Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

ZKM

Ziren zkVM

Veridise Inc.
October 9, 2025

► Prepared For:

ZKM
https://www.zkm.io/

► Prepared By:

Kostas Ferles
Shankara Pailoor
Alp Bassa
Daniel Domínguez Álvarez

► Contact Us:

contact@veridise.com

► Version History:

Dec. 15, 2025	V5
Dec. 12, 2025	V4
Dec. 7, 2025	V3
Dec. 3, 2025	V2
Nov. 26, 2025	V1

Contents

1 Executive Summary

From Oct. 9, 2025 to Nov. 12, 2025, ZKM engaged Veridise to conduct a security assessment of their Ziren zkVM. Veridise conducted the assessment over 15 person-weeks, with 3 security analysts reviewing the project over 5 weeks on commit `ad200e3`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review. This included a manual circuit review of the verifier logic and a tool assisted validation of ALU, CPU, control flow, operations, and other miscellaneous circuits using Picus, Veridise's verification tool for zero-knowledge circuits. Furthermore, this review employed fuzzing to systematically test the witness generation logic of the zkVM.

Project Summary. Ziren is a zero-knowledge virtual machine (zkVM), developed by ZKM (formerly known as "zkMIPS"), based on the MIPS instruction-set architecture. It allows the execution of arbitrary MIPS programs and the generation of succinct proofs of correct execution, which are fast to verify.

MIPS Instructions are mapped to algebraic constraints (AIR constraints). The Ziren zkVM has a chip-based modular architecture with separate chips for different kind of operations, e.g., the CPU chip for the core state transitions, the program chip for fetching and decoding of program instructions, the ALU Chip for arithmetic/logic operations, the memory chip for memory operations and ensuring consistency, the global chip for cross-chip coordination and global cross-table look-ups etc. Chips communicate only indirectly through shared trace columns and cross-table lookup/permutation arguments. The AIR system ties their behavior together, so the prover must produce a globally consistent execution.

The MIPS binary (ELF), either written in MIPS assembly, or compiled from a high level language using the respective toolchain, is converted into a representation suitable for proof generation. This involves decoding of the MIPS instructions, identifying syscalls, memory events etc. The virtual machine then executes the program within its modular state machine in collaboration with all chips. This results in the execution trace (forming the witness of the computation). To make proof generation more efficient, traces are split into shards which are combined via a recursive verifier.

Given the constraints and the populated execution trace, STARK proofs can be generated and subsequently wrapped into a SNARK for efficient verification.

Code Assessment. The Ziren zkVM developers provided the source code of the Ziren zkVM contracts for the code review. It is a fork of the SP1 zkVM developed by Succinct Labs (which is based on the RISC-V instruction-set architecture) with modifications by the Ziren zkVM developers. It contains some documentation explaining the general logic and architecture and documentation comments on functions and variables.

The Ziren zkVM development team was responsive in addressing questions and clarifying ambiguities through multiple communication channels. Their prompt and transparent engagement facilitated the Veridise security analysts' understanding of the codebase and its underlying design.

The source code contained a test suite, which the Veridise security analysts noted that shares similar structure to the upstream repository but it is adjusted and extended where necessary for the MIPS architecture. The Veridise security analysts found the test suite helpful in understanding the code and when developing harnesses to fuzz different parts of the zkVM. The test suite covers the normal intended behavior, however, it does not contain "negative" tests to check against potential malicious traces and invalid proofs constructed from them to be used against the verifier. Such test are crucial to catch any major missing constraints. Furthermore, our fuzzing campaigns demonstrated that the zkVM is susceptible to panics during trace generation, which demonstrates another under-tested of the system.

