Architectural Leakage Analysis of Masked Cryptographic Software on RISC-V Cores

Siddhartha Chowdhury, Nimish Mishra, Sarani Bhattacharya and Debdeep Mukhopadhyay

Indian Institute of Technology Kharagpur, India
{siddhartha.chowdhury92,neelam.nimish,bhattacharya.sarani.iitkgp,debdeep.mukhopadhyay}@gmail.com

Abstract. Software masking—particularly through threshold implementations—has long been regarded as a foundational defense mechanism against side-channel attacks. These schemes enforce the principles of non-completeness and uniformity, offering provable first-order resistance even under realistic leakage assumptions. However, such assurances were primarily developed under the simplified assumption of scalar or in-order execution, where instruction flow and data dependencies are well-behaved and predictable.

Contemporary processors, by contrast, employ deeply optimized out-of-order (OoO) pipelines that rely on dynamic scheduling, register renaming, operand forwarding, and speculative execution. These aggressive microarchitectural behaviors, while improving performance, also create intricate dataflow interactions that can inadvertently violate the independence of masked shares. As a result, secret shares that are theoretically isolated may recombine transiently through register reuse, speculative reordering, or transition-based leakage in the physical register file. This raises a critical question: to what extent do masking countermeasures remain secure when executed on modern OoO architectures?

To explore this question, we develop a systematic methodology for architectural leakage analysis based on fine-grained execution traces from an OoO RISC-V processor. The analysis correlates microarchitectural events with instruction-level behavior, reconstructs the evolution of physical register assignments, and identifies instances where fundamental masking principles such as non-completeness or uniformity are violated. Through this structured approach, we establish a taxonomy of microarchitectural leakage classes and formalize two dominant categories: (1) *Rename-Induced Transition Leakage (RIL)*, arising from register reuse and transitional dependencies between masked shares; and (2) *IEW-Induced Dispatch Leakage*, originating from speculative instruction issue and execution reordering.

The methodology is demonstrated across three representative studies: (i) a masked Toffoli gate, where violations of non-completeness are revealed; (ii) a masked PRESENT cipher, where 571 leakage events are identified due to register-induced transitions; and (iii) a Probe Isolating Non-Interference (PINI) experiment, which exposes how speculative and dynamic scheduling can compromise isolation guarantees, thereby weakening composability of masked implementations that are provably secure under idealized models.

These results underline a fundamental insight: the security of masking countermeasures cannot be evaluated in abstraction from the hardware on which they execute. Ensuring true resistance to side-channel attacks demands a holistic view—one that jointly considers both the algorithmic soundness of masking constructions and the microarchitectural realities of modern processors.

Keywords: Out-of-Order · Masking · RISC-V

1 Introduction

Since the introduction of side-channel analysis (SCA) [Koc96,KJJ99], it has become evident that cryptographic algorithms—although mathematically sound and provably secure in the black-box model—may inadvertently leak sensitive information when implemented on physical devices. Masking, which divides a secret into randomized shares and processes each share independently, remains one of the most established countermeasures against SCA [CJRR99,ISW03]. Threshold Implementations (TIs) and related masking refinements further strengthen this defense by addressing physical effects such as glitches and transitions. When correctly applied, these methods yield designs that are provably first-order secure under conventional micro-architectural leakage models.

Several studies have demonstrated that such theoretically secure masked implementations can leak due to such micro-architectural effects. Practical analyses have revealed leakages originating from memory transitions, register overwrites, and pipeline forwarding phenomena [RP10, LCGD18, MM22]. The PROLEAD framework and its software adaptation, PROLEAD_SW [ZMM23], represent significant progress in the direction of automated, probing-based leakage evaluation. These tools introduce a CPU-independent leakage model capable of simulating realistic physical effects, including glitches and transitions [MM22]. These prior works propose mechanisms to prevent such leakage in these masked implementations.

However, the guarantees of these prior works like [MM22] typically rely on the assumption of scalar, in-order execution, where instructions are processed sequentially and intermediate states remain isolated as intended by the software. Modern high-performance processors deviate from this assumption by employing *out-of-order* (OoO) execution to maximize instruction throughput. OoO cores dynamically rename architectural registers, dispatch instructions to functional units as soon as their operands are ready, and retire them later to maintain architectural correctness. The presence of complex micro-architectural components—such as register alias tables (RATs), reorder buffers (ROBs), issue queues, forwarding networks, and speculative execution logic—introduces transient states that are largely transparent to software developers. The question we ask in this work is whether such OoO execution properties compromise the logical independence of shares in masked implementations that are otherwise proven secure on in-order processors. Concretely, whether multiple shares of a secret (although securely isolated in software) still coexist within the same physical register, functional unit, or forwarding path on an OoO processor, leading to transition-based leakage, speculative recombination, or forwarding-path collisions that violate the non-completeness and uniformity conditions essential to TI security. Because existing methodologies for automated leakage detection like [MM22] focus primarily on scalar or simple pipelined processors, they cannot fully characterize the leakages that emerge under OoO specific hardware optimizations.

In this work, we bridge this gap by introducing `OoOLyzer`: a comprehensive analysis framework that detects and classifies micro-architectural leakages arising when masked software executes on out-of-order processors. `OoOLyzer` operates on fine-grained execution logs—such as those obtained from the gem5 simulator with O3PipeView instrumentation—and a configuration file that defines relationships between architectural registers and their corresponding masked shares. From these traces, `OoOLyzer` reconstructs the mapping between architectural and physical registers, tracks their lifetimes, and identifies events where distinct shares are assigned to the same physical resource. By correlating static assembly information with dynamic rename and forwarding behaviors, the framework isolates hazardous interactions, associates them with program counter (PC) locations, and ranks their severity. The resulting analysis provides developers with an actionable diagnostic view of where and how OoO execution undermines the security guarantees of masking.

Our decision to base `OoOLyzer` on simulation rather than post-silicon is intentional:

the targeted leakage sources are not architecturally observable by software in silicon. Simulation, however, enables both detection and precise *attribution* of leakage to OoO execution. Any leakage identified through this process—even if weak or noisy—remains exploitable, as share mixing during masked block cipher execution on an OoO processor may reveal key-dependent intermediates.

Why these shares are vulnerable and how mixing becomes exploitable. While masking provides theoretical first-order security, OoO scheduling compromises isolation through two effects:

- *Physical-register reuse:* rapid renaming can reassign the same physical register before prior state is cleared, causing transitions from $v_A = S_A \oplus K_A$ to $v_B = S_B \oplus K_B$ that leak $\mathrm{HD}(v_A, v_B)$—a function of $S \oplus K$ exploitable via first-order TVLA/CPA.

- *Transient co-residency:* concurrent issue of domain-specific AND/XOR operations enables temporary hardware-level mixing of shares, producing measurable first-order correlations.

In the analyzed trace, many candidates cluster in the S-box layer, where transitions of the form $(S_A \oplus K_A) \rightarrow (S_B \oplus K_B)$ reveal linear combinations of state and key bits under a Hamming-distance model, making each reuse event a potential source of exploitable leakage.

