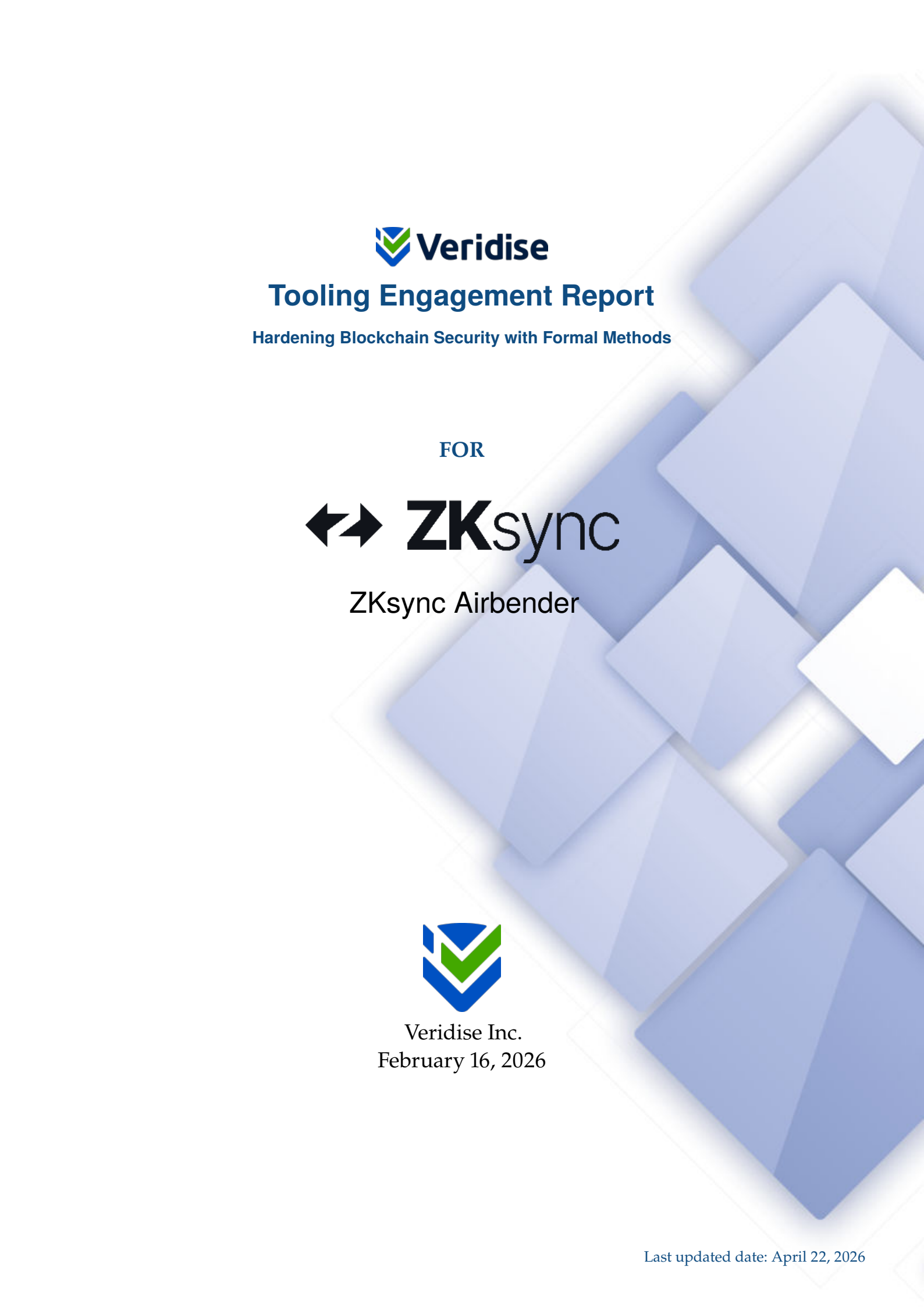

Tooling Engagement Report

Hardening Blockchain Security with Formal Methods

FOR

ZKsync Airbender

Veridise Inc.
February 16, 2026

Last updated date: April 22, 2026

► **Prepared For:**

Matter Labs
https://matter-labs.io/

► **Prepared By:**

Shankara Pailoor
Daniel Dominguez
Ian Neal

► **Contact Us:**

contact@veridise.com

► **Version History:**

April 27, 2026 V1
April 22, 2026 Initial Draft

Contents

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the Matter Labs team using AuditHub, enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through AuditHub. AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at `https://audithub.dev/case-studies/`.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of Veridise should be considered official Veridise reports.

`https://veridise.com/audits-archive/`

1 Executive Summary

From Feb. 16, 2026 to Apr. 15, 2026, Matter Labs engaged Veridise to conduct a tooling-based security assessment of their RISC-V zkVM, ZKsync Airbender. Unlike a traditional end-to-end code audit, this engagement emphasized formal constraint extraction and verification, Picus-based determinism analysis, fuzzing-based testing of selected proving and execution components, and targeted manual review of areas that were not fully covered by automation.

The assessment covered Picus-based determinism checks for selected unified proving and unrolled family circuits, together with fuzzing-based testing of the prover and related components. Veridise conducted the assessment over 21 person-weeks, with 3 security analysts reviewing the project over 7 weeks on commit 5ecb674.

Where tool coverage was incomplete, or where the automated analysis surfaced potentially security-relevant behavior, Veridise supplemented the tooling effort with focused manual review of the relevant code paths. Accordingly, this engagement should be understood as a scoped tooling and testing review with manual follow-up, rather than a comprehensive end-to-end manual audit of the entire repository.

Project Summary ZKsync Airbender is a STARK-based RISC-V proving system that combines a constrained RV32I+M-style execution model with a family of AIR circuits, a multi-stage proving pipeline, and a recursive verification stack. The repository contains the core components needed to compile, simulate, prove, and verify RISC-V execution traces for the ZKsync ecosystem, including machine-operation circuits, circuit optimization/compiler infrastructure, witness generation logic, CPU and GPU provers, generated single-circuit verifiers, and higher-level statement verifiers that compose many chunk proofs into one execution claim.

At a high level, ZKsync Airbender proves execution by modeling a RISC-V machine over the Mersenne31 field and enforcing instruction semantics with AIR constraints. Rather than keeping a large monolithic CPU state in every row, the design pushes much of the global consistency work into shared arguments: RAM/register accesses are enforced through a permutation-style memory argument, delegation requests are handled through a separate set-equality-style argument, and control-flow continuity is enforced through machine-state permutation logic. This lets the system reduce the amount of explicit state carried inside each circuit while still proving that all pieces describe one coherent execution.

Circuit Architecture. In the context of this engagement, Airbender has two primary circuit architectures, which Matter Labs referred to as unified and unrolled. The unified architecture is centered on a reduced-machine circuit that is more suitable for recursive verification and higher-layer proof composition. The unrolled architecture specializes execution into opcode-family circuits in order to improve proving efficiency at the base layer.

Rather than assigning one circuit per opcode, the unrolled architecture groups multiple related opcodes into a single circuit family. For example, arithmetic-style instructions are grouped into circuits such as `add_sub_lui_auipc_mop`, control-flow and comparison instructions are grouped into `jump_branch_slt`, shift, binary, and CSR-related behavior is grouped into `shift_binary_csr`, and memory operations are split across dedicated word and subword load/store families. These grouped opcode-family circuits reduce wasted logic while still allowing the verifier to enforce one coherent execution through shared memory, delegation, and machine-state arguments.

The two architectures are complementary layers of the same proving stack rather than mutually exclusive alternatives. In the current pipeline, the base layer is proved with the unrolled opcode-family circuits, the first recursion stage still uses a reduced unrolled configuration, and a higher recursion stage uses the unified reduced-machine circuit. In other words, the unrolled architecture is not limited to the base layer; it is also used in an intermediate recursion layer before proofs are lifted into the unified recursion-friendly form.

Prover and Verifier Pipeline. ZKsync Airbender also contains a substantial proving and verification pipeline around those circuits. A lightweight simulator/tracer executes programs and prepares witness data, after which the prover runs a five-stage STARK workflow that commits trace data, constructs lookup and permutation arguments, evaluates quotient polynomials, applies DEEP/FRI reductions, and produces the final proof. The repository contains both CPU and GPU proving paths, as well as generated verifier crates that verify individual circuit proofs and a full-statement verifier that checks setup consistency, transcript consistency, accumulator closure, final register/PC claims, and recursive chaining metadata across many proof fragments.

Execution model and trust assumptions. ZKsync Airbender also makes a deliberate trust-model simplification: programs are proved against a fixed bytecode image that is preprocessed into ROM and decoder tables, rather than against adversarial code loaded at runtime into generic executable RAM. This allows some checks that might appear as explicit local decoder constraints in a more general CPU arithmetization to instead be enforced through ROM preprocessing, opcode-family partitioning, restricted machine configurations, and global consistency arguments; unsupported behaviors are typically made unprovable rather than dynamically trapped. Delegation circuits serve as protocol-level precompiles for operations such as bigint and hash-related logic, while the recursive verifier path runs over a narrower machine profile specialized for proof verification workloads.