Summary of Issues Detected. The security assessment uncovered 23 issues, 8 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, both the manual review and our verifier uncovered several issues where the zkVM's AIR was missing critical constraints (e.g., V-ZKM-VUL-001, V-ZKM-VUL-004, V-ZKM-VUL-006). The Veridise analysts also identified 10 medium-severity issues, including several cases where our fuzzing campaigns caused the zkVM to panic (e.g., V-ZKM-VUL-011, V-ZKM-VUL-012, V-ZKM-VUL-013), as well as 2 low-severity issues, 2 warnings, and 1 informational finding. The Ziren zkVM developers have addressed 22 of the 23 issues raised by the report.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Ziren zkVM. Note that the following recommendations only apply to the scope of this engagement and not the entire codebase.

Documenting implicit assumptions. Given the inherent complexity that comes with a zkVM implementation, it is essential that all implicit design assumptions are thoroughly documented by the development team. Ensuring that the assumptions are fully documented will help maintain architectural integrity, prevent misaligned implementations, ease on-boarding for external security analysts, and reduce the risk of security or functional issues as the codebase evolves.

Monitoring upstream changes. The Ziren team is encouraged to actively monitor all relevant parts of the upstream repository and development channels for any newly discovered security vulnerabilities, bug fixes, or critical updates that can affect the security of the Ziren zkVM. Staying aligned with upstream changes will help ensure that any relevant patches or mitigations are promptly reviewed and, where applicable, integrated into the ZKM Ziren codebase. This will help maintain security parity, improve stability, and reduce the risk of inherited vulnerabilities over time.

Expand testing. To increase the robustness of the system, it is essential to increase coverage of existing tests and expand testing in under-tested areas of the system. As mentioned above, there are some areas where the project currently lacks testing, e.g., the trace generator and negative tests for the verifier. Finally, these types of tests can also be included in the project's CI/CD via several analysis frameworks, such as fuzzers, static analyzers, and verifiers, to ensure that subtle bugs are being caught early in the development cycle and minimize the risk of shipping the zkVM with critical vulnerabilities.

Disclaimer. Given the complexity of the Ziren zkVM, the size of the code base, and the scope of this security review, the Veridise team cannot make guarantees about the absence of high/critical issues in the protocol beyond the 8 identified. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Ziren zkVM	`ad200e3`	Rust	Plonky3

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 9–Nov. 12, 2025	Manual & Tools	3	15 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	7	7	7
High-Severity Issues	1	1	1
Medium-Severity Issues	10	10	10
Low-Severity Issues	2	2	1
Warning-Severity Issues	2	2	2
Informational-Severity Issues	1	1	1
TOTAL	23	22	22

Table 2.4: Category Breakdown.

Name	Number
Denial of Service	9
Data Validation	7
Logic Error	3
Underconstrained Circuit	2
Maintainability	2

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Ziren zkVM's source code. During the assessment, the security analysts aimed to answer questions such as:

- Do the Ziren zkVM circuits correctly encode the MIPS semantics?
- Are there any underconstrained vulnerabilities in the Ziren zkVM circuits?
- Does the zkVM correctly implement the MIPS ISA using the Koalabear prime?
- Are various memory and register operations constrained properly?
- Is offline-memory consistency checking based on multiset-hashing realized correctly?
- Has foreign field operation emulation been constrained properly?
- Does the zkVM perform adequate validation checks during trace generation?
- Are the different components of Ziren zkVM thoroughly tested and documented?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Formal Verification*. To identify underconstrained vulnerabilities in the ZKM Ziren circuits, the security analysts utilized a new, proprietary version of Picus. Picus can prove (or find counterexamples) that a circuit is deterministic using a combination of static analysis and smt solvers.
- *Fuzzing/Property-based Testing*. Since the Ziren zkVM generates traces of MIPS executions, the Veridise analysts utilized fuzz testing to ensure that the prover both conforms to the MIPS specification (ELF binary parsing, instruction decoding, and instruction execution) and does not contain any common vulnerabilities like buffer overflows, memory leaks, double-frees, etc.
- *Manual review*. All other parts of the scope's engagement not mentioned above were reviewed manually by the Veridise analysts.

Scope. The scope of this security assessment is limited to the following folders of the source code provided by the Ziren zkVM developers.