Summarily, our main contributions are as follows:

- We formalize *leakage signals* that arise when masked software executes on an OoO core and develop a taxonomy of OoO-induced leakage mechanisms—including rename-induced transitions and IEW-induced dispatch overlaps—relating them to existing probing and robust-probing models. The taxonomy defines predicates for Rename-Induced Transition Leakage (RIL) and IEW-Induced Dispatch Leakage, and formalizes their violation conditions.

- We propose a trace-driven methodology to detect and attribute such leakages by reconstructing rename mappings, register reuse, and forwarding conflicts from detailed OoO execution logs. The six-stage analysis pipeline—covering event extraction, correlation, and predicate evaluation—is integrated into the `OoOLyzer` framework.

- We implement and evaluate `OoOLyzer` on realistic masked implementations executed on an OoO RISC-V core, showing that designs provably secure under TI-style assumptions can still leak under OoO execution. Three case studies—masked Toffoli, masked PRESENT (571 leakage points), and PINI—illustrate distinct forms of RIL and IEW-induced leakages.

- We discuss practical mitigations, including compiler-assisted register allocation, selective serialization, and hardware-aware scheduling, and advocate for joint hardware–software verification of masking security.

The remainder of this paper is organized as follows. Section 2 provides background on masking, threshold implementations, and the key aspects of out-of-order microarchitecture that give rise to leakage phenomena. Section 3 defines the threat model and introduces a taxonomy of out-of-order–induced leakages. Section 3.3 formalizes the `OoOLyzer` framework, detailing its trace-driven analysis model, leakage predicates, and detection pipeline. Section 4 describes the experimental setup and presents three case studies—on masked Toffoli, masked PRESENT, and PINI gadgets—demonstrating the framework's applicability. Section 5 concludes the paper.

Notation and scope. We focus primarily on first-order Boolean masking and threshold-implementation style software, though the methodology generalizes to higher-order and alternative sharing schemes. All experiments are performed on a detailed OoO RISC-V model instrumented to emit rename, dispatch, execute, write-back, and retire events. The algorithms underlying `OoOLyzer` assume the availability of such events but can be extended to other trace formats. We adopt the robust probing framework introduced in prior work [FGMDP+18] to reason about glitches and transitions, extending it to account for OoO-specific phenomena recently identified in the literature [GPM21, GD23].

2 Background and Motivation

This section provides concise background material and motivation that the reader needs to follow the remainder of the paper. We recall the essentials of Boolean masking and threshold implementations (sufficiently briefly to keep the narrative compact), summarize the relevant out-of-order (OoO) micro-architectural mechanisms that give rise to the phenomena studied in this work, enumerate the principal leakage pathways that the analysis targets, and conclude with an illustrative example that ties the background to the empirical observations reported later.

2.1 Masking and non-completeness

Boolean masking is a countermeasure that splits a sensitive value into multiple randomized *shares* and operates on these shares independently so that any single observation yields no information about the secret. In the simplest, first-order case a value v is represented as $v = s_0 \oplus s_1$ with s_0 uniformly random; operations are transformed so that intermediate values do not combine more than one share at the same hardware node. Threshold implementations (TIs) refine this idea by enforcing two core properties: *non-completeness*, which forbids a single implementation node from depending on all shares of a secret, and *uniformity*, which forces intermediate distributions to remain data-independent when appropriate random masks are used. These properties underpin many compositional security proofs in the probing model. For our purposes we assume familiarity with the standard masking literature and the TI methodology [CJRR99, ISW03, GD23]; we only emphasize that TI proofs rely critically on assumptions about which hardware nodes may be influenced by which share at each program point. The remainder of this paper studies how OoO execution can invalidate those assumptions at runtime.

2.2 Out-of-order execution basics

This subsection briefly summarizes the micro-architectural components whose interactions create the transient hardware state exploited by OoO specific leakage like the rename unit. The goal is descriptive: we introduce terminology and clarify which internal structures govern allocation, scheduling, and short-lived data movement. Analysis and detection logic remain in Section 3.

Modern high-performance cores decouple architectural program order from physical execution to exploit instruction-level parallelism. This is enabled by several key structures. The **Register Alias Table (RAT)** maps logical to physical registers: when an instruction is decoded, the RAT and free-list allocate a new physical register (PReg) for its destination. The mapping persists until the instruction commits, after which the previous PReg is released. This mechanism eliminates false write dependencies and supports multiple in-flight updates to the same architectural register.

The **Issue Queue (IQ)** buffers decoded micro-operations until their operands and a functional unit (FU) are available, allowing issue based on readiness rather than program

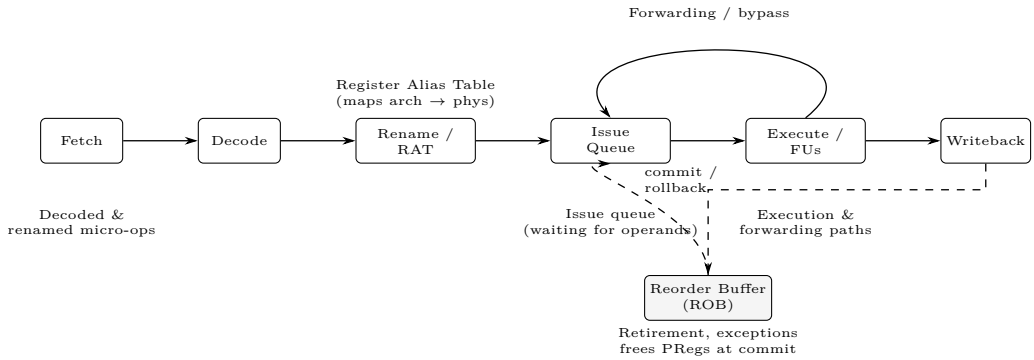

Figure 1: Compact schematic of a canonical OoO pipeline highlighting the Register Alias Table (RAT), Issue Queue, forwarding paths, and the Reorder Buffer (ROB). Dashed arrows indicate commit/rollback interactions mediated by the ROB.

order. Executed results are held in the **Reorder Buffer (ROB)**, which tracks in-flight instructions, preserves precise exceptions, and ensures in-order retirement. Upon commit, the architectural state is updated and the old PRegs are freed. Results can also be forwarded directly to dependent instructions via the bypass network, allowing immediate data reuse without waiting for writeback.

Taken together, these elements create multiple layers of transient state. The RAT and free-list determine where values are stored physically; the issue/scheduler and FUs determine when values are computed and thus when writes or forwards occur; and the ROB determines when speculative state becomes permanent. Critically for leakage analysis, allocation (RAT/free-list) and scheduling (issue/FU) operate on independent timing axes: a PReg freed at commit can be reassigned quickly depending on free-list policy, and instructions that are adjacent in program order may nonetheless execute with small or reversed temporal separation due to operand readiness or FU latencies. It is this decoupling of *where* values are stored and *when* they are produced that underlies rename-induced transition leakage.