In summary, ZKsync Airbender should be understood as a specialized proving architecture that combines: (1) instruction-family AIR circuits, (2) global RAM/delegation/machine-state arguments, (3) a chunked STARK/FRI prover, and (4) recursive full-statement verification. The result is a system that can transform constrained RISC-V execution into efficiently provable statements that are then recursively composed into higher-level zkSync proofs.

Code Assessment. The ZKsync Airbender developers provided the source code of the ZKsync Airbender repository for review. The codebase appears to be largely original Rust code written by the ZKsync Airbender developers, together with generated verifier artifacts and standard third-party dependencies.

The repository contains substantial in-repo documentation, including top-level and crate-level READMEs, architecture and design notes, repository-layout documentation, tutorials, and end-to-end usage guidance. To facilitate the Veridise security analysts' understanding of the system, the ZKsync Airbender developers shared this documentation and also answered targeted design and implementation questions during the engagement.

From a reviewability perspective, ZKsync Airbender is a sophisticated and highly optimized proving system whose correctness depends on interactions across circuit builders, generated verifier code, full-statement composition logic, recursion layers, and host-side tooling. As a result, some important security and correctness properties are distributed across multiple layers of the implementation rather than being obvious from any one local code path in isolation. This increased the effort required to validate certain invariants and helps explain why relatively small local mistakes could have broader soundness implications. It also affected the formal verification workflow: in several places, the Veridise analysts needed to encode explicit environmental

and trust assumptions in the Picus/LLZK model so that the automated analysis reflected Airbender's intended execution model rather than the raw constraints in isolation. Without those assumptions, the extracted constraints could produce false positives that were outside the system's intended threat model.

The source code also contained a broad but primarily component-level test suite, including unit tests across prover, simulator, verifier, and generated-circuit crates, as well as ISA tests and recursive proving smoke tests. The repository's CI configuration further indicates that the developers use supporting engineering checks such as `cargo fmt`, `cargo-deny`, generated-artifact consistency checks, and multiple build and test workflows.

Summary of Issues Detected. The security assessment uncovered 6 issues, of which 3 are assessed to be of critical severity by the Veridise analysts. Specifically, the most severe findings affected core proving soundness: one unrolled circuit omitted a required `enforce_all()` call, leaving deferred arithmetic relations unconstrained and enabling forged branch, SLT, link-register, and next-PC behavior (V-AIRBENDER-VUL-001); two additional verifier-composition flaws allowed constituent proofs to be checked under inconsistent external Fiat-Shamir and machine-state permutation challenges, undermining the soundness of the composed execution claim (V-AIRBENDER-VUL-002, V-AIRBENDER-VUL-003).

The Veridise analysts identified one low-severity issue (V-AIRBENDER-VUL-004), in which a decoder variable mismatch broke the binding of `rs2` bits to the fetched instruction in a currently unused circuit, as well as two warnings (V-AIRBENDER-VUL-005, V-AIRBENDER-VUL-006) involving out-of-bounds instruction-fetch and RAM accesses in the transpiler VM under conditions that violate Airbender's trusted-bytecode assumptions. The ZKsync Airbender developers resolved all critical findings during the engagement and acknowledged the remaining lower-severity issues, which they did not prioritize for remediation because they either affect currently unused code paths or fall outside the intended trusted-bytecode execution model.

Summary of Verification Results. A central focus of this engagement was the use of Picus to perform formal determinism checks over extracted Airbender circuits, including per-opcode specializations, individual operation implementations, parameterized operation families, and selected delegation circuits. Across the analyzed targets, Picus initially verified 19 components, falsified 2, and left 5 unresolved. The two falsified targets corresponded to genuine bugs in the unrolled jump/branch/SLT circuit and the bytecode decoder (V-AIRBENDER-VUL-001, V-AIRBENDER-VUL-004), both of which were later fixed and successfully re-verified. This brought the final number of verified targets to 21, with the remaining unresolved cases concentrated in larger whole-circuit and delegation-circuit analyses. We describe the results in more detail in Chapter 7.

Recommendations. After this tooling-focused assessment, the Veridise analysts identified steps that would materially improve assurance around ZKsync Airbender's circuit and verifier glue code.

Consolidate verifier composition logic. The unrolled and unified verifiers implement nearly identical logic with minor variations, increasing the risk of inconsistent fixes and missed invariants as seen in (V-AIRBENDER-VUL-003). Refactor shared functionality into common helpers or data structures, and enforce critical invariants—such as challenge equality, transcript binding, and accumulator continuity—through these abstractions rather than duplicated code.

Modularize large security-critical implementations. Several parts of the codebase, especially in the circuit-construction and verifier-composition layers, are implemented as large functions that combine multiple logical responsibilities in one place. This increases review effort and makes it harder to isolate which invariants are established locally versus assumed from surrounding code. The Veridise analysts recommend gradually decomposing these large routines into smaller modules with clearer interfaces and narrower responsibilities. This would improve maintainability, make future fixes easier to apply consistently, and reduce the chance that local changes have unintended effects on adjacent security-critical logic.

Reduce API footguns in optimized circuit construction. The missing `enforce_all()` issue suggests that the current optimized-circuit API makes it too easy to accumulate deferred relations and forget to materialize them. The developers should consider changing this interface so that finalization is mandatory, as they once did previously with a drop guard which ensured `enforce_all()` was called before the guard dropped.

Continue integrating formal tooling into the development workflow. This engagement found meaningful issues through Picus-based analysis, fuzzing, and targeted manual follow-up. The analysts therefore recommend continuing to expand Picus coverage, preserving regression harnesses for resolved findings, and rerunning these checks after circuit, verifier, or generator changes. For a system as optimization-heavy as Airbender, automated regression checks provide stronger long-term assurance than manual review alone.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
ZKsync Airbender	`5ecb674`	Rust	RISC-V zkVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 16–Apr. 15, 2026	Manual & Tools	3	21 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	3	3	3
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	1	0
Warning-Severity Issues	2	2	0
Informational-Severity Issues	0	0	0
TOTAL	6	6	3

Table 2.4: Category Breakdown.

Name	Number
Logic Error	3
Memory safety	2
Underconstrained Circuit	1

Table 2.5: Picus Verification Outcome Summary.

Category	Count
Circuits Initially Verified	19
Circuits Initially Falsified	2
Unknown	5
Verified After Fix	2
Total Verified	21
TOTAL	26

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped as a tooling-focused security assessment of ZKsync Airbender's source code rather than a comprehensive end-to-end manual audit of the entire repository. The primary objective was to apply and extend formal verification, static analysis, and fuzzing workflows to selected Airbender circuits and supporting proving infrastructure, and to use targeted manual review to validate and interpret the behaviors surfaced by those tools. In particular, the assessment focused on selected circuit logic, proving and verification components, and bytecode-execution tooling relevant to the agreed Picus and fuzzing workstreams.

During the assessment, the security analysts aimed to answer questions such as:

- Are selected Airbender circuits deterministic and free of underconstrained behavior under the system's intended execution and trust assumptions?
- Can the planned fuzzing campaigns uncover crashes, malformed states, unexpected transcript behavior, or other security-relevant deviations in selected executor and prover components?
- Do selected proving and verification components behave consistently with the protocol's intended trust assumptions and execution model when analyzed through automated tooling and targeted manual validation?
- Where automated analysis is inconclusive or requires additional assumptions, what manual follow-up is necessary to distinguish real security issues from behaviors outside the intended threat model?

3.2 Security Assessment Methodology & Scope

To address the questions above, the security assessment combined automated formal methods, static analysis, dynamic testing, and targeted manual review. In particular, the assessment relied on the following techniques:

- *Formal verification.* Security analysts used Picus to analyze determinism and related underconstraint risks in selected Airbender circuits and extracted LLZK models. Picus was used both to prove determinism where feasible and to identify cases where additional modeling assumptions or manual review were required. More details about how Picus was used are provided in Chapter 7.
- *Fuzzing/property-based testing.* Security analysts developed and ran fuzzing and invariant-based tests for selected bytecode-execution, prover, and verifier-adjacent components to determine whether they could deviate from expected behavior, crash, or violate important correctness properties. Additional detail on this testing effort is provided in the dedicated fuzzing section of this report.
- *Targeted manual review.* Security analysts performed focused manual review of selected circuits, verifier-composition logic, and related proving components to validate design assumptions, investigate suspicious behaviors surfaced by tooling, and assess exploitability

of identified issues. This manual work was selective and tool-guided rather than a complete line-by-line review of the full repository.

Scope. The scope of this security assessment is best understood in terms of the specific tooling workstreams defined for the engagement, rather than as a uniform review of the entire ZKsync Airbender repository.