For the manual code review (excluding any logic related to trace generation) :

- `crates/core/machine/src/operations/`
  - `global_accumulation.rs`
  - `global_lookup.rs`
  - `mod.rs`
  - `field/`
  - `field_den.rs`
  - `field_inner_product.rs`
  - `field_op.rs`

    - `field_sqrt.rs`
    - `mod.rs`
    - `params.rs`
    - `range.rs`
    - `util.rs`
    - `util_air.rs`
- `crates/core/machine/src/air/`
- `crates/core/machine/src/bytes/`
- `crates/core/machine/src/syscall`
- `crates/core/machine/src/global/`
- `crates/core/machine/src/memory/`

For the use of the tool Picus for extraction and verification of determinism:

- `crates/core/machine/src/`
    - `alu/`
    - `cpu/`
    - `control_flow/`
    - `misc/`
    - `operations/`
      `EXCLUDING:`
        - `field/`
        - `global_accumulation.rs`
        - `global_lookup.rs`
        - `mod.rs`

The fuzzing campaigns covered the trace generation logic of all files covered by the both the manual review and our verifier.

Methodology. Veridise security analysts inspected the provided tests and read the Ziren zkVM documentation. They then began a manual review and a formal verification of relevant parts of the code. The witness generation lifecycle was extensively stress-tested for denial of service or logical errors through developed programs.

During the security assessment, the Veridise security analysts regularly communicated with the Ziren zkVM developers to ask questions about the code.

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 any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Ziren zkVM.

- The Ziren zkVM developers are aware of any implicit assumptions of the original code base.
- Code outside the scope of this review does not violate any assumptions required by the files included in the security review.
- Out-of-scope components are assumed to be correct.

5 Picus

5.1 Overview

This section describes how the Veridise analysts used Picus to check the determinism of the Ziren Chips.

5.2 Determinism

Given a circuit $C(I, O)$ with inputs 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.

5.3 Methodology

Picus takes as input a circuit written in its custom circuit language called PCL. Since the Ziren circuits were written in Plonky3, the Veridise analysts wrote a transpiler to convert circuits written in Plonky3 to PCL. They did so by first converting the circuit to the LLZK IR, applying constraint simplification and rewrite passes before transforming it into PCL. One key challenge in performing this conversion was identifying the input and output signals for each circuit since the Chip consisted solely of constraints and lookups. The Veridise addressed this challenge by utilizing the interactions produced in each chip. In particular, the order of the values passed into the receive instruction interaction were used to determine which columns should be considered inputs and outputs.

5.4 Results Summary

We used Picus to check the determinism of 10 chips – namely, all `alu` and `control_flow` chips. Picus proved the determinism of 8/10 chips except for the `CloClz` and `Branch` chips which counts the leading ones (or zeros) of a 32-bit number and branching instructions. For those chips, Picus found 3 underconstrained bugs (V-ZKM-VUL-006, V-ZKM-VUL-007, V-ZKM-VUL-018). After fixing those, Picus proved the determinism of the remaining chips.

6 Fuzz Testing

6.1 Methodology

One of the goals of the security assessment was to fuzz test Ziren zkVM to evaluate its resilience against arbitrary inputs and to evaluate the correctness of its components.

The Veridise security analysts targeted two crates in Ziren zkVM: `zkm-core-executor` and `zkm-core-machine` to test the core executor and trace generation logic. For each crate, the analysts created small testing harnesses that exercised the target code paths. These harnesses were then fuzzed using AFL++.

6.2 Properties Fuzzed

For `zkm-core-executor`, the Veridise security analysts created a test harness that consumes arbitrary binary data interpreted as MIPS machine code and feeds it to the virtual machine. The harness crashes if the virtual machine panics during execution or ends in an error condition.

