fuzzing with deep reinforcement learning. In: 2025 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 358–369 (2025). https://doi.org/10.1109/HOST64725.2025.11050051
14. Hasan, O., Tahar, S.: Formal verification methods. In: Encyclopedia of Information Science and Technology, Third Edition, pp. 7162–7170. IGI Global Scientific Publishing (2015)
15. Hertogh, M., Quakkelaar, D., Raymakers, T., Sarma, M.H., Muench, M., Bos, H., van der Kouwe, E.: Rain: Transiently leaking data from public clouds using old vulnerabilities. In: Symposium on Security and Privacy. IEEE (2026)
16. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. Communications of the ACM **63**(7), 93–101 (2020)
17. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96. pp. 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
18. Liang, E., Wu, Z., Luo, M., Mika, S., Gonzalez, J.E., Stoica, I.: Rllib flow: Distributed reinforcement learning is a dataflow problem. Advances in Neural Information Processing Systems **34**, 5506–5517 (2021)
19. Lipp, M., Gruss, D., Schwarz, M.: Armageddon: Last-level cache attacks on mobile devices. In: USENIX Security Symposium (2016)
20. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., et al.: Meltdown: Reading kernel memory from user space. Communications of the ACM **63**(6), 46–56 (2020)

21. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE symposium on security and privacy. pp. 605–622. IEEE (2015)
22. Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Beckmann, B., Bharadwaj, S., et al.: The gem5 simulator: Version 20.0+. arXiv preprint arXiv:2007.03152 (2020)
23. Luo, M., Xiong, W., Lee, G., Li, Y., Yang, X., Zhang, A., Tian, Y., Lee, H.H.S., Suh, G.E.: Autocat: Reinforcement learning for automated exploration of cache-timing attacks. In: 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 317–332. IEEE (2023)
24. MITRE Corporation: Common Vulnerabilities and Exposures (CVE). `https://cve.mitre.org`
25. MITRE Corporation: Common Weakness Enumeration (CWE). `https://cwe.mitre.org`
26. Mushtaq, M., Mukhtar, M.A., Lapotre, V., Bhatti, M.K., Gogniat, G.: Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for rsa. Information Systems **92**, 101524 (2020)
27. Oleksenko, O., Fetzer, C., Köpf, B., Silberstein, M.: Revizor: Testing black-box cpus against speculation contracts. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 226–239 (2022)
28. Oleksenko, O., Guarnieri, M., Köpf, B., Silberstein, M.: Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1737–1752. IEEE (2023)
29. Patnaik, S., Gohil, V., Guo, H., Rajendran, J.J.: Reinforcement learning for hardware security: Opportunities, developments, and challenges. In: 2022 19th International SoC Design Conference (ISOCC). pp. 217–218. IEEE (2022)
30. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: {DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks. In: 25th USENIX security symposium (USENIX security 16). pp. 565–581 (2016)
31. Rajapaksha, C., Delshadtehrani, L., Egele, M., Joshi, A.: Sigfuzz: A framework for discovering microarchitectural timing side channels. In: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2023)
32. RISC-V International: The "G" Standard Extension for Compressed Instructions. In: The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA (2019), chapter 24 - RV32/64G Instruction Set Listings
33. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: Zombieload: Cross-privilege-boundary data sampling. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 753–768 (2019)
34. Snyder, W., et al.: Verilator: Open-source SystemVerilog simulator and lint system. GitHub repository, `https://github.com/verilator/verilator`, accessed: Feb. 24, 2026
35. Solt, F., Ceesay-Seitz, K., Razavi, K.: Cascade:{CPU} fuzzing via intricate program generation. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 5341–5358 (2024)
36. Thomas, F., Arribas, E.G., Hetterich, L., Weber, D., Gerlach, L., Zhang, R., Schwarz, M.: Riscover: Automatic discovery of user-exploitable architectural security vulnerabilities in closed-source risc-v cpus. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 3326–3340 (2025)

37. Thomas, F., Torres, M., Moghimi, D., Schwarz, M.: Exfilstate: Automated discovery of timer-free cache side channels on arm cpus. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2564–2578 (2025)
38. Tol, M.C., Derya, K., Sunar, B.: {\mu} rl: Discovering transient execution vulnerabilities using reinforcement learning. arXiv preprint arXiv:2502.14307 (2025)
39. Towers, M., Kwiatkowski, A., Terry, J., Balis, J.U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al.: Gymnasium: A standard interface for reinforcement learning environments. arXiv preprint arXiv:2407.17032 (2024)
40. Trippel, C., Lustig, D., Martonosi, M.: Checkmate: Automated synthesis of hardware exploits and security litmus tests. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 947–960. IEEE (2018)
41. Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., Rossow, C.: Osiris: Automated discovery of microarchitectural side channels. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1415–1432 (2021)
42. Wu, L., Rostami, M., Li, H., Rajendran, J., Sadeghi, A.R.: GenHuzz: An efficient generative hardware fuzzer. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 1787–1805 (2025)
43. Wu, L., Rostami, M., Li, H., Sadeghi, A.R.: Hfl: Hardware fuzzing loop with reinforcement learning. In: 2025 Design, Automation & Test in Europe Conference (DATE). pp. 1–7 (2025). https://doi.org/10.23919/DATE64628.2025.10993080
44. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: USENIX Security Symposium (2014)
45. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (Nov 2019). https://doi.org/10.1109/TVLSI.2019.2926114
46. Zhang, S., Wang, H., Qiu, P., Lyu, Y., Wang, H., Wang, D.: Scafinder: Formal verification of cache fine-grained features for side channel detection. IEEE Transactions on Information Forensics and Security **19**, 8079–8093 (2024)
47. Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., et al.: Qwen3 embedding: Advancing text embedding and reranking through foundation models. arXiv preprint arXiv:2506.05176 (2025)
48. Zhao, J., Korpan, B., Gonzalez, A., Asanovic, K.: Sonicboom: The 3rd generation berkeley out-of-order machine. In: Fourth Workshop on Computer Architecture Research with RISC-V. vol. 5, pp. 1–7. International Symposium on Computer Architecture Valencia (2020)