For the formal-methods portion of the engagement, the scope centered on Picus/LLZK-based determinism analysis of selected Airbender circuit components. This included extractor development, selected unrolled instruction-family circuits, selected legacy instruction implementations, selected whole-circuit state-transition logic, and selected delegation/precompile components, with targeted manual follow-up where full automation was incomplete or required additional assumptions. In particular, the circuits defined in the following files were in scope:

1. `cs/src/machine/ops/unrolled/decoder/decoder_circuit.rs` (untrusted bytecode decoder)
2. `cs/src/machine/ops/unrolled/add_sub_lui_auipc_mop.rs` (`apply_add_sub_lui_auipc_mop`)
3. `cs/src/machine/ops/unrolled/jump_branch_slt.rs` (`apply_jump_branch_slt`)
4. `cs/src/machine/ops/unrolled/load_store_subword_only.rs` (`apply_subword_only_load_store`)
5. `cs/src/machine/ops/unrolled/load_store_word_only.rs` (`apply_word_only_load_store`)
6. `cs/src/machine/ops/unrolled/load_store.rs` (`apply_load_store`)
7. `cs/src/machine/ops/unrolled/mul_div.rs` (`apply_mul_div`)
8. `cs/src/machine/ops/unrolled/shift_binary_csr.rs` (`apply_shift_binop_csrrw`)
9. `cs/src/machine/ops/unrolled/reduced_machine_ops.rs` (`apply_reduced_machine_circuit` and reduced-machine sequencing logic)
10. `cs/src/machine/ops/add_sub.rs` (`AddOp` and `SubOp`)
11. `cs/src/machine/ops/binops.rs` (`BinaryOp`)
12. `cs/src/machine/ops/shift.rs` (`ShiftOp`)
13. `cs/src/machine/ops/mul_div.rs` (`MulOp` and `DivRemOp`)
14. `cs/src/machine/ops/load.rs` (`LoadOp`)
15. `cs/src/machine/ops/conditional.rs` (`ConditionalOp`)
16. `cs/src/machine/ops/store.rs` (`StoreOp`)
17. `cs/src/machine/ops/lui_auipc.rs` (`LuiOp` and `AuiPc`)
18. `cs/src/machine/ops/jump.rs` (`JumpOp`)
19. `cs/src/machine/ops/mop.rs` (`MopOp`)
20. `cs/src/machine/machine_configurations/full_isa_no_exceptions/optimized_state_transition.rs` (`optimized_base_isa_state_transition`)
21. `cs/src/delegation/bigint_with_control/mod.rs`
22. `cs/src/delegation/blake2_round_with_extended_control/mod.rs`
23. `cs/src/delegation/blake2_single_round/mod.rs`
24. `cs/src/delegation/keccak_special5/mod.rs`

For the fuzzing and dynamic-testing workstreams, the scope was narrower and split between execution-oriented testing and prover-oriented testing. On the execution side, the assessment covered the bytecode/VM pipeline, including differential fuzzing against a Unicorn-based oracle and crash-oriented testing of executor and transpiler behavior. On the prover side, the assessment covered selected unrolled prover stages under `prover/src/prover_stages/unrolled_prover/`, including `mod.rs`, `stage2.rs`, `stage_2_shared.rs`, `stage_2_ram_shared.rs`, `stage3.rs`, and the quotient-part logic under `prover/src/prover_stages/unrolled_prover/quotient_parts/`. These

campaigns emphasized crash detection, malformed-state discovery, invariant checking, behaviors related to prover correctness, and differential validation of execution behavior against an external oracle (Unicorn). See Chapter 6 for more details.

Where these tooling efforts surfaced potentially security-relevant behavior, the Veridise analysts also performed targeted manual review of code which was unable to be verified by Picus, and core verifier logic to make sure that assumptions made in the Picus verification effort were actually enforced by the verifier. Components outside these defined workstreams were not reviewed exhaustively.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions

In addition to assuming that out-of-scope components behave correctly, Veridise analysts assumed the following properties when modeling security for ZKsync Airbender.

- The bytecode/ROM image being proved is fixed in advance and correctly preprocessed into the ROM and decoder tables expected by the circuits. Airbender does not model arbitrary runtime-loaded executable code in generic RAM.
- For the Picus/LLZK work stream, the global permutation-style arguments used to connect rows and circuit families were treated as trusted outer mechanisms. In particular, the formal verification work assumed the memory-permutation machinery and related outer proof-system checks correctly enforce cross-row and cross-circuit memory consistency.
- Similarly, the machine-state and other shared cross-proof consistency arguments enforced by the outer proving system were treated as correct unless they were explicitly brought into scope by targeted manual review. The formal-methods work therefore focused primarily on determinism and local under-constraint properties of the selected circuit logic rather than re-proving the full soundness of the outer composition layer.
- Airbender's circuit model does not attempt to support every behavior of a fully general RISC-V machine. Some behaviors are intentionally excluded for efficiency (signed division), and others are assumed not to occur under the expected compiler and execution environment. In many such cases, the intended behavior is that execution becomes unprovable rather than being handled through explicit in-circuit trap logic.
- Some inputs to the verification pipeline, including non-deterministic advice values and setup-related data, are provided externally rather than being fixed directly in the verifier code. The security model therefore assumes that these inputs are supplied consistently with the intended proving statement, and that any downstream system relying on final output commitments to identify the proven program or setup validates those commitments correctly.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AIRBENDER-VUL-001	Missing ‘enforce_all()‘ in Unrolled . . .	Critical	Fixed
V-AIRBENDER-VUL-002	Unrolled Statement Verifier Only . . .	Critical	Fixed
V-AIRBENDER-VUL-003	Verifier Failed To Enforce Continuity of . . .	Critical	Fixed
V-AIRBENDER-VUL-004	Wrong Decoder Variable Breaks Binding . . .	Low	Acknowledged
V-AIRBENDER-VUL-005	Out of bounds read while fetching instruction	Warning	Acknowledged
V-AIRBENDER-VUL-006	Out of bounds reads and writes on RAM . . .	Warning	Acknowledged

5.1 Detailed Description of Issues

5.1.1 V-AIRBENDER-VUL-001: Missing 'enforce_all()' in Unrolled 'jump_branch_slt' Leaves Deferred Arithmetic Constraints Unenforced

Severity	Critical	**Commit**	5ecb674
Type	Underconstrained Circuit	**Status**	Fixed
Location(s)	`cs/src/machine/ops/unrolled/[...]/jump_slt_branch.rs:364`		
Confirmed Fix At	zksync-airbender/pull/212, 77e979ed		

Description In the affected version of the unrolled `jump_branch_slt` family circuit, `apply_jump_branch_slt()` built the family logic through an `OptimizationContext` but did not call `opt_ctx.enforce_all(cs)` before consuming the derived outputs. This is critical because the family relies on deferred optimized relations for its core semantics:

- branch/SLT comparison results via `append_sub_relation()` (line 106),
- JAL/JALR link-register value `(pc + 4)` via `append_add_relation()` (line 140),
- next-PC computation for `SLT/SLTI`, `JAL/JALR`, and taken/skipped branches via repeated `append_add_relation()` (line 251).

Crucially, the optimization context does not immediately emit the final arithmetic constraints, but only materializes them using `OptimizationContext::enforce_all()` (line 1803). If that call is omitted, the comparison registers, carry/overflow flags, and next-PC scratch registers are no longer tied to the intended arithmetic over `rs1`, `rs2`, `imm`, and `pc`.

As a result, a malicious prover can:

- forge branch comparison outcomes and therefore choose taken vs not-taken behavior inconsistent with the real operands,
- forge SLT/SLTI results by driving the comparison witness into a desired condition-lookup output,
- forge JAL/JALR link-register writes because the shared pc + 4 scratch register is unconstrained,
- forge the cycle-end PC:
  - for JAL/JALR and taken branches, to an arbitrary aligned target subject only to the residual cleanup constraint,
  - for SLT/SLTI and skipped branches, to an arbitrary 32-bit value rather than the required pc + 4.

Recommendation Add `opt_ctx.enforce_all` after `cs.set_values(value_fn)` in the function.

Impact As described above, a malicious prover can:

- forge branch comparison outcomes and therefore choose taken vs not-taken behavior inconsistent with the real operands,
- forge SLT/SLTI results by driving the comparison witness into a desired condition-lookup output,

- forge JAL/JALR link-register writes because the shared pc + 4 scratch register is unconstrained,
- forge the cycle-end PC:
  - for JAL/JALR and taken branches, to an arbitrary aligned target subject only to the residual cleanup constraint,
  - for SLT/SLTI and skipped branches, to an arbitrary 32-bit value rather than the required pc + 4.

Developer Response The developers have applied the recommended fix.

5.1.2 V-AIRBENDER-VUL-002: Unrolled Statement Verifier Only Transcript-Binds External Challenges for One Circuit Family

Severity	Critical	**Commit**	5ecb674
Type	Logic Error	**Status**	Fixed
Location(s)	`full_statement_verifier/src/unrolled_proof_statement.rs`		
Confirmed Fix At	zksync-airbender/pull/258, 4844b40a		

Background `verify_full_statement_for_unrolled_circuits()` is the top-level verifier for the unrolled proving system. Its job is not to verify a single proof in isolation, but to verify and compose:

- multiple unrolled opcode-family proofs,
- init/teardown proofs,
- and optional delegation/precompile proofs,

into one final execution statement.