For `zkm-core-machine`, the Veridise security analysts created a set of test harnesses targeting low level operations of the ZK circuit. The operations under test share a common structure; each operation implements a method that fills the columns of its part of the circuit (*populate* step) and a method that evaluates the constraints related to the columns (*eval* step). The tests generate plausible inputs from arbitrary data and check two properties of each operation:

- Is the result of the *populate* step equal to the equivalent off-circuit operation?
- Do the constraints validate in the *eval* step after completing the *populate* step?

If any property does not hold, the harness crashes, signaling the fuzzer that a counterexample was found. In addition, any unexpected panic would also be considered by the fuzzer to be a crash.

6.3 Fuzzing campaign

The harness for `zkm-core-executor` was fuzzed for a total of 72 compute-hours, 24 in which the target had AddressSanitizer enabled. The fuzzer found 610 crashes. After triage, the set was consolidated into 19 unique crashes. None of the crashes were triggered by AddressSanitizer. The Veridise team identified 9 bugs after manually analyzing the crashes.

The harness for `zkm-core-machine` was fuzzed for a total of 48 compute-hours and no crashes were found.

The Veridise team devoted a total of 120 compute-hours to fuzzing Ziren zkVM, identifying a total of 9 bugs.

7 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 7.1 summarizes the issues discovered:

Table 7.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ZKM-VUL-001	Syscalls can be manipulated by malicious . . .	Critical	Fixed
V-ZKM-VUL-002	Malicious provers can inject syscall events	Critical	Fixed
V-ZKM-VUL-003	Malicious prover can manipulate the . . .	Critical	Fixed
V-ZKM-VUL-004	Result of LL instruction is under-constrained	Critical	Fixed
V-ZKM-VUL-005	Memory instructions can access the . . .	Critical	Fixed
V-ZKM-VUL-006	Unsound Zero Check on 'bb' word in CloClz	Critical	Fixed
V-ZKM-VUL-007	Unsound equality check on 'sr1' in CloClz . . .	Critical	Fixed
V-ZKM-VUL-008	Function eval_syscall can contribute terms . . .	High	Fixed
V-ZKM-VUL-009	Panic prone code in syscalls/commit.rs	Medium	Fixed
V-ZKM-VUL-010	Overflowing arithmetic	Medium	Fixed
V-ZKM-VUL-011	Panic in write syscall	Medium	Fixed
V-ZKM-VUL-012	Panic prone code in hook.rs	Medium	Fixed
V-ZKM-VUL-013	Panic prone code in memory.rs	Medium	Fixed
V-ZKM-VUL-014	Panic prone code in executor.rs	Medium	Fixed
V-ZKM-VUL-015	Panic prone code in events/precompiles/ec.rs	Medium	Fixed
V-ZKM-VUL-016	Panic prone code in syscalls/verify.rs	Medium	Fixed
V-ZKM-VUL-017	Panic prone code in hint.rs	Medium	Fixed
V-ZKM-VUL-018	Missing Data Validation on next_next_pc	Medium	Fixed
V-ZKM-VUL-019	Issue with converting BigUint with large . . .	Low	Acknowledged
V-ZKM-VUL-020	Inconsistent constraints on syscall_id in . . .	Low	Fixed
V-ZKM-VUL-021	Maintainability concerns	Warning	Fixed
V-ZKM-VUL-022	Constraint on send_to_table has multiple . . .	Warning	Fixed
V-ZKM-VUL-023	Unnecessary constraints	Info	Fixed

7.1 Detailed Description of Issues

7.1.1 V-ZKM-VUL-001: Syscalls can be manipulated by malicious provers

Severity	Critical	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	crates/core/machine/src/syscall/chip.rs:205-208		
Confirmed Fix At	8a14e38		