2.3 Example leakage scenario

To illustrate the connection between the micro-architecture and masking assumptions, consider two architectural registers a_0 and a_1 that hold the two shares of a masked value. In an in-order execution model, the programmer or compiler can ensure these live in separate architectural registers and are written at times that prevent any single hardware node from being influenced by both shares. In an OoO core, however, the rename unit may map a_0 to a physical register P_i, and, shortly thereafter, map a_1 to the same P_i while the writes occur in close succession. **The resulting transition $V_{\text{old}} \rightarrow V_{\text{new}}$ on P_i has Hamming-distance components that depend on both a_0 and a_1 and thus violates non-completeness.** Figure 2 provides a conceptual timeline of such an event based on the RIL observations we report in Section 4.6.

The background presented above motivates the need for a trace-driven framework that: (i) reconstructs rename mappings and per-cycle IEW (Issue-Execute-Writeback) occupancy[1], (ii) correlates physical-register writes with assembly-level sources, and (iii) scores and ranks candidate leakage events for further experimental validation. The next section presents the `OoOLyzer` architecture and the concrete detection algorithms we employ.

[1]IEW: Combined abstraction of Issue, Execute, and Writeback phase of an OoO execution pipeline.

3 Threat Model and Leakage Taxonomy

This section specifies the adversarial model used throughout the paper, summarizes the two classes of micro-architectural leakage we target, and provides a formal description of our trace-driven analyzer, `OoOLyzer`. The exposition emphasizes the concrete micro-architectural hazards that are central to our experiments and to the tool evaluation, and it describes the methodology used to transform simulator/instrumentation traces into precise, triageable candidate events for focused physical validation.

3.1 Adversarial model

In this work, we address the problem of attributing leakages in otherwise secure masked implementations when executed on out-of-order (OoO) processors. We consider a two-phase adversarial model: in the *offline* phase, the adversary profiles the implementation on its own OoO core to detect and attribute leakages to specific micro-architectural elements; in the *online* phase, the adversary targets the victim core at those elements but is restricted to passive side-channel trace collection.

During the offline phase, the adversary (or analyst) is passive and non-invasive, observing time-stamped micro-architectural state transitions—such as physical-register file updates, forwarding-path activity, pipeline latches, and issue-queue occupancy—while masked software executes. The program and hardware remain unmodified; instead, execution records are correlated with assembly-level information and a semantic mapping of architectural registers to masked shares.

We make the following assumptions:

- The processor preserves architectural correctness and in-order retirement; any leakage therefore arises from transient micro-architectural interactions below the architecture boundary.
- The analyst can associate trace events (rename, issue, execute, writeback, retire) with runtime PCs and register indices to *detect* leakage in the offline phase; such association is unnecessary during the online attack.

The adversary's **goal** is structural detection: to identify micro-architectural violations of masking invariants (e.g., non-completeness or PINI isolation) and produce compact, explainable candidates for physical validation via power or EM measurements.

3.2 Targeted Leakage Mechanisms : Leakage Taxonomy

Out-of-order (OoO) cores introduce dynamic hardware resources and timing interactions that can invalidate masking proofs formed under simpler in-order models. We identify two complementary leakage forms arising from such behavior:

1. **Transition-based signals:** capture bitwise changes at hardware nodes (e.g., Hamming distance between consecutive writes to the same physical register). Register renaming may map two shares a_0 and a_1 of a secret a to the same physical register, violating the non-completeness assumption and exposing transition-based leakage.
2. **Co-residency signals:** capture transient overlap of share-updating instructions within internal structures (issue queue, forwarding paths, or pipeline latches). For instance, reordering within the IEW[2] pipeline can cause unsafe co-execution of instructions that should be sequential, voiding isolation guarantees.

We focus on two representative mechanisms underlying these signals.

[2]Combined abstraction of Issue, Execute, and Writeback stages of an OoO pipeline.

Table 1: Summary of targeted OoO-induced leakage mechanisms.

Mechanism	Root cause / description	Impact on masking
Rename-Induced Transition Leakage (RIL)	Reuse of the same physical register for different shares causes value transitions dependent on both, forming observable **transition-based** leakage.	Violates *non-completeness*: one hardware node transiently holds multiple shares.
IEW-Induced Dispatch Leakage	OoO scheduling and speculation lead to transient co-residency of share-updating instructions in internal paths, producing **co-residency** leakage.	Breaks *PINI* isolation: transient recombination undermines composability guarantees.

Rename-Induced Transition Leakage (RIL). RIL occurs when the renamer reuses a physical register for two architectural destinations holding different shares. Consecutive writes to the same PReg cause transitions dependent on both shares, producing observable power or EM variations that violate threshold non-completeness.

IEW-Induced Dispatch Leakage. This leakage arises when OoO scheduling issues conflicting share-updating instructions concurrently, creating transient co-residency on internal paths not modeled in classical leakage frameworks. Such overlaps effectively introduce new hardware probe points, invalidating the isolation assumptions of the Probe-Isolating Non-Interference (PINI) model.

Table 1 summarizes these mechanisms and their effects on classical masking properties.

3.3 Formalization of the trace-driven analyzer (`OoOLyzer`)

We now present a compact formal model of the trace semantics, detection predicates implemented by `OoOLyzer`, and the analysis pipeline. This formalization makes the assumptions explicit and connects them to the pragmatic parsing heuristics used in practice.

Trace model. Let $\mathcal{L} = (L_1, L_2, \ldots, L_N)$ denote a time-ordered sequence of trace lines produced by a simulator or instrumentation front-end. Each line L_i may contain zero or more of the following (possibly partial) event records; each event can carry optional metadata (simulator tick, runtime PC, textual payload):

1. $\mathrm{Rename}\big(t, \mathit{arch}, \mathit{preg}, \mathit{pc}\big)$ — (architectural register *arch* renamed to physical register *preg*).
2. $\mathrm{Write}\big(t, \mathit{preg}, v, \mathit{pc}\big)$ — (write to physical register *preg* with value v).
3. $\mathrm{IEW}\big(t, \sigma, \mathrm{SN}, \mathit{pc}\big)$ — (IEW stage occupancy or serial-number reporting at stage σ).

Here t denotes a simulator tick (or, lacking an explicit tick, the monotone line index), *pc* is an optional runtime PC, *arch* is an architectural register index, *preg* is a physical-register index, v is the written value (when available), σ denotes a pipeline stage, and SN is an instruction serial number. We explicitly differentiate between *pc* and SN to capture OoO dispatcher semantics: *pc* represents the architectural order of instructions, while SN represents the dispatch order (and is different than the architectural order). The analyzer

reconstructs event streams by parsing $\mathcal{L}$ and associating available tick/PC metadata with the extracted events.

Architectural mapping and conflict relation. Let $\mathcal{M}$ be the analyst-supplied mapping that translates architectural-register indices (or small human-readable labels such as `a2`, `a5`) into *share labels* (for example, c_0, c_1, or domain identifiers). An analyst also provides a set of unordered *conflicting pairs* $\mathcal{C}$; each pair identifies two share labels that must not jointly influence a single micro-architectural node if masking invariants are to be respected. `OoOLyzer` uses $\mathcal{M}$ and $\mathcal{C}$ to decide which rename/write or SN-co-residency events are relevant. We say `OoOLyzer` detects a **violation** when any conflicting pair in $\mathcal{C}$ has a mapping to the same architectural register in $\mathcal{M}$.