As it does so, it maintains a Fiat-Shamir transcript by absorbing public execution data and proof commitments, including:

- final register values and timestamps,
- final PC and timestamp,
- per-family circuit identifiers,
- and each proof's memory commitment caps.

At the end of this process, the verifier finalizes the transcript and derives the external random challenges used by the recursive arguments. The core process is illustrated in the following code snippet taken from the function:

Why the challenges matter These externally derived challenges are security-critical. They randomize the main global arguments used to prove soundness across the composed execution:

- memory_challenges randomize the memory multiset/grand-product argument,
- delegation_challenges randomize the delegation request/response argument,
- machine_state_permutation_challenges randomize the PC/timestamp permutation argument that links the machine state across chunks.

These values must be derived from the final transcript so that the prover cannot choose them after seeing the witness and commitment structure. If some subproofs are verified under challenges that are not forced to equal the transcript-derived values, then those subproofs are no longer verified under the intended Fiat-Shamir transform.

Issue Details In the implementation, this global transcript-binding invariant was not fully enforced. The verifier reused two scratch proof-output buffers, `proof_output_0` and `proof_output_1`, while iterating across many distinct unrolled circuit families as seen in the following code snippet:

```
let mut proof_output_0: ProofOutput<...> = MaybeUninit::uninit().assume_init();
let mut proof_output_1: ProofOutput<...> = MaybeUninit::uninit().assume_init();

for ((circuit_family, capacity, verifier_fn), setup) in circuits_families_verifiers
    .iter()
    .zip(circuits_families_setups.iter())
{
    ...
    for circuit_sequence in 0..num_circuits {
        let (current, previous) = if circuit_sequence & 1 == 0 {
            (&mut proof_output_0, &proof_output_1)
        } else {
            (&mut proof_output_1, &proof_output_0)
        };

        (verifier_fn)(current, &mut state_variables);
        ...
    }
}
```

Within each family, the verifier checked challenge continuity only against the immediately previous proof of the same family:

```
if circuit_sequence > 0 {
    assert_eq!(previous.memory_challenges, current.memory_challenges);
    assert_eq!(
        previous.machine_state_permutation_challenges,
        current.machine_state_permutation_challenges
    );
}
```

This enforces same-family consistency, but only locally. It does not ensure that all families use the same transcript-derived challenge tuple. At the end of verification, the verifier finalized the transcript, derived expected_challenges, and compared them only against `proof_output_0`:

```
let expected_challenges =
    ExternalChallenges::draw_from_transcript_seed_with_state_permutation(
        memory_seed,
        MEMORY_DELEGATION_POW_BITS,
        pow_challenge,
    );

assert_eq!(
    expected_challenges.memory_argument,
    proof_output_0.memory_challenges
);
assert_eq!(
    expected_challenges
        .machine_state_permutation_argument
        .unwrap_unchecked(),
    proof_output_0.machine_state_permutation_challenges[0]
);
```

Because `proof_output_0` was a reused scratch buffer, this final check only bound whatever family happened to be stored at the end of the loop to the Fiat-Shamir transcript. Other circuit

families were only required to remain internally consistent, but were not forced to use the same transcript-derived external challenges. This is a soundness issue because the composed verifier is intended to establish that all the constituent proofs together describe one valid execution under a shared Fiat-Shamir transcript. That claim only holds if every family participating in the composed proof uses the same transcript derived values.

Recommendation The verifier should maintain a canonical challenge tuple for the entire composed statement and require every main-family proof, init/teardown proof, and delegation proof to match it. Only after enforcing that global equality should it compare the canonical tuple against the transcript-derived `expected_challenges`.

Impact A malicious prover can cause the verifier to accept malformed multi-family proofs whose constituent family proofs correspond to inconsistent executions, thereby certifying an invalid overall RISC-V execution.

Developer Response The developers have modified the core loop to check the first challenge of each family against the transcript-derived `expected_challenges`. This, along with the continuity checks ensures all the challenges are the same as the transcript-derived one.

5.1.3 V-AIRBENDER-VUL-003: Verifier Failed To Enforce Continuity of Machine-State Permutation Challenges Across Proofs

Severity	Critical	**Commit**	5ecb674
Type	Logic Error	**Status**	Fixed
Location(s)	`full_statement_verifier/src/` ▶ `unified_circuit_statement.rs:124` ▶ `unrolled_proof_statement.rs:287`		
Confirmed Fix At	zksync-airbender/pull/258, 4844b40a		

Background The full-statement verifiers are responsible for composing many lower-level proofs into one final execution claim.

- `verify_unified_circuit_statement()` verifies chunks of the unified reduced-machine circuit.
- `verify_full_statement_for_unrolled_circuits()` verifies multiple unrolled opcode-family proofs, plus init/teardown and optional delegation proofs.

In both paths, the verifier must ensure that all constituent proofs were verified under one coherent set of external Fiat-Shamir challenges. These challenges parameterize the recursive global arguments used to prove soundness across proof boundaries.

Significance of the Machine-State Challenges One of the key external challenge families is `machine_state_permutation_challenges`. These challenges randomize the permutation argument that ties together machine-state continuity, in particular the PC/timestamp accumulator used to connect the beginning and end of the execution.

This is visible at the end of the verifier, where the machine-state contribution is folded into the grand product using the proof output's machine-state challenges:

```
let machine_state_contribution =
    prover::definitions::produce_pc_into_permutation_accumulator_raw(
        INITIAL_PC,
        split_timestamp(INITIAL_TIMESTAMP),
        final_pc,
        (final_ts_low, final_ts_high),
        &proof_output_0.machine_state_permutation_challenges[0].
    linearization_challenges,
        &proof_output_0.machine_state_permutation_challenges[0].additive_term,
    );
grand_product_accumulator.mul_assign(&machine_state_contribution);
```

If different chunks or families are allowed to use different `machine_state_permutation_challenges`, then the verifier is no longer composing one single permutation argument. It is multiplying together accumulators that were randomized under different challenge tuples.

Main Issue Both verifier paths enforced continuity for some external challenges but omitted the corresponding equality check for `machine_state_permutation_challenges`.

Recommendation We recommend adding the missing check:

```
// missing in the affected version
assert_eq!(
    previous.machine_state_permutation_challenges,
    current.machine_state_permutation_challenges
);
```

in both verifiers.

Impact A malicious prover can construct a composed proof in which different chunks or circuit families use inconsistent machine-state permutation challenges. This breaks the soundness of the PC/timestamp continuity argument and causes the verifier to accept malformed composed proofs that do not correspond to one coherent RISC-V execution.

Developer Response The developers have applied the recommended fix.

5.1.4 V-AIRBENDER-VUL-004: Wrong Decoder Variable Breaks Binding of rs2 Bits to the Fetched Instruction

Severity	Low	**Commit**	5ecb674
Type	Logic Error	**Status**	Acknowledged
Location(s)	`cs/src/machine/ops/unrolled/[...]/decoder_circuit.rs:98`		
Confirmed Fix At	N/A		

Description The decoder circuit translates raw bytecode instructions into the decoded fields used by the RV32IM execution circuits. Each instruction is a 32-bit word represented as two field elements, each range-constrained to 16 bits, corresponding to the low and high 16-bit halves of the instruction. For the high 16-bit half, the decoder is intended to enforce the following layout:

- `rs1_high` from bits `[19:16]`,
- `rs2` from bits `[24:20]`,
- `imm[10:5]` from bits `[30:25]`,
- and the sign bit from bit `[31]`

The decomposition constraint for this high-word layout (shown below) incorrectly used `rs1_from_decoder` where it should have used `rs2_from_decoder`. As a result, bits `[24:20]` of the instruction were constrained against the wrong decoded field. This breaks the binding between the fetched instruction word and the decoded `rs2` value.

```
// and we do not need sign bit, as we can span a linear constraint on it
// insn_high <=> rs1_high: [19:16], rs2: [24:20], imm[10-5]: [30:25], imm12: [31]
let mut sign_bit_constraint = {
    Constraint::from(high_insn)
        - rs1_high.clone()
        - Term::from(rs1_from_decoder) * Term::from(1 << 4)
        - Term::from(imm10_5) * Term::from(1 << 9)
};
```

Recommendation Use `rs2_from_decoder` instead of `rs1_from_decoder` in `sign_bit_constraint`.

Impact This allows the proof system to decouple execution semantics from the actual bytecode. Downstream unrolled execution circuits consume the decoder outputs to choose source registers and construct immediates, so an attacker may be able to:

- execute an instruction using a forged rs2 register index,
- and for I/J formats, execute with a forged immediate assembled from unconstrained rs2-derived bits,
  even though the fetched instruction word in ROM encodes different values.

Developer Response The developers have acknowledged the behavior but have not prioritized the fix since the circuit is currently not used.

5.1.5 V-AIRBENDER-VUL-005: Out of bounds read while fetching instruction

Severity	Warning	**Commit**	5ecb674
Type	Memory safety	**Status**	Acknowledged
Location(s)	`riscv_transpiler/src/vm/simple_tape.rs:20-21`		
Confirmed Fix At	Will not fix		

Vulnerability Description The VM stores decoded instructions in a boxed slice and accesses them using unchecked indexing guarded only by a `debug_assert!`. Because debug_assertions are disabled by default in release builds, this bounds check is not enforced in production configurations.