Function `eval` in `crates/core/machine/src/syscall/chip.rs` "constrains" `local.is_real` with a reflexive equality (`is_real^3 == is_real^3`) that does not restrict the witness. The same row passes `local.is_real` directly to `builder.receive_syscall` and to the global lookup weight and send/receive flags in `builder.send` (see snippet below), so a prover may supply any field element as the multiplicity and send/receive flags for a syscall row.

```rust
builder.assert_eq(
    local.is_real * local.is_real * local.is_real,
    local.is_real * local.is_real * local.is_real,
);

match self.shard_kind {
    SyscallShardKind::Core => {
        builder.receive_syscall(
            local.shard,
            local.clk,
            local.syscall_id,
            local.arg1,
            local.arg2,
            local.is_real,
            LookupScope::Local,
        );

        // Send the lookup to the global table.
        builder.send(
            AirLookup::new(
                vec![
                    local.shard.into(),
                    local.clk.into(),
                    local.syscall_id.into(),
                    local.arg1.into(),
                    local.arg2.into(),
                    AB::Expr::ZERO,
                    AB::Expr::ZERO,
                    local.is_real.into() * AB::Expr::ONE,
                    local.is_real.into() * AB::Expr::ZERO,
                    AB::Expr::from_canonical_u8(LookupKind::Syscall as u8),
                ],
                local.is_real.into(),
                LookupKind::Global,
            ),
            LookupScope::Local,
        );
    }
    SyscallShardKind::Precompile => {
        builder.send_syscall(
            local.shard,
            local.clk,
            local.syscall_id,
            local.arg1,
            local.arg2,
            local.is_real,
            LookupScope::Local,
        );

        // Send the lookup to the global table.
        builder.send(
            AirLookup::new(
                vec![
                    local.shard.into(),
                    local.clk.into(),
                    local.syscall_id.into(),
                    local.arg1.into(),
                    local.arg2.into(),
                    AB::Expr::ZERO,
                    AB::Expr::ZERO,
                    local.is_real.into() * AB::Expr::ZERO,
                    local.is_real.into() * AB::Expr::ONE,
                    AB::Expr::from_canonical_u8(LookupKind::Syscall as u8),
                ],
                local.is_real.into(),
                LookupKind::Global,
            ),
            LookupScope::Local,
        );
    }
}
```

Snippet 7.1: Snippet from `fn eval`

Impact Weighted syscalls might let an attacker forge, cancel, or scale host interactions while keeping the lookup multiset balanced, breaking the VM-host integrity guarantees (e.g., faking a host response or suppressing an expected syscall outcome).

Recommendation Constrain `is_real` to be boolean so each syscall row contributes either zero or one unit to the lookup.

Developer Response The developers fixed the issue in commit `8a14e38`.

7.1.2 V-ZKM-VUL-002: Malicious provers can inject syscall events

Severity	Critical	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/core/machine/src/syscall/[...]/air.rs:121-122`		
Confirmed Fix At	05e8ad8		

Function `eval_syscall` in `crates/core/machine/src/syscall/instructions/air.rs:117-124` computes `send_to_table = syscall_code[2] + local.is_sys_linux` and only checks that this sum is zero on padding rows. Since `syscall_code[2]` is guaranteed to only be a byte (by the CPU chip) and `local.is_sys_linux` is not range-constrained anywhere, a prover can pick arbitrary field elements for both variables while keeping the padding constraint satisfied (e.g., set `syscall_code[2] = -local.is_sys_linux`). The unchecked value then feeds the multiplicity for `builder.send_syscall` into the precompile table.

```
// SAFETY: Assert that for non real row, the send_to_table value is 0 so that the ‘
    send_syscall‘
// interaction is not activated.
builder.when(AB::Expr::ONE - local.is_real).assert_zero(send_to_table.clone());

// Compute whether this syscall is SYS_NOP.
let is_sys_nop = {
    IsZeroOperation::<AB::F>::eval(
        builder,
        local.syscall_id - AB::Expr::from_canonical_u32(SyscallCode::SYS_NOP.
    syscall_id()),
        local.is_sys_nop,
        local.is_real.into(),
    );
    local.is_sys_nop.result
};

builder.send_syscall(
    local.shard,
    local.clk,
    syscall_id.clone(),
    local.op_b_value.reduce::<AB>(),
    local.op_c_value.reduce::<AB>(),
    send_to_table - is_sys_nop,
    LookupScope::Local,
);
```

Snippet 7.2: Snippet from `fn eval_syscall`

Impact An attacker can inject weighted syscall events (by choosing `send_to_table` != {0,1}) while the AIR still verifies, breaking the integrity of the syscall dispatch argument.