RIL predicate (rename→write transitions). We now explain how `OoOLyzer` detects Rename-Induced Transition Leakage (RIL). Fix a physical-register index p. Let $\text{Renames}_p = (R_{p,1}, R_{p,2}, \dots)$ be the time-ordered sequence of rename events that assign architectural registers to p. Let $\text{Writes}_p = (W_{p,1}, W_{p,2}, \dots)$ be the ordered writes observed to p. Consider consecutive rename events $R_{p,i}$ and $R_{p,i+1}$ whose architectural sources map under $\mathcal{M}$ to share labels s and s'. Let W_{old} be the first write to p after $R_{p,i}$ but before $R_{p,i+1}$, and let W_{new} be the first write following $R_{p,i+1}$.

The RIL predicate holds when:

1. $\{s, s'\} \in \mathcal{C}$ (the two sources form a configured conflicting pair);
2. W_{old} and W_{new} both exist and are contiguous in the p-write stream (no intervening write from a third source); and
3. $\Delta t = t(W_{\text{new}}) - t(W_{\text{old}})$ is within a configured maximum gap $\tau_{\max}$ (measured in ticks or line indices).

When the predicate holds and both written values are available, `OoOLyzer` computes the transition magnitude

$$L_{\text{RIL}}(p) \;=\; \text{HD}\big(v(W_{\text{old}}), v(W_{\text{new}})\big),$$

the Hamming distance between the two values, which serves as a conservative structural indicator of potential transition-based leakage. `OoOLyzer` reports this as a **violation**.

IEW-Induced Dispatch Leakage. We now explain how `OoOLyzer` detects IEW-Induced Dispatch Leakage. Let $\mathcal{S}_t(\sigma)$ be the set of instruction serial numbers (SN) resident in pipeline stage σ at tick t. For any pair $\{a, b\} \subseteq \mathcal{S}_t(\sigma)$, `OoOLyzer` attempts to resolve the architectural-register sources that produced SNs a and b by scanning nearby rename events and mapping them via $\mathcal{M}$. If the resolved labels form a conflicting pair $\{s, s'\} \in \mathcal{C}$ and their co-residency duration exceeds a small threshold $\delta_{\min}$, the analyzer reports a potential IEW-Induced Dispatch Leakage: a sustained co-residency of conflicting share-updating instructions on an internal hardware path that is not modeled by software.

Implementation sketch (analysis pipeline). The analyzer implements the formal model as a sequence of robust parsing and correlation steps:

1. **Pre-scan:** annotate each trace line with the last-seen runtime PC (fall back to line index when explicit tick/PC fields are absent).
2. **Event extraction:** apply lightweight substring filtering followed by tolerant regular-expression extraction to recover Rename, RegFile Write, and IEW events.

3. **Collapse & deduplicate:** collapse noisy or duplicate rename messages within a small configurable window to avoid verbosity-induced duplicates.
4. **Correlation:** for each physical register p, align renames and RegFile writes to reconstruct Renames_p and Writes_p.
5. **Predicate evaluation:** evaluate the RIL and IEW-Induced Dispatch Leakage predicates and compute raw, explainable metrics (HD, Δt, overlap durations, PC/SN alignment confidence).
6. **Reporting:** produce a human-readable consolidated report and a structured machine-readable output that contains the full candidate set and contextual trace excerpts for reproducible triage and physical validation.

The `OoOLyzer` framework intentionally emits raw, explainable metrics for each candidate (Hamming distance, Δt, PC/SN proximity confidence and surrounding trace excerpts). The framework supports analyst-driven configuration options such as:

- a semantic mapping $\mathcal{M}$ (architectural-register index $\rightarrow$ share label),
- a conflict set $\mathcal{C}$ of register-pair labels to monitor,
- a duplicate-collapse window for noisy rename messages,
- an optional symbol-range filter (to focus analysis on a single function or code region), and
- the ability to skip IEW analysis where pipeline-occupancy records are not present.

Outputs are two-fold: a human-readable consolidated report with contextual trace excerpts for each candidate, and a structured machine-readable file for downstream filtering, ranking, and experimental planning.

Heuristic robustness and generalization. In practice, trace formats and event structures vary across simulators and instrumentation environments. `OoOLyzer` integrates a series of adaptive parsing and correlation strategies designed to generalize across such heterogeneity while preserving analytical precision. These include lightweight substring filtering prior to full regular-expression parsing, fallback mechanisms that use line indices when explicit timestamps are unavailable, automatic consolidation of duplicate rename messages within a configurable temporal window, and bounded contextual analysis for program-counter and serial-number resolution.

The detection predicates are constructed to operate on well-ordered rename and writeback events and can selectively focus on individual threads or execution contexts. The framework's modular design allows analysts to refine architectural-to-share mappings and expand instrumentation coverage incrementally, ensuring that detection sensitivity remains high even in partially instrumented environments. In this way, `OoOLyzer` emphasizes analytical robustness and cross-platform generalizability—transforming raw trace data from diverse microarchitectural sources into consistent, explainable indicators of potential leakage behavior.

4 Experimental Setup

This section describes the methodological framework used to evaluate the proposed `OoOLyzer` analyzer. The experiments are designed to systematically capture, reconstruct, and evaluate leakage-relevant microarchitectural behavior in a controlled simulation environment. The overall pipeline comprises three tightly coupled stages: (i) generation

of RISC-V workloads, (ii) cycle-accurate simulation and trace acquisition, and (iii) post-simulation analysis using `OoOLyzer`. Each stage was executed under consistent architectural and timing parameters to ensure reproducibility across different masked designs.

4.1 Workload design and compilation

The workloads used in this study are written at the assembly level to provide complete visibility into the instruction stream and register allocation. Each workload implements a cryptographic operation that employs first-order Boolean masking at the instruction level—examples include the masked Toffoli gate, masked PRESENT round function, and masked Keccak permutation. Working directly at the assembly level allows explicit control over the register usage of individual shares and randomness variables, ensuring that the architectural mapping between the share domains and the processor register file is unambiguous.

All workloads target the 32-bit integer and multiplication subset (RV32IM) of the RISC-V ISA. The assembly sources are compiled using a cross-compilation toolchain that produces statically linked ELF binaries. These ELF images preserve full symbol information and maintain a flat memory layout to simplify mapping between runtime program counters and static symbol addresses. The inclusion of debug symbols is essential, as it enables the analyzer to associate low-level trace events with their corresponding code regions during post-analysis.

4.2 Microarchitectural simulation and trace acquisition

To emulate realistic register renaming and out-of-order scheduling behavior, each workload is executed on a cycle-accurate RISC-V CPU model within the `gem5` simulation framework. An out-of-order (OoO) core model based on the DerivO3CPU pipeline is used, as it closely resembles the behavior of contemporary superscalar processors that employ aggressive register renaming, speculative issue, and dynamic writeback scheduling. The simulation environment operates in syscall-emulation mode to eliminate operating-system noise, ensuring that the traces exclusively reflect user-level instruction activity.