If execution advances past the end of the decoded instruction list, the VM performs an out-of-bounds read on the boxed slice via unchecked access. As a result, memory beyond the end of the slice is reinterpreted as though it were a decoded RISC-V instruction.

This condition can occur in multiple scenarios:

- During normal execution if the guest program does not conform to the expected ABI and its final instruction is not an infinite loop. In this case, execution may continue past the last decoded instruction.
- If the guest program explicitly sets the program counter to an address outside of the `.text` section, causing the VM to fetch instructions from a location beyond the decoded instruction list.

Developer response The developers acknowledge the out-of-bound reads but will not fix because they should not occur under their trusted bytecode assumption.

In most cases this results in a panic because the resulting value does not correspond to a valid instruction. However, unintended instruction execution is theoretically possible if the surrounding memory happens to match a valid decoded instruction.

Impact The primary impact of this issue is an **out-of-bounds forward memory read** during instruction fetch. This impact is limited in several ways:

1. **Sequential out-of-bounds access:**
   The VM can only read memory located after the instruction slice in memory as execution advances. It does not enable arbitrary memory reads or backward access.
2. **Read data is not raw machine code:**
   The out-of-bounds read does not fetch raw RISC-V instruction bytes. Instead, memory is interpreted as though it were an element of the Rust `Instruction` type stored in the boxed slice. This type is a Rust struct containing decoded instruction fields, and its in-memory representation differs from the RISC-V machine-code encoding.
3. **Low likelihood of unintended execution:**
   Because adjacent memory is interpreted as the Rust `Instruction` struct rather than raw instruction bytes, most out-of-bounds reads will not correspond to a valid decoded instruction and will result in a panic. However, if the surrounding memory happens to match a valid in-memory representation of the instruction struct, the VM may execute unintended instructions derived from that memory.

In practice, the vulnerability is therefore more likely to result in **unexpected panics or crashes** than reliable unintended instruction execution.

Steps to reproduce

1. Clone the forked repository that contains the fuzzing harness. The repository is in Github. The fuzzing harness can be found in the `dani/rv32im-fuzz` branch.
2. Install dependencies
   a) Install `afl.rs` following the instructions provided in their documentation: `https://rust-fuzz.github.io/book/afl/setup.html`
   b) (Optional) Install a RISC-V C toolchain. On macOS, this can be done by installing the `riscv64-elf-gcc` and `riscv64-elf-binutils` Homebrew packages.
3. Build the harness (note that these commands overwrite the previous build; you must rebuild when comparing different configurations).
   - To build without debug assertions: `cargo build -p fuzzing --release`
   - To build with assertions enabled: `cargo afl build -p fuzzing --release`
4. Run the test case:
   `target/release/rv32im-afl --mode unicorn --test-one < <test-case-file>`

Recommendation Replace the following lines with `self.instructions[word]` so that the bounds check is preserved in release builds.

```
debug_assert!(word < self.instructions.len());
*self.instructions.get_unchecked(word)
```

Test cases

Single instruction A test case containing the single instruction `xori a0,a6,-220` (`0xf2484513`) can trigger this behavior.

With debug assertions enabled:

```
> printf "\x13\x45\x48\xf2" | RUST_LOG=debug target/release/rv32im-afl --mode unicorn
    --test-one
[2026-03-12T10:26:38.694Z INFO ] Debug assertions are enabled!
[2026-03-12T10:26:38.694Z DEBUG] Created vm: Unicorn { uc: 0xa82c00000 }
[2026-03-12T10:26:38.694Z DEBUG] Creating memory map at address 0x0 with 4194304
   bytes
[2026-03-12T10:26:38.694Z DEBUG] Creating memory map at address 0x4194304 with
   1069547520 bytes
[2026-03-12T10:26:38.694Z DEBUG] Wrote program to entrypoint
[2026-03-12T10:26:38.694Z DEBUG] Unicorn VM configured
[2026-03-12T10:26:38.695Z DEBUG] CODE HOOK!! (0x00000000) (4)
[2026-03-12T10:26:38.695Z DEBUG] instr = 0xf2484513
[2026-03-12T10:26:38.695Z DEBUG] Execution completed
[2026-03-12T10:26:38.695Z INFO ] Oracle: Some([4294967076, 0, 0, 0, 0, 0, 0, 0])

thread 'main' (411579) panicked at riscv_transpiler/src/vm/simple_tape.rs:20:13:
assertion failed: word < self.instructions.len()
```

```
note: run with ‘RUST_BACKTRACE=1‘ environment variable to display a backtrace
[2026-03-12T10:26:39.822Z ERROR] Target raised error: assertion failed: word < self.
    instructions.len()
zsh: done       printf "\x13\x45\x48\xf2" |
zsh: abort      RUST_LOG=debug target/release/rv32im-afl --mode unicorn --test-one
```

Without debug assertions enabled:

```
> printf "\x13\x45\x48\xf2" | RUST_LOG=debug target/release/rv32im-afl --mode unicorn
     --test-one
[2026-03-12T10:11:14.479Z DEBUG] Created vm: Unicorn { uc: 0x7b5004000 }
[2026-03-12T10:11:14.479Z DEBUG] Creating memory map at address 0x0 with 4194304
    bytes
[2026-03-12T10:11:14.479Z DEBUG] Creating memory map at address 0x4194304 with
    1069547520 bytes
[2026-03-12T10:11:14.479Z DEBUG] Wrote program to entrypoint
[2026-03-12T10:11:14.479Z DEBUG] Unicorn VM configured
[2026-03-12T10:11:14.480Z DEBUG] CODE HOOK!! (0x00000000) (4)
[2026-03-12T10:11:14.480Z DEBUG] instr = 0xf2484513
[2026-03-12T10:11:14.480Z DEBUG] Execution completed
[2026-03-12T10:11:14.480Z INFO ] Oracle: Some([4294967076, 0, 0, 0, 0, 0, 0, 0])

thread ’main’ (380310) panicked at riscv_transpiler/src/vm/instructions/mod.rs:81:5:
Illegal instruction encounteted at PC = 0x00000004
note: run with ‘RUST_BACKTRACE=1‘ environment variable to display a backtrace
[2026-03-12T10:11:15.007Z INFO ] Target: None
Oracle and result produced different register outputs!
oracle: Some([4294967076, 0, 0, 0, 0, 0, 0, 0])
target: None
zsh: done       printf "\x13\x45\x48\xf2" |
zsh: abort      RUST_LOG=debug target/release/rv32im-afl --mode unicorn --test-one
```

Modified example One of the examples in the repository can be modified to trigger the same behavior.

```
> ls -l examples/basic_fibonacci/app.bin
-rwxr-xr-x  1 foldr  staff  388 Feb 16 17:56 examples/basic_fibonacci/app.bin
> riscv64-elf-objdump -D -b binary -m riscv:rv32 examples/basic_fibonacci/app.bin |
    tail
 15c:   01cd2883            lw  a7,28(s10)
 160:   020d2903            lw  s2,32(s10)
 164:   024d2983            lw  s3,36(s10)
 168:   028d2a03            lw  s4,40(s10)
 16c:   02cd2a83            lw  s5,44(s10)
 170:   030d2b03            lw  s6,48(s10)
 174:   034d2b83            lw  s7,52(s10)
 178:   038d2c03            lw  s8,56(s10)
 17c:   03cd2c83            lw  s9,60(s10)
 180:   0000006f            j   0x180
> head -c 384 examples/basic_fibonacci/app.bin > badfib.bin
> riscv64-elf-objdump -D -b binary -m riscv:rv32 badfib.bin | tail
 158:   018d2803            lw  a6,24(s10)
 15c:   01cd2883            lw  a7,28(s10)
 160:   020d2903            lw  s2,32(s10)
 164:   024d2983            lw  s3,36(s10)
```

```
 168:   028d2a03            lw  s4,40(s10)
 16c:   02cd2a83            lw  s5,44(s10)
 170:   030d2b03            lw  s6,48(s10)
 174:   034d2b83            lw  s7,52(s10)
 178:   038d2c03            lw  s8,56(s10)
 17c:   03cd2c83            lw  s9,60(s10)
```

With debug assertions enabled:

```
> target/release/rv32im-afl --mode unicorn --test-one < badfib.bin
[2026-03-12T10:37:47.826Z INFO ] Debug assertions are enabled!
[2026-03-12T10:37:47.827Z INFO ] Oracle: Some([144, 0, 0, 0, 0, 0, 0, 0])

thread 'main' (437996) panicked at riscv_transpiler/src/vm/simple_tape.rs:20:13:
assertion failed: word < self.instructions.len()
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[2026-03-12T10:37:48.894Z ERROR] Target raised error: assertion failed: word < self.
    instructions.len()
zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < badfib.bin
```