Recommendation Constrain `syscall_code[2]` and `local.is_sys_linux` to be booleans.

Developer Response The developers fixed this issue in commit `05e8ad8`.

7.1.3 V-ZKM-VUL-003: Malicious prover can manipulate the memory argument

Severity	Critical	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/core/machine/src/memory/local.rs:195-198`		
Confirmed Fix At	1d4b21a		

Function `eval` at `crates/core/machine/src/memory/local.rs:206` only asserts `local.is_real^3 == local.is_real^3`, leaving the lookup weight witness-controlled. The same row later uses `local.is_real` as the multiplicity for both local and global memory lookups (see snippet below). As a result, a prover may submit fractional or negative weights that keep the multiset hash consistent while forging, canceling, or scaling memory accesses. For example, when `is_real = 2` it doubles the contribution of a fabricated row, letting the prover recreate an earlier state without producing the matching write.

```rust
for local in local.memory_local_entries.iter() {
    builder.assert_eq(
        local.is_real * local.is_real * local.is_real,
        local.is_real * local.is_real * local.is_real,
    );

    let mut values =
        vec![local.initial_shard.into(), local.initial_clk.into(), local.addr.into()
    ];
    values.extend(local.initial_value.map(Into::into));
    builder.receive(
        AirLookup::new(values.clone(), local.is_real.into(), LookupKind::Memory),
        LookupScope::Local,
    );

    // Send the lookup to the global table.
    builder.send(
        AirLookup::new(
            vec![
                local.initial_shard.into(),
                local.initial_clk.into(),
                local.addr.into(),
                local.initial_value[0].into(),
                local.initial_value[1].into(),
                local.initial_value[2].into(),
                local.initial_value[3].into(),
                local.is_real.into() * AB::Expr::ZERO,
                local.is_real.into() * AB::Expr::ONE,
                AB::Expr::from_canonical_u8(LookupKind::Memory as u8),
            ],
            local.is_real.into(),
            LookupKind::Global,
        ),
        LookupScope::Local,
    );

    // Send the lookup to the global table.
    builder.send(
        AirLookup::new(
            vec![
                local.final_shard.into(),
                local.final_clk.into(),
                local.addr.into(),
                local.final_value[0].into(),
                local.final_value[1].into(),
                local.final_value[2].into(),
                local.final_value[3].into(),
                local.is_real.into() * AB::Expr::ONE,
                local.is_real.into() * AB::Expr::ZERO,
                AB::Expr::from_canonical_u8(LookupKind::Memory as u8),
            ],
            local.is_real.into(),
            LookupKind::Global,
        ),
        LookupScope::Local,
    );

    let mut values =
        vec![local.final_shard.into(), local.final_clk.into(), local.addr.into()];
    values.extend(local.final_value.map(Into::into));
    builder.send(
        AirLookup::new(values.clone(), local.is_real.into(), LookupKind::Memory),
        LookupScope::Local,
    );
}
```

Snippet 7.3: Snippet from `fn eval`

Impact Forged multiplicities let an adversary rebalance the memory accumulator, replay stale values, or erase writes, undermining memory soundness for every layer that relies on this lookup argument.

Recommendation Constraint `local.is_real` to be boolean.

Developer Response The developers fixed the issue in commit `1d4b21a`.