The simulation records a detailed, time-ordered sequence of internal microarchitectural events encompassing register renaming, register file writes, pipeline occupancy updates, and instruction issue or retirement information. Each event is timestamped with a simulation tick and tagged with the corresponding runtime program counter, enabling a precise reconstruction of temporal and spatial dependencies between operations. By enabling specific microarchitectural debug streams related to rename, writeback, issue/execute/writeback (IEW), and pipeline scoreboard behavior, the simulator generates a comprehensive execution log that represents the fine-grained dynamics of register allocation and reuse. This trace constitutes the primary data source for the subsequent analysis phase.

4.3 Trace-driven analysis with `OoOLyzer`

The captured execution logs are analyzed offline by `OoOLyzer`, which reconstructs dataflow and evaluates formally defined leakage predicates. The analyzer takes two primary inputs: the simulator trace and the corresponding ELF binary. The ELF file resolves symbol boundaries, aligns runtime and static PCs, and filters events for selected masked regions (e.g., Keccak). A configuration file specifies the mapping of architectural registers to logical shares and defines conflict pairs under the masking scheme.

Internally, `OoOLyzer` sequentially parses events, correlates renames and writebacks per physical register, and applies leakage predicates. It detects consecutive renames of conflicting shares, measures their temporal proximity, and—when both values are

available—computes their Hamming distance as a proxy for dynamic power variation due to register reuse, enabling trace-based reasoning about leakage.

4.4 Controlled parameterization and reproducibility

All experiments are executed under uniform microarchitectural parameters to isolate data-dependent effects from architectural noise. The out-of-order CPU configuration maintains a fixed rename queue, reorder buffer, and issue width across all workloads, ensuring that observed differences in trace behavior stem solely from workload characteristics. Clock frequencies for the core and memory subsystems are kept constant, and the memory model is configured for deterministic latency to simplify temporal comparison across runs.

Each simulation produces an execution log ranging from several megabytes to tens of megabytes, depending on the length of the workload and the level of instrumentation enabled. The tool automatically infers the ELF base address from the simulator's load information or, when unavailable, estimates it by aligning runtime program counters with static symbol ranges. This automatic calibration ensures that the runtime trace data can be reliably mapped to specific static instruction regions without manual intervention.

4.5 Analytical methodology

The post-simulation analysis proceeds in a structured and reproducible manner. For each workload:

1. Rename and writeback events are extracted and grouped by physical register index.
2. The Rename-Induced Leakage (RIL) predicate is evaluated to identify consecutive renames involving conflicting shares and their associated writeback transitions.
3. The IEW-Induced Dispatch Leakage predicate is applied, when available, to detect temporally overlapping share updates within the issue or writeback pipeline stages.
4. Quantitative metrics—including Hamming distance of value transitions, temporal gap between successive writes, and duration of conflicting co-residencies—are computed and stored in a structured report for further interpretation.

By employing a uniform trace-driven workflow, the methodology achieves a controlled and transparent evaluation of out-of-order leakage phenomena. Each experiment thus represents a closed loop—from low-level cryptographic assembly to cycle-accurate simulation and formal leakage detection—ensuring that all observed violations are traceable back to specific architectural interactions within the execution pipeline.

4.6 Case Study 1: Rename-Induced Transition Leakage (RIL) on Masked Toffoli

The renamer maps architectural destinations to entries in the physical-register file to avoid false data hazards and enable instruction-level parallelism. From a security perspective, this indirection introduces transient physical states that can temporarily violate share isolation between masked variables.

Formally, when two architectural registers r_a, r_b holding shares S_a, S_b are at different instants mapped to a single physical register P_i, and P_i receives two consecutive writes with values V_{old} and V_{new}, the observable transition magnitude is:

$$L_{\text{RIL}}(P_i) \;=\; \text{HD}\big(V_{\text{old}}, V_{\text{new}}\big).$$

Larger Hamming-distance transitions are prioritized for analysis because they correspond to greater simultaneous bit-line activity, which increases the likelihood of observable side-channel emissions in power or electromagnetic domains.

Masked Toffoli implementation. We analyze a two-share masked Toffoli gate implementation compiled for a 32-bit RISC-V target. The design separates computations into two logical domains: domain A and domain B, corresponding to the two masked shares of the same sensitive variable. Temporary registers are used to hold intermediate cross terms and are cleared immediately after use to maintain register uniformity.

A representative assembly excerpt (annotated with domain comments) is:

```
# domain A (first share)
and     t0, a0, a1        # t0 = a0 & b0
xor     a2, a2, t0        # update first share of c (arch a2)

...

# domain B (second share)
and     t0, a3, a4        # t0 = a1 & b1
xor     a5, a5, t0        # update second share of c (arch a5)

# restore temporaries
li      t0, 0
ret
```

This routine demonstrates the intended temporal separation between domain updates and the controlled restoration of temporary registers. The objective is to prevent intermediate overlaps of the two shares in architectural state; however, this guarantee may not hold in out-of-order hardware, where physical-register reuse and pipeline reordering can cause transient violations.

Experimental configuration and rationale. The masked Toffoli experiment served as the first validation of the proposed Rename-Induced Transition Leakage (RIL) detection methodology. To expose realistic yet stress-inducing leakage conditions, we executed the masked Toffoli routine on a gem5-based out-of-order RISC-V processor model configured to emulate a compact embedded-class OoO core under moderate resource pressure. Detailed rename, issue, and writeback activity was captured using gem5 debug flags (`Rename`, `FreeList`, `Writeback`, `IEW`, `Exec`, `O3PipeView`, `IQ`, `LSQ`, `Commit`, and `Scoreboard`), providing full microarchitectural visibility. The experiment employed a custom script with the following key parameters: 36 physical registers, a 64-entry reorder buffer (ROB), 16-entry issue queue (IQ), 16 kB L1 instruction and data caches, a 256 kB L2 cache with 60-cycle latency, and a 2 GHz core frequency.

This configuration mirrors a low-power OoO embedded processor while maintaining sufficient rename and scheduling pressure to reveal transient leakage. The small 36-entry physical register file enforces frequent register recycling, increasing rename collisions, while the moderate ROB and IQ depths balance instruction reordering with realistic backpressure. The memory hierarchy—with compact L1 caches and a higher-latency L2—extends in-flight instruction lifetimes, heightening temporal adjacency between dependent writebacks. The 2 GHz frequency approximates modern RISC-V and ARM-based embedded OoO cores, preserving architectural realism.

Overall, this setup establishes a representative stress environment for compact OoO designs, such as secure microcontrollers and IoT-class SoCs. Under these conditions, `OoOLyzer` successfully identified distinct rename-induced transitions, confirming that even small-scale, power-efficient OoO processors can exhibit structural recombination of masked shares when operating near resource limits.