Without debug assertions enabled:

```
> target/release/rv32im-afl --mode unicorn --test-one < badfib.bin
[2026-03-12T10:39:04.395Z INFO ] Oracle: Some([144, 0, 0, 0, 0, 0, 0, 0])

thread 'main' (440991) panicked at riscv_transpiler/src/vm/instructions/mod.rs:81:5:
Illegal instruction encounteted at PC = 0x00000180
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[2026-03-12T10:39:04.950Z INFO ] Target: None
Oracle and result produced different register outputs!
oracle: Some([144, 0, 0, 0, 0, 0, 0, 0])
target: None
zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < badfib.bin
```

Unsafe C code The following C snippet forces execution to jump outside the `.text` section. During execution of `SimpleTape::read_instruction`, the VM attempts to read the element at index `0x1000000` of the `SimpleTape::instructions` slice, resulting in an out-of-bounds access relative to the slice's base address in memory.

```c
#include "rt.h"

#define HEAP_ADDR (void *)0x04000000

Result main() {
  void (*f)() = (void (*)())(HEAP_ADDR);
  (*f)();
  return success(0, 0, 0, 0, 0, 0, 0, 0);
}
```

The C example is available in the forked repository. To test this case, install the RISC-V C toolchain and run the following commands.

```
> make -C c_examples/ badread
riscv64-elf-gcc -march=rv32im -mabi=ilp32 -nostdlib -O0 -nolibc -fno-builtin -
    ffreestanding -T memmap.ld -o badread.elf badread.c rt.c start.s
riscv64-elf-objcopy -O binary badread.elf badread.bin
riscv64-elf-objcopy -O binary --only-section=.text badread.elf badread.text
rm badread.elf
> target/release/rv32im-afl --mode unicorn --test-one < c_examples/badread.text
[2026-03-12T13:14:15.103Z INFO ] Oracle: None
zsh: segmentation fault  target/release/rv32im-afl --mode unicorn --test-one <
    c_examples/badread.text
```

5.1.6 V-AIRBENDER-VUL-006: Out of bounds reads and writes on RAM abstraction

Severity	Warning	**Commit**	5ecb674
Type	Memory safety	**Status**	Acknowledged
Location(s)	`riscv_transpiler/[...]/ram_with_rom_region.rs:69-70, 135-139`		
Confirmed Fix At	Will not fix		

Vulnerability Description The VM implements a RAM abstraction that provides access to guest memory through a contiguous buffer. Access to this buffer relies on unchecked indexing guarded only by a `debug_assert!`. Because `debug_assertions` are disabled by default in release builds, these bounds checks are not enforced in production configurations.

If an address outside the allocated RAM region is used, the RAM abstraction may perform unchecked memory accesses beyond the bounds of the underlying buffer. This can result in **out-of-bounds reads and writes**, allowing memory located after the RAM buffer to be read from or overwritten.

Impact The primary impact of this issue is the ability to perform **out-of-bounds reads and writes** relative to the RAM buffer used by the VM. This is constrained by several factors:

1. **Forward-only memory access:**
   The vulnerability only allows accesses to memory located after the RAM buffer in memory. It does not permit reading or writing addresses located before the buffer.
2. **Partial control over the accessed data:**
   Each RAM cell is represented by a 16-byte structure. However, the attacker can only reliably control or observe a subset of this memory layout, specifically bytes 9 through 12 of each cell. The remaining bytes depend on the internal structure representation and cannot be directly manipulated through the guest interface.
3. **Potential stack corruption:**
   Because the RAM buffer is allocated on the heap, sufficiently large out-of-bounds accesses may reach other memory regions used by the host process, including the stack. In principle, this could allow an attacker to overwrite stack values such as saved frame pointers or return addresses. Such corruption could lead to control-flow hijacking.

While this scenario is difficult to exploit reliably due to the limited control over individual bytes within each memory cell and the dependency on the host process memory layout, the potential impact is significant if such corruption can be achieved.

Recommendation Replace unchecked indexing operations in the RAM abstraction with bounds-checked indexing to ensure that memory accesses remain within the allocated buffer.

Currently, memory accesses rely on unchecked indexing guarded only by `debug_assert!`. Because these checks are not enforced in release builds, accesses using out-of-range addresses may result in out-of-bounds reads or writes.

All accesses to the underlying RAM buffer should instead use bounds-checked indexing (for example, via safe indexing or `get`/`get_mut`). This ensures that invalid addresses are detected and handled appropriately in both debug and release builds, preventing memory accesses outside the allocated RAM region.

Steps to reproduce

1. Clone the forked repository that contains the fuzzing harness. The repository is in Github. The fuzzing harness can be found in the `dani/rv32im-fuzz` branch.
2. Install dependencies
   a) Install `afl.rs` following the instructions provided in their documentation: `https://rust-fuzz.github.io/book/afl/setup.html`
   b) Install a RISC-V C toolchain. On macOS, this can be done by installing the `riscv64-elf-gcc` and `riscv64-elf-binutils` Homebrew packages.
3. Build the harness (note that these commands overwrite the previous build; you must rebuild when comparing different configurations).
   - To build without debug assertions: `cargo build -p fuzzing --release`
   - To build with assertions enabled: `cargo afl build -p fuzzing --release`
4. Run the test case:
   `target/release/rv32im-afl --mode unicorn --test-one < <test-case-file>`

Test cases The following test cases are written in C and are intended to exercise the affected RAM interface through guest programs. Each example includes a short description, the corresponding C source, and the commands used to reproduce the issue with and without debug assertions enabled.

Out of bounds read The guest program performs a sequence of reads from addresses beyond the allocated RAM region. Specifically, it reads from offsets `0, 10, 20, ..., 70` starting from the address immediately following the end of the allocated RAM buffer. This causes the RAM abstraction to access memory outside the bounds of the underlying buffer.

```c
#include "rt.h"

#define RAM_SIZE (1 << 30)
#define HEAP_ADDR (void *)(RAM_SIZE)

Result main() {
  unsigned int *values = (unsigned int *)HEAP_ADDR;

  return success(values[0], values[10], values[20], values[30], values[40],
                 values[50], values[60], values[70]);
}
```

With debug assertions enabled:

```
> make -C c_examples/ badread2
riscv64-elf-gcc -march=rv32im_zicsr -mabi=ilp32 -nostdlib -O0 -nolibc -fno-builtin -
    ffreestanding -T memmap.ld -o badread2.elf badread2.c rt.c start.s uart.c
riscv64-elf-objcopy -O binary badread2.elf badread2.bin
riscv64-elf-objcopy -O binary --only-section=.text badread2.elf badread2.text
rm badread2.elf
> target/release/rv32im-afl --mode unicorn --test-one < c_examples/badread2.text
[2026-03-13T11:55:56.944Z INFO ] Debug assertions are enabled!
[2026-03-13T11:55:56.945Z INFO ] Oracle: None
instructions (254) is at 0x103ed9f30
backing (268435456) is at 0x4d8000000
```

```
11
12 thread 'main' (3492848) panicked at riscv_transpiler/src/vm/ram_with_rom_region.rs
       :122:13:
13 assertion failed: word_idx < self.backing.len()
14 note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
15 [2026-03-13T11:55:58.059Z ERROR] Target raised error: assertion failed: word_idx <
       self.backing.len()
16 zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < c_examples/
       badread2.tex
```

Without debug assertions enabled:

```
1  > make -C c_examples/ badread2
2  riscv64-elf-gcc -march=rv32im_zicsr -mabi=ilp32 -nostdlib -O0 -nolibc -fno-builtin -
       ffreestanding -T memmap.ld -o badread2.elf badread2.c rt.c start.s uart.c
3  riscv64-elf-objcopy -O binary badread2.elf badread2.bin
4  riscv64-elf-objcopy -O binary --only-section=.text badread2.elf badread2.text
5  rm badread2.elf
6  > target/release/rv32im-afl --mode unicorn --test-one < c_examples/badread2.text
7  [2026-03-13T11:58:10.797Z INFO ] Oracle: None
8  instructions (254) is at 0x105509800
9  backing (268435456) is at 0xcb8000000
10 [2026-03-13T11:58:11.317Z INFO ] Target: Some([0, 1073741824, 0, 0, 0, 0, 0, 0])
11 Oracle and result produced different register outputs!
12 oracle: None
13 target: Some([0, 1073741824, 0, 0, 0, 0, 0, 0])
14 zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < c_examples/
       badread2.tex
```

Out of bounds write The guest program writes two strings to memory: `"Valid write!"` is written to valid addresses near the end of the allocated RAM buffer, while `"Hello from the Guest"` is written to addresses beyond the end of the buffer.