7.1.4 V-ZKM-VUL-004: Result of LL instruction is under-constrained

Severity	Critical	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/core/machine/src/memory/[...]/air.rs:470-473`		
Confirmed Fix At	https://github.com/ProjectZKM/Ziren/pull/327		

Function `eval_unsigned_mem_value` in `crates/core/machine/src/memory/instructions/air.rs:381-444` does not constraint `unsigned_mem_val` when `local.is_ll = 1`, so the witness may choose any value for `unsigned_mem_val` for LL instructions. Furthermore, `eval_memory_load` enforces `unsigned_mem_val == op_a_value` whenever `mem_value_is_pos` is 1, which includes the case when `local.is_ll = 1` (see snippet below). For an LL row this means the prover can set `unsigned_mem_val` and `op_a_value` to an arbitrary field element while the circuit never links them back to the real memory word. The load-linked semantics are therefore unprotected: LL can return any value without matching the underlying memory access.

```rust
pub(crate) fn eval_memory_load<AB: ZKMAirBuilder>(
    &self,
    builder: &mut AB,
    local: &MemoryInstructionsColumns<AB::Var>,
) {
    // Assert that correct value of `mem_value_is_pos`.
  // SAFETY: If it's a store instruction or a padding row, `mem_value_is_pos = 0`.
  // If it's an unsigned instruction (LBU, LHU, LW), then `mem_value_is_pos = 1`.
  // If it's signed instruction (LB, LH), then `most_sig_bit` will be constrained
    correctly, and same for `mem_value_is_pos`.
  let mem_value_is_pos = (local.is_lb + local.is_lh) * (AB::Expr::ONE - local.
    most_sig_bit)
      + local.is_lbu
      + local.is_lhu
      + local.is_lw
      + local.is_ll;
  builder.assert_eq(local.mem_value_is_pos, mem_value_is_pos);

    // [VERIDISE]: ... code omitted for brevity

/// This function is used to evaluate the unsigned memory value for the load memory
/// instructions.
pub(crate) fn eval_unsigned_mem_value<AB: ZKMAirBuilder>(
    &self,
    builder: &mut AB,
    local: &MemoryInstructionsColumns<AB::Var>,
) {

    // [VERIDISE]: ... code omitted for brevity

    // Compute the expected stored value for a LL instruction.
    builder.when(local.is_ll).assert_word_eq(a_val.map(|x| x.into()), mem_val);
    // Ensure that the offset is 0.
    builder.when(local.is_ll).assert_one(offset_is_zero.clone());
```

Snippet 7.4: Snippet from `crates/core/machine/src/memory/instructions/air.rs:381-444`

Impact By forging the LL result, an attacker can load arbitrary register values independent of memory state, allowing bogus computations and breaking the soundness of load-linked/store-conditional protocols that rely on accurate reads.

Recommendation When `local.is_ll` holds, constrain `unsigned_mem_val` directly to the accessed memory word.

Developer Response The developers have fixed the issue in this pull request.

7.1.5 V-ZKM-VUL-005: Memory instructions can access the register region of the memory.

Severity	Critical	**Commit**	ad200e3
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/core/machine/src/memory/[...]/air.rs:130-199`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/337`		

The zkVM uses its memory area to store both the memory contents of the program trace and the values of its registers. More specifically, the VM designates the lower 36 memory addresses to store the values of the VM's registers. However, the function `eval_memory_address_and_access` (in `crates/core/machine/src/memory/instructions/air.rs`) which evaluates the memory address and the actual access for memory instructions does not prevent memory instructions to access the designated register area.

Impact This allows attackers to craft malicious traces that can overwrite or read contents of registers directly, since `eval_memory_address_and_access` does not impose this artificial barrier between memory and registers.

Recommendation Add constraints in `eval_memory_address_and_access` to enforce that the memory address is outside the register region.

Developer Response The developers fixed the issue in this PR.