Analyst-supplied semantic mapping and danger pairs. The semantic mapping $\mathcal{M}$ defines the correspondence between architectural registers and their logical share domains, while

the conflict set $\mathcal{C}$ enumerates pairs that should never co-reside within a single micro-architectural resource. For the Toffoli gate, the mapping and monitored conflict set were defined as:

```
arch_to_label = {10: "a0", 11: "a1", 12: "a2",
                 13: "a3", 14: "a4", 15: "a5"}
danger_pairs = { {"a0","a3"}, {"a1","a4"}, {"a2","a5"} }
```

`OoOLyzer` tracked and correlated **rename, writeback, and execution events** across these labels to reconstruct the full micro-architectural timeline of each instruction and to detect cases where different shares transiently influenced the same physical resource. Specifically, the framework maintains a synchronized view of three event streams:

- **Rename stream:** captures architectural-to-physical register mappings, enabling `OoOLyzer` to determine when two architectural destinations (e.g., `a2` and `a5`) are mapped to the same physical register.
- **Writeback stream:** records the data values written to the physical register file, allowing computation of the Hamming-distance transition between consecutive writes and identifying whether the two writes belong to conflicting shares.
- **Execution/IEW stream:** provides pipeline timing and serial-number context, which helps verify whether the two conflicting writebacks occurred adjacently in time, thus strengthening confidence in the detected RIL event.

By systematically cross-referencing these synchronized event streams, `OoOLyzer` establishes a direct linkage between the semantic layer—representing masked software shares—and the underlying micro-architectural execution dynamics. This integrated view enables precise identification of moments where the expected share isolation breaks down, particularly due to physical register reuse or transient overlaps introduced by speculative execution.

Representative diagnostic RIL event. A characteristic Rename-Induced Transition Leakage event detected in the Toffoli experiment was recorded at physical register P_{34}. The diagnostic summary is as follows:

> Total collapsed renames parsed: 22
> Total RegFile writes parsed: 55
> — Conflict event at physical register P_{34}:
> Rename (old): arch `x12`, label `a2`, runtime_pc = $0xC8$
> Rename (new): arch `x15`, label `a5`, runtime_pc = $0xD8$
> write_old: tick = 377000, val = `0x44444444`
> write_new: tick = 384000, val = `0x22222222`
> Consecutive writes on same PReg? YES; Δticks = 7000

This event satisfies the RIL predicate, meeting all structural and temporal conditions (conflicting register pair, contiguous write operations, and an acceptable inter-write interval). The measured Hamming distance between the two consecutive write values is considerably high, meaning that a significant fraction of bits within the physical register changed state simultaneously. Such large-scale bit transitions cause substantial instantaneous switching activity within the register file circuitry, which, on real hardware, can manifest as observable variations in dynamic power draw or electromagnetic emissions. Consequently, events exhibiting high Hamming distances are regarded as strong indicators of potential side-channel leakage, as they correspond to moments of intensified data-dependent switching behavior.

Conceptual interpretation. Figure 2 illustrates the temporal relationship between two rename operations (R_1 and R_2) mapped to the same physical register. The first rename maps `x12` (`a2`) to P_i, followed shortly by `x15` (`a5`). Their respective writebacks, W_1 and W_2, occur sequentially on the same hardware storage element, producing a data-dependent Hamming-distance transition $\mathrm{HD}(V_{\mathrm{old}}, V_{\mathrm{new}})$. This transition represents a structural leakage channel caused by transient reuse of the same physical register across different masked domains, an effect that OoOLyzer automatically detects and quantifies.

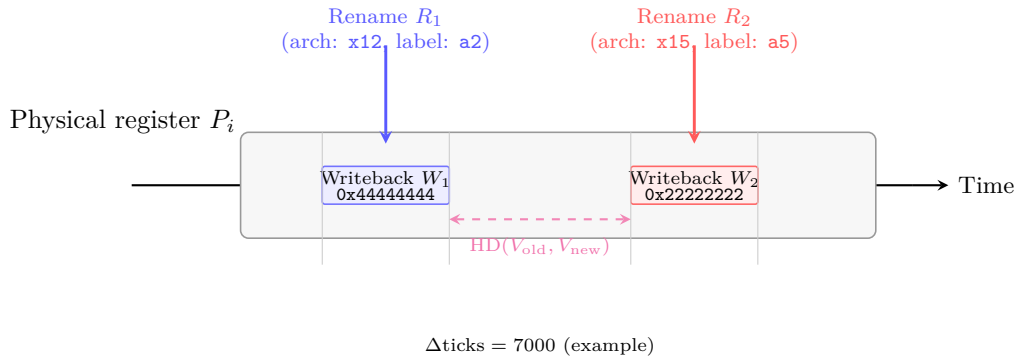

Figure 2: Conceptual timeline of Rename-Induced Transition Leakage (RIL). Two distinct shares (`x12` and `x15`) are renamed to the same physical register P_i, resulting in consecutive writebacks whose Hamming-distance transition forms the structural leakage channel.

Experimental validation through TVLA analysis. To empirically validate the Rename-Induced Transition Leakage (RIL) predicted by `OoOLyzer`, we performed a fixed-vs-random Test Vector Leakage Assessment (TVLA) on the masked Toffoli implementation. The goal was to verify whether the structural hazard detected—specifically, reuse of a single physical register for `a2` and `a5`—translates into measurable side-channel leakage on real hardware.

The experiment executed the masked Toffoli routine over 500 runs using two input classes: (i) fixed inputs (constant secret and public values) and (ii) random inputs (uniformly random shares). Simulated power traces were derived from Hamming-distance activity at writeback events, capturing the switching behavior when `a2` and `a5` wrote to the same physical register.

The resulting TVLA curve (Figure 3) exhibited a statistically significant deviation during the consecutive writeback window, with peaks exceeding the common leakage threshold ($|\mathrm{TVLA}| > 4.5$). This confirms that the identified register reuse event produces observable leakage even with only 500 traces, demonstrating that the structural transitions flagged by `OoOLyzer` correspond to physically measurable effects.

Overall, these results substantiate `OoOLyzer`'s analytical predictions: rename-induced transitions are not merely theoretical artifacts but can manifest as exploitable side-channel emissions, underscoring the need for RIL-aware verification and mitigation in masked software and OoO microarchitectural co-design.