In the execution without debug assertions enabled, both strings can be observed in the resulting hexdump. This demonstrates that the valid writes occur within the buffer while the second string is written to memory outside the allocated RAM buffer. The RAM buffer spans the address range `0x4a8000000` to `0x5a7ffffff`.

```
1  #include "rt.h"
2
3  #define RAM_SIZE (1 << 30)
4  #define HEAP_ADDR (void *)(RAM_SIZE)
5
6  Result main() {
7    unsigned int *values = (unsigned int *)HEAP_ADDR;
8
9    // Vali        d wr        ite!
10   // 0x696c6156  0x72772064  0x21657469
11   unsigned int *valid = values - 5;
12   valid[0] = 0x696c6156;
13   valid[1] = 0x72772064;
14   valid[2] = 0x21657469;
15
16   // Hell        o fr        om t        he G        uest
17   // 0x6c6c6548  0x7266206f  0x74206d6f  0x47206568  0x74736575
```

```
  values[1] = 0x6c6c6548;
  values[2] = 0x7266206f;
  values[3] = 0x74206d6f;
  values[4] = 0x47206568;
  values[5] = 0x74736575;

  return success(values[0], values[10], values[20], values[30], values[40],
                 values[50], values[60], values[70]);
}
```

With debug assertions enabled:

```
> make -C c_examples/ badwrite
riscv64-elf-gcc -march=rv32im_zicsr -mabi=ilp32 -nostdlib -O0 -nolibc -fno-builtin -ffreestanding -T memmap.ld -o badwrite.elf badwrite.c rt.c start.s uart.c
riscv64-elf-objcopy -O binary badwrite.elf badwrite.bin
riscv64-elf-objcopy -O binary --only-section=.text badwrite.elf badwrite.text
rm badwrite.elf
> target/release/rv32im-afl --mode unicorn --test-one < c_examples/badwrite.text
[2026-03-13T11:56:22.614Z INFO ] Debug assertions are enabled!
[2026-03-13T11:56:22.615Z INFO ] Oracle: None
instructions (296) is at 0x983444000
backing (268435456) is at 0xc82000000

thread 'main' (3493796) panicked at riscv_transpiler/src/vm/ram_with_rom_region.rs:190:13:
assertion failed: word_idx < self.backing.len()
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[2026-03-13T11:56:23.715Z ERROR] Target raised error: assertion failed: word_idx < self.backing.len()
zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < c_examples/badwrite.tex
```

Without debug assertions enabled:

```
> make -C c_examples/ badwrite
riscv64-elf-gcc -march=rv32im_zicsr -mabi=ilp32 -nostdlib -O0 -nolibc -fno-builtin -ffreestanding -T memmap.ld -o badwrite.elf badwrite.c rt.c start.s uart.c
riscv64-elf-objcopy -O binary badwrite.elf badwrite.bin
riscv64-elf-objcopy -O binary --only-section=.text badwrite.elf badwrite.text
rm badwrite.elf
> target/release/rv32im-afl --mode unicorn --test-one < c_examples/badwrite.text
[2026-03-13T11:57:54.865Z INFO ] Oracle: None
instructions (296) is at 0x1021be700
backing (268435456) is at 0x4a8000000
0x5a7ffffb0: 9a* 00* 00* 00* 00* 00* 00* 00*  56* 61* 6c* 69* 00* 00* 00* 00*  |........Vali....|
0x5a7ffffc0: ae* 00* 00* 00* 00* 00* 00* 00*  64* 20* 77* 72* 00* 00* 00* 00*  |........d wr....|
0x5a7ffffd0: c2* 00* 00* 00* 00* 00* 00* 00*  69* 74* 65* 21* 00* 00* 00* 00*  |........ite!....|
0x5a7ffffe0: 00* 00* 00* 00* 00* 00* 00* 00*  00* 00* 00* 00* 00* 00* 00* 00*  |................|
0x5a7fffff0: 00* 00* 00* 00* 00* 00* 00* 00*  00* 00* 00* 00* 00* 00* 00* 00*  |................|
```

```
0x5a8000000: 2d  01  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |-...............|
0x5a8000010: d6  00  00  00  00  00  00  00   48  65  6c  6c  00  00  00  00
    |........Hell....|
0x5a8000020: ea  00  00  00  00  00  00  00   6f  20  66  72  00  00  00  00
    |........o fr....|
0x5a8000030: fe  00  00  00  00  00  00  00   6f  6d  20  74  00  00  00  00
    |........om t....|
0x5a8000040: 12  01  00  00  00  00  00  00   68  65  20  47  00  00  00  00
    |........he G....|
0x5a8000050: 26  01  00  00  00  00  00  00   75  65  73  74  00  00  00  00
    |&.......uest....|
0x5a8000060: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
0x5a8000070: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
0x5a8000080: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
0x5a8000090: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
0x5a80000a0: 39  01  00  00  00  00  00  00   00  00  00  40  76  00  00  00
    |9..........@v...|
0x5a80000b0: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
0x5a80000c0: 00  00  00  40  7e  00  00  00   00  00  00  00  00  00  00  00  |...@
    ~...........|
0x5a80000d0: 00  00  00  00  00  00  00  00   ec  ff  ff  3f  8a  00  00  00
    |...........?....|
0x5a80000e0: 00  00  00  00  00  00  00  00   00  00  00  00  00  00  00  00
    |................|
[2026-03-13T11:57:55.390Z INFO ] Target: Some([0, 1073741824, 0, 0, 0, 0, 0, 0])
Oracle and result produced different register outputs!
oracle: None
target: Some([0, 1073741824, 0, 0, 0, 0, 0, 0])
zsh: abort      target/release/rv32im-afl --mode unicorn --test-one < c_examples/
    badwrite.tex
```

Developer response The developers acknowledge the out-of-bound writes but will not fix because they should not occur under their trusted bytecode assumption.

6 Fuzz Testing

6.1 Methodology

One of the goals of the security assessment was to fuzz test ZKsync Airbender in order to evaluate the robustness and correctness of selected execution, proving, and proof-validation components under malformed, adversarial, or randomized inputs. In particular, the fuzzing workflows were designed to answer three main questions: whether selected components could be crashed or driven into abnormal states, whether the transpiler/VM behavior could diverge from a reference execution oracle, and whether structured mutations of prover inputs could expose completeness or soundness issues in the proving pipeline.

Executor fuzzing. For the bytecode executor/transpiler workflow, the analysts used an AFL.rs harness to test the Airbender VM on randomized inputs. This campaign was run in two modes. In the first mode, arbitrary inputs were executed directly in the Airbender VM in order to detect crashes, panics, malformed execution states, and other robustness issues. In the second mode, the same inputs were executed both in Airbender and in a Unicorn-based RISC-V reference emulator, and the harness compared the ABI-visible output registers of the two executions. This differential setup was intended to detect semantic divergence from the reference model in addition to crash behavior. This executor based fuzzing found two crashes which resulted in issues V-AIRBENDER-VUL-005 and V-AIRBENDER-VUL-006.

Prover fuzzing. The prover campaign was not based primarily on mutating raw bytecode. Instead, the Veridise analysts built a structure-aware harness that started from valid seed programs and then mutated the prover's internal per-circuit inputs directly. Concretely, the harness first loaded a corpus of seed programs as `.bin`/`.text` pairs, executed each program once through the Airbender VM, and cached the resulting proof inputs for several unrolled circuit families. In the implemented scaffold, this included the main arithmetic, control-flow.

Each cached seed was then expanded into multiple fuzzing cases, one per circuit family. The fuzzer repeatedly selected one such seed case and applied one or more randomized mutations to the corresponding proof input. These mutations were structure-aware: rather than flipping arbitrary bytes, they directly modified semantically meaningful parts of the prover input such as trace rows, row order, row multiplicity, cycle timestamps, read timestamps, the initial PC, decoder entries, decoder-row ordering, memory discriminators, and preprocessed-table rows or cells. This made the campaign substantially better targeted to Airbender's proving logic than generic byte-level mutation would have been.

After mutation, the harness re-ran the relevant circuit-family prover on the tampered input. If proof generation failed or panicked, the case was treated as a robustness signal. If proof generation succeeded, the harness then invoked the corresponding proof-validation routine on the same mutated input and produced proof. This let the campaign distinguish between different classes of behavior: cases where malformed structured inputs caused proof generation to fail, cases where the prover emitted a proof that the validator rejected, and cases where mutated inputs still led to a proof that validated successfully. The latter category was treated as

the most security-relevant, since it may indicate that malformed internal prover inputs can still satisfy the downstream acceptance checks.

To reduce noise, interesting cases were persisted together with the originating seed, targeted circuit family, and list of applied mutations. The harness also supported offline replay-based triage: it reloaded both the original cached seed input and the mutated artifact, replayed each multiple times, and compared compact execution traces across prover stages. This replay step helped distinguish stable, reproducible divergences from unstable outcomes and false positives.

Compliance Testing Complementary to the fuzzing campaign described above, we conducted a series of compliance tests based on a publicly available test suite *. These tests are designed to systematically validate conformance with the RISC-V specification across a range of instruction semantics and edge cases.

The compliance test suite was executed in three distinct configurations:

- Against the Unicorn-based RISC-V reference emulator;
- Against the Airbender virtual machine (VM) in isolation;
- Against the combined Airbender VM and prover system.

The results indicate that all tests passed successfully in the first two configurations. In the third configuration (Airbender VM with prover), all tests passed except for a single failure related to the `udiv` instruction. Specifically, the failing test exposed an issue in the arithmetic circuit when handling division by zero.

This issue was already known to the development team and had been disclosed to the Veridise auditors.