4.7 Case Study 2: Rename-Induced Transition Leakage (RIL) on Masked PRESENT

We applied the same OoOLyzer methodology to a bitsliced masked implementation of PRESENT that follows the two-share threshold-masking construction proposed by Gaspoz et al. [GD23]. This implementation protects the S-box computations using domain-oriented

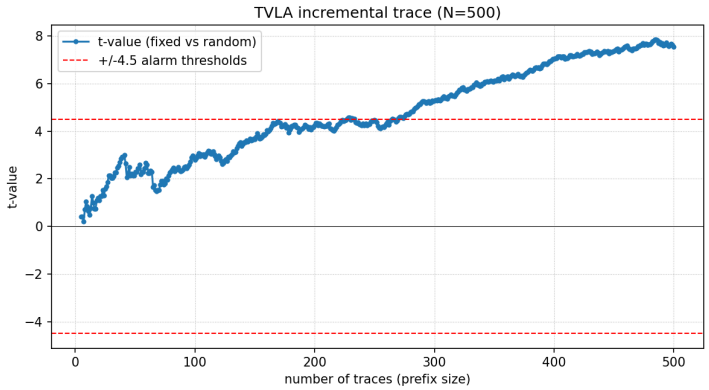

Figure 3: Fixed-vs-random TVLA result for the masked Toffoli routine. A clear leakage peak ($|\text{TVLA}| > 4.5$) is observed during the execution window where shares in `a2` and `a5` are consecutively written to the same physical register, validating the Rename-Induced Transition Leakage predicted by `OoOLyzer`.

Boolean masking, where each logical nibble of the internal state is distributed across two independent domains (`domain A` and `domain B`). The masking ensures that each share is processed in isolation and that all non-linear operations (notably the cubic terms of the PRESENT S-box) are decomposed into sequences of masked AND and XOR operations between the two domains.

The masked implementation adopts a bitsliced style, operating on 32-bit words where each bit position represents an independent S-box evaluation across 32 nibbles in parallel. The code maintains separate register banks for the two domains. The S-box computation is implemented through repeated masked Toffoli-like sequences and affine linear transformations, ensuring first-order resistance in the ideal (in-order) execution model. A simplified fragment from the analyzed RISC-V assembly is reproduced below, annotated to highlight the per-domain flow:

```
# Domain A (first share)
and     t0, a0, a1          # partial product for share A
xor     a2, a2, t0          # update masked output share A
xor     a2, a2, a0          # linear combination (domain A)
xor     a2, a2, a1          # completes share A transformation
...
# Domain B (second share)
and     t0, a3, a4          # partial product for share B
xor     a5, a5, t0          # update masked output share B
xor     a5, a5, a3
xor     a5, a5, a4
...
# Final recombination (linear diffusion)
ror     s1, a2, 1
xor     t1, s1, a5          # temporary holding inter-domain data
li      t1, 0               # clear temporary for uniformity
```

This snippet exemplifies how PRESENT enforces masking semantics at the software level: each share (`a0-a2` for domain A and `a3-a5` for domain B) is updated independently, and intermediates (`t0, t1, s1`) are cleared immediately after use. In a strictly in-order pipeline, the two domains would never overlap in time at any hardware node. However, out-of-order execution can invalidate these temporal separations.

Semantic mapping and conflict set for PRESENT. To monitor micro-architectural violations, we constructed an explicit mapping $\mathcal{M}$ from architectural registers to share labels, and a conflict set $\mathcal{C}$ that enumerates register pairs corresponding either to (i) matching domain pairs representing the same logical nibble, or (ii) temporary and restore registers that may transiently contain share fragments. The mapping and conflict-set fragment used for the PRESENT analysis is:

```
{
  "arch_map": {
    10: "a0", 11: "a1", 12: "a2", 13: "a3",
    14: "a4", 15: "a5",  9: "s1",  6: "t1",
    18: "s2",  7: "t2", 19: "s3", 28: "t3",
    20: "s4", 29: "t4"
  },
  "conflicts": [
    ["a0","a3"], ["a1","a4"], ["a2","a5"],
    ["s1","t1"], ["s2","t2"], ["s3","t3"], ["s4","t4"]
  ]
}
```

Why these pairs are targeted.

- **Domain pairing.** Each pair such as $\langle \texttt{a2}, \texttt{a5} \rangle$ corresponds to the same logical nibble represented in two independent domains. The masked code guarantees isolation in theory, but physical-register reuse by the renamer can cause both shares to be stored in the same physical register at adjacent times, violating non-completeness.
- **Temporary/auxiliary conflicts.** Registers like `s1`/`t1` are short-lived temporaries used in masked rotations or recombinations. Under OoO conditions, their values may coexist transiently in rename maps or forwarding paths with share registers, creating unintentional share overlap.

Representative consolidated run statistics and diagnostic excerpt. To verify that the observed Rename-Induced Leakage (RIL) was not an artifact of simplified models, we used parameters representative of a mid-scale out-of-order RISC-V core, reflecting realistic design trade-offs and moderate instruction-level parallelism (ILP).

The **physical register file** contained 192 entries—typical of embedded and mid-range OoO cores—providing ample temporaries while inducing realistic register pressure and turnover, naturally promoting rename-induced transitions between conflicting shares.

The **reorder buffer (ROB)** had 256 entries, enabling deep instruction windows and extensive reordering, which increase temporal overlap between share-updating instructions from different domains and promote transient co-residency and register reuse.

A **64-entry issue queue (IQ)** offers balanced throughput and complexity. While sustaining instruction-level parallelism, it also allows share-updating instructions from different domains to coexist within shared execution and forwarding resources, creating transient recombination points.

The **core frequency** was set to 2.5 GHz, typical for 28–45 nm OoO designs. The resulting short cycle times tighten overlaps across rename, execute, and writeback stages, making successive writes to the same physical register more likely before full charge dissipation, thereby enhancing the plausibility of measurable correlations.

Finally, the **cache hierarchy** included 32 kB L1 instruction/data caches and a 512 kB shared L2 cache, ensuring the masked workload remains on-chip and avoiding artificial serialization due to misses. The **L2 latency** of 30 cycles provides realistic memory delay

without excessive flushes, sustaining speculation and rename pressure consistent with practical OoO pipelines.

Collectively, these parameters emulate realistic pipeline behavior rather than synthetic stress conditions. The analysis revealed **571 RIL candidates**, concentrated in the masked S-box loops. This density confirms that, even under production-grade OoO configurations, the interaction of rename reuse, deep ROB buffering, and wide issue parallelism inherently exposes transient recombination of masked shares—demonstrating that the observed leakage is a fundamental property of modern OoO microarchitecture, not an experimental artifact.

The `OoOLyzer` analysis reported **571 Rename-Induced Leakage (RIL)** events in total, demonstrating that share-mixing at the physical level is widespread under high instruction-level parallelism. A representative top-ranked event (abbreviated) is:

— RIL Candidate (example) at physical register P_{14}:
Rename (old): arch `x11` label `a1`
Rename (new): arch `x14` label `a4`
write_old: tick = 32373200, val = `0x7ffff6cf`
write_new: tick = 32438000, val = `0x3`
Hamming distance (HD): 25 (width 32), Δticks = 64800

This event satisfies all RIL conditions: (1) both renames correspond to a conflict pair $\{a1, a4\}$, (2) the writes are contiguous in the physical-register stream, and (3) the tick difference falls within the configured temporal window. The large Hamming distance indicates that multiple bit-lines flipped in the same physical register, making it a plausible source of side-channel activity.

Why these shares are vulnerable and how mixing becomes exploitable. Although the masked PRESENT code is theoretically first-order secure, OoO scheduling undermines its isolation guarantees in two ways:

- *Physical-register reuse:* The renamer reassigns freed physical registers under high parallelism. When two share-updating instructions (e.g., `a2` and `a5`) are issued in quick succession, the same PReg can be reassigned before all microarchitectural state has been overwritten or aged out. In such cases, a physical register first stores a value from domain A ($v_A = S_A \oplus K_A$) and is later reused for domain B ($v_B = S_B \oplus K_B$). The corresponding hardware transition exhibits power or EM activity proportional to HD(v_A, v_B), effectively combining both shares and correlating with the true unmasked intermediate $S \oplus K$. This transition thus leaks a function of the secret state and round key that can be statistically exploited using first-order CPA or TVLA techniques.
- *Transient co-residency:* During the non-linear S-box computation, masked AND and XOR instructions on domain A and domain B may overlap in the issue queue, forwarding paths, or execution stages. These overlaps act as implicit probes within the microarchitecture, allowing speculative and out-of-order execution to momentarily mix masked values at the hardware level. Such events enable attackers to capture measurable correlations that bypass the expected need for second-order analysis, effectively downgrading the masking order.

In the analyzed trace, most of the 571 candidates appear in short bursts within the S-box layer, where multiple domain-specific AND/XOR chains are active. This shows that even semantically secure masked implementations can undergo share recombination on realistic OoO hardware. The affected share pairs (`s1-s4`, `t1-t4`) correspond to bitsliced 32-bit PRESENT state containers, and their reuse exposes transitions $(S_A \oplus K_A) \rightarrow (S_B \oplus K_B)$—directly linked to the unmasked intermediate $(S \oplus K)$. Thus, **each reuse**

event leaks a linear combination of state and key bits, making the vulnerability exploitable under a Hamming-distance leakage model.

4.8 Case Study 3: IEW-Induced Dispatch Leakage on Composable PINI Gadgets

PINI is defined relative to a static set of probe locations and assumes scheduling preserves share isolation. OoO processors create many transient probe points (forwarding networks, issue-queue ports, pipeline latches, speculative buffers) that are not modeled in the PINI framework. In traces, micro-architectural PINI violations appear as persistent co-residency of instruction serial numbers that originate from conflicting share-updating instructions within a single pipeline stage for a duration exceeding a small tick threshold. `OoOLyzer` reconstructs per-tick IEW occupancy and resolves SN→architectural sources using nearby rename events; when conflicting labels co-reside persistently, the analyzer reports a PINI candidate with stage, SN pair, PC context, and an estimated confidence.

PINI-AND (marker excerpts). To expose microarchitectural ordering, the gadget was instrumented with ELF-visible markers placed at the ALU producers for each z_{ij} and at the share-update sites. The RISC-V implementation of the composable PINI-AND gadget follows the adaptation described in [MP25], where the authors implemented the PINI masking approach for a 32-bit RV32IM architecture as part of a masked and shuffled Ascon permutation. We adopted the same RV32 assembly layout for our PINI-AND case study, retaining instruction-level semantics while extending it with marker instrumentation. Example fragments (abridged):

```
marker_z13_instr:
    xori t3, t1, -1
    and  t4, t3, t0
    xor  t5, t2, t0
    and  t5, t1, t5
    xor  t5, t5, t4    # t5 = z13

marker_z31_instr:
    xori t3, t1, -1
    and  t4, t3, t0
    xor  t5, t2, t0
    and  t5, t1, t5
    xor  t5, t5, t4    # t5 = z31
```

Analyzer output (condensed). Running the gem5 trace through the parser produced per-marker timing (rename / IQ-add / execute / memwrite / commit) and per-pair conclusions. A representative excerpt is shown below:

```
--- Marker info (inferred) ---
marker_z13: execute=37295, commit=37411
marker_z31: execute=38386, commit=38503
marker_z32: execute=40093, commit=40452
...
--- Final stores info ---
c0: execute=39630, commit=40455
c1: execute=39637, commit=40458
c2: execute=39738, commit=40461
c3: execute=40100, commit=40465
```

```
--- Execution ordering conclusions ---
Pair marker_z23 vs marker_z32:
  -> YES: final executed BEFORE contributor (execution-level preorder)
```

Short interpretation. The parser shows that, at IEW/execution granularity, some final share-update instructions (the "final" stores that update c_i) were observed executing *before* the ALU producer of a contributing z_{ji}. Although commits eventually respect architectural order (no commit-level inversion was detected), this IEW-level reordering demonstrates microarchitectural co-residency/reordering that violates the isolation assumption used by PINI proofs—specifically, an IEW-induced dispatch overlap that creates transient mixing of share-processing contexts and thus a potential leakage surface not captured by the original PINI model.

5 Conclusion

In this work, we ask a previously unanswered question in masking literature: how does characteristics of the hardware executor undermine the guarantees of algorithmic descriptions. We begin by identifying two leakage *signals*, and create an implementation agnostic taxonomy for out-of-order induced leakages: (1) Rename-Induced Transition Leakage (RIL); and (2) IEW-Induced Dispatch Leakage. We then provide an end-to-end methodology - `OoOLyzer` - that implements predicates for these leakages. Our evaluations of these leakage predicates are spread over three case studies. In the first two case studies - (1) Masked Tofolli gate implementation and (2) Masked PRESENT implementation - we demonstrate how Rename-Induced Transition Leakage (RIL) predicate is discovered by `OoOLyzer`, that introduces leakage by mapping architectural registers holding conflicting masking shares to the *same* physical register. Likewise, in the third case study - (3) PINI - we show how IEW-Induced Dispatch Leakage predicate is discovered by `OoOLyzer`, that introduces leakage by dispatching an *unsafe* instruction sequence and violating the otherwise provable PINI composability. We back up the discoveries of `OoOLyzer` by demonstrating Test Vector Leakage Assessment on the discovered leakages, further underscoring the general exploitability of `OoOLyzer`'s finding.

From a general perspective, this work puts into perspective the need to rethink side-channel protections (like masking, PINI composability) in terms of the *hardware* on which these implementations are expected to execute; since otherwise theoretically secure constructs become insecure when *hardware optimizations* (like the out-of-order optimizations considered in this work) play a role in execution. Our results are expected to further inform investigation that includes hardware execution characteristics while developing algorithmic/theoretical arguments for side-channel protections.

References

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[FGMDP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, Aug. 2018.

[GD23] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):155–179, Mar. 2023.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 3–32, Cham, 2021. Springer International Publishing.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[LCGD18] Yann LE CORRE, Johann GROSZSCHÄDL, and Dumitru-Daniel DINU. Micro-architectural power simulator for leakage assessment of cryptographic software on arm cortex-m3 processors. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Springer, April 2018.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD - a probing-based hardware leakage detection tool. Cryptology ePrint Archive, Paper 2022/965, 2022.

[MP25] Linus Mainka and Kostas Papagiannopoulos. Combined masking and shuffling for side-channel secure ascon on RISC-v. Cryptology ePrint Archive, Paper 2025/564, 2025.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 413–427, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. Prolead_sw: Probing-based software leakage detection for arm binaries. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):391–421, Jun. 2023.