6.2 Fuzzer Execution Summary

Table 6.1: Summary of fuzzing campaign execution statistics

Fuzzing Campaign	Run Time	Approx. Executions	Issues Found
Executor fuzzing	4 days	~1,000,000	2
Prover fuzzing	2 days	~10,000	0

* *https://github.com/riscv/riscv-arch-test*

7 Picus

This section describes how the Veridise analysts used Picus to check the determinism of the ZKsync Airbender circuits.

7.1 Determinism

Given a circuit $C(I, O)$ with input signals I and output signals O, a circuit C is deterministic if and only if:

$$\forall I, O, O'.\ C(I, O) \wedge C(I, O') \rightarrow O \equiv O'$$

In essence, a deterministic circuit encodes a function (or partial function) from I to O. Determinism is an important property that most ZK Circuits should satisfy since most circuits are intended to encode some deterministic computation.

7.2 Methodology

Picus takes as input programs written in the Picus Constraint Language (PCL). Thus, to use Picus for this engagement, the Veridise analysts needed a translation pipeline from Airbender circuits into PCL. Conceptually, this pipeline proceeded in two stages: first, the selected Airbender circuits were translated into LLZK, an intermediate constraint language used in the formal-methods workflow; second, an LLZK-to-PCL lowering pass was used to convert the resulting LLZK programs into PCL for analysis by Picus.

Unrolled Circuit Harnesses. For each selected unrolled opcode-family circuit, the Veridise analysts built a dedicated harness that instantiated the corresponding Airbender circuit in isolation and extracted the resulting constraint system. Each harness was modeled as one local state transition. In practice, this meant treating the starting machine state, decoded instruction information, and values read from registers or memory as inputs, while treating the ending machine state and any write-side effects as outputs. For decoder-oriented harnesses, the instruction word and decoded fields were exposed directly so that the decoder logic could be checked independently of any particular ROM instance.

Extraction and Interface Modeling. After instantiating a harness, the analysts translated the resulting circuit-builder output into LLZK. The builder already separates ordinary polynomial constraints from lookup queries and records auxiliary metadata such as boolean variables, which allowed the extractor to preserve that structure in the LLZK model. To formulate a determinism query, the analysts then determined which AIR columns should be treated as inputs and which should be treated as outputs. The general rule was to model externally supplied values and pre-state values as inputs, and newly computed values and post-state values as outputs (like the `rd` register and values written to memory).

Because Airbender relies on outer memory and permutation arguments for some global consistency properties, the analysts also added explicit local assumptions that are implicit in the full proving system. In particular, the extracted models included one-hot constraints for selector

and control flags, as well as range constraints for values that are intended to be bounded but may enter the local transition as externally supplied inputs. These included halfword-sized values such as memory limbs, immediate limbs, PC limbs, and timestamp limbs, byte-sized values introduced by certain lookup relations, and bounded decoded fields such as register indices and opcode subfields.

Lookups. Lookups were extracted as first-class parts of the LLZK model rather than discarded as opaque implementation details. Since many Airbender transitions depend on lookup outputs for decoder behavior, memory alignment, masking, sign extraction, and related helper logic, preserving those relations was necessary for a meaningful determinism query. The auxiliary metadata recorded by the circuit builder, especially boolean constraints and local range information, was used to supply the side conditions needed to model these lookup-driven computations faithfully.

Opcode Specialization. Many unrolled circuits implement several related opcodes inside one shared circuit family. In those families, opcode-specific logic is guarded by selector columns that are expected to be mutually consistent with the decoder and, in practice, to encode one valid branch at a time. To avoid asking Picus about invalid mixed-selector states, the Veridise analysts partially evaluated the circuit constraints under fixed selector assignments and extracted per-opcode or per-valid-branch variants of the shared circuit. This let the determinism checks focus on one concrete opcode behavior at a time while still analyzing the real production constraints.

Unified Circuit Harnesses. The workflow for the unified reduced-machine circuit was conceptually similar to their unrolled counterparts but required more setup. Unlike the unrolled families, the unified circuit is built as one large circuit intended to contain many instruction families at once rather than as a naturally per-opcode artifact. For the purposes of extraction, the Veridise analysts instantiated harnesses that built the relevant circuit families with fresh optimization contexts and enforced the accumulated constraints after each family was constructed. This differs from the production workflow, where the developers instantiate all families under one shared optimization context and enforce deferred constraints only at the end. The per-family extraction approach produced cleaner LLZK models for analysis while preserving the local semantics of the unified circuit logic under study.

7.3 Results

Table 7.1 summarizes the outcomes of the Picus determinism checks over the main extraction targets. Most of the listed targets were verified deterministic under the modeling assumptions described above. In particular, Picus successfully verified most of the unrolled instruction-family circuits, all of the listed legacy instruction-operation implementations, and the `bigint_with_control` delegation circuit.

Two targets were falsified. First, Picus found a determinism counterexample for unrolled circuit which handles the jumping, branching, and comparison logic (issue V-AIRBENDER-VUL-001). Second, Picus also falsified determinism for `describe_decoder_cycle_from_opcode` which is supposed to decode untrusted bytecode, leading to the creation of issue V-AIRBENDER-VUL-004. We note that after fixing both issues, Picus verified the determinism of these circuits.

Some additional targets remained unresolved. The `apply_mul_div` family and the whole-circuit targets `optimized_base_isa_state_transition` and `apply_reduced_machine_circuit` were not fully discharged within the engagement due to their complexity. The remaining unresolved delegation/precompile circuits, namely `blake2_round_with_extended_control`, `blake2_single_round`, and `keccak_special5`, were especially challenging because of their size and the density of interacting constraints. In principle, verification of these larger components could likely be accelerated by extracting them into smaller logical submodules, but the current extraction infrastructure does not make it easy to partition constraints along those boundaries.

Table 7.1: Picus verification targets.

Target	Verification Unit	Result	After Fix
`apply_add_sub_lui_auipc_mop`	Per-opcode specialization	Verified	N/A
`apply_jump_branch_slt`	Per-opcode specialization	Falsified	Verified
`apply_subword_only_load_store`	Per-opcode specialization	Verified	N/A
`apply_word_only_load_store`	Per-opcode specialization	Verified	N/A
`apply_word_only_load_store`	Per-opcode specialization	Verified	N/A
`apply_load_store`	Per-opcode specialization	Verified	N/A
`apply_mul_div`	Per-opcode specialization	Verified	N/A
`describe_decoder_cycle_from_opcode`	Bytecode decoder	Falsified	Verified
`apply_shift_binop_csrrw`	Per-opcode specialization	Verified	N/A
`AddOp`, `SubOp`	Operation implementation	Verified	N/A
`BinaryOp`	Operation implementation	Verified	N/A
`ShiftOp<SUPPORT_SRA, SUPPORT_ROT>`	Parameterized operation	Verified	N/A
`MulOp<SUPPORT_SIGNED>`	Parameterized operation	Verified	N/A
`DivRemOp<SUPPORT_SIGNED>`	Parameterized operation	Verified	N/A
`LoadOp<...>`	Parameterized operation	Verified	N/A
`ConditionalOp<SUPPORT_SIGNED>`	Parameterized operation	Verified	N/A
`StoreOp<SUPPORT_LESS_THAN_WORD>`	Parameterized operation	Verified	N/A
`LuiOp`, `AuiPc`	Operation implementation	Verified	N/A
`JumpOp`	Operation implementation	Verified	N/A
`MopOp`	Operation implementation	Verified	N/A
`optimized_base_isa_state_transition`	Whole circuit	Unknown	N/A
`apply_reduced_machine_circuit`	Whole circuit	Unknown	N/A
`bigint_with_control`	Delegation circuit	Verified	N/A
`blake2_round_with_extended_control`	Delegation circuit	Unknown	N/A
`blake2_single_round`	Delegation circuit	Unknown	N/A
`keccak_special5`	Delegation circuit	Unknown	N/A

7.3.1 Circuits Requiring Customized Verification

While many circuits were amenable to largely push-button analysis, some required additional analyst guidance before Picus could successfully discharge the determinism proof. In particular, the `mul_div` family, including `apply_mul_div` and `DivRemOp<SUPPORT_SIGNED>`, was ultimately verified, but only after applying verification-oriented rewrites and case decomposition. The main challenge was that the original circuit structure was not immediately solver-friendly. To make the determinism proof tractable, we rewrote certain constraints, such as `IsZero`-style conditions, into more explicit logical assertions. For example, we replaced the constraints with assertions of

the form `(assert (<=> (= out 1) ...))`, which more directly expose the intended semantics to the solver. We also split the verification task into semantically distinct cases, including positive numerators, division-by-zero behavior, and overflow-related corner cases. This case-splitting was necessary to control path complexity and allow Picus to prove determinism for each case individually.

Glossary

AFL.rs A rust interface into AFL-plus-plus (https://github.com/AFLplusplus/AFLplusplus). See https://github.com/rust-fuzz/afl.rs/ to learn more. 29

Fiat-Shamir A well-known method for converting interactive proofs to non-interactive ones. See https://en.wikipedia.org/wiki/Fiat-Shamir_heuristic to learn more. 3

LLZK LLZK is a family of MLIR dialects and compiler optimizations and analysis passes for ZK circuits. See https://github.com/project-llzk/llzk-lib/ . 7

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 35

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1