7.1.6 V-ZKM-VUL-006: Unsound Zero Check on 'bb' word in CloClz

Severity	Critical	**Commit**	ad200e3
Type	Underconstrained Circuit	**Status**	Fixed
Location(s)	`crates/core/machine/src/alu/clo_clz/mod.rs:239`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/328`, 5900b25		

Description The `CloClz` chip takes as input a word `b` and returns `a` which denotes the number of leading ones or zeroes of a 32 bit word (depending on which selector is applied). The chip handles both cases similarly: if the operation is `CLZ` then `b` is converted to `!b` (denoted by `bb` in the circuit) and then the following constraints are asserted:

1. 1. `bb >> (31 - a[0]) == 1`
2. `a[1] = a[2] = a[3] = 0`

For the `CLO` opcode, `bb = b`. The above constraints only work if `b != 0` so the circuit handles that case separately. In particular, to distinguish the cases, the circuit uses the following constraints:

```
    {
                builder.assert_bool(local.is_bb_zero);
        builder.when(local.is_bb_zero).assert_zero(local.bb.reduce::<AB>());
        builder.when(local.is_bb_zero).assert_eq(local.a[0], AB::Expr::
    from_canonical_u32(32));
    }
```

The problem is that `local.bb.reduce::<AB>()` computes $\sum_{i=0}^{3} 256^i bb_i$ which can overflow the prime field which means a non-zero word can reduce to 0. In that case, the prover has a choice of setting `local.is_bb_zero` to 1 to force the return value to be 32 when it isn't.

Impact A malicious prover has a free choice of setting `is_bb_zero` to be 1 or 0 when `bb` represents a multiple of the prime field. The number of such values is small (2) given that the field is KoalaBear and the outer ring is the set of 32-bit integers. In either of those case, the prover can make the instruction return `32` or the correct number of leading ones or zeros. The main concern with this bug is if an attacker can control the value `b` that is passed into this chip. For example, if the previous instruction added two values which were inputs to the zkVM application, then the attacker could trigger the underconstrained behavior.

Recommendation The simplest solution would be to add the following check:

```
        builder.when(local.is_bb_zero).assert_zero(local.bb[3]);
```

Essentially, if `bb_zero` is zero then all the bytes should be zero. By explicitly checking the last byte is zero, the sum cannot overflow making the check sound.

Developer Response Developers have fixed the issue in commit `5900b25`.

7.1.7 V-ZKM-VUL-007: Unsound equality check on ‘sr1‘ in CloClz Chip

Severity	Critical	**Commit**	ad200e3
Type	Underconstrained Circuit	**Status**	Fixed
Location(s)	`crates/core/machine/src/alu/clo_clz/mod.rs:262-263`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/328`, 5900b25		

The CloClz chip counts the number of leading ones and zeros of a 32-bit word. In the circuit, the input word is represented by an length-4 array of field elements `b` each representing a byte of the word. The circuit exploits the fact that the number of leading zeros, denoted by `a[0]`, must satisfy the relation: `b >> a[0] == 1` for any non-zero `b`. This is shown in the following constraints:

```
{
    // Use the SRL table to compute bb >> (31 - result).
    builder.send_alu(
        Opcode::SRL.as_field::<AB::F>(),
        local.sr1,
        local.bb,
        Word([
            AB::Expr::from_canonical_u32(31) - local.a[0],
            zero.clone(),
            zero.clone(),
            zero.clone(),
        ]),
        one.clone() - local.is_bb_zero,
    );
}

// if bb!=0, check sr1 == 1
{
    builder.when_not(local.is_bb_zero).assert_one(local.sr1.reduce::<AB>());
}
```

The issue is that `local.sr1.reduce::<AB>()` performs the sum $\sum_{i=0}^{3} 256^i * sr1_i$, which can overflow the order of the field. This means that for any input bb , the attacker can select any $a[0]$ for which `bb >> a[0]` is congruent to $1 \mod p$ and use that value for the result.

Impact This attack will not work for all inputs, but only inputs of the form $0 \leq 2^k * a < 2^{32}$ where $a \equiv 1 \mod p$. There are only 3 such inputs given that the prime is KoalaBear and the outer ring is the 32-bit integers. Although the number of such inputs is small, it is quite plausible that programs will trigger these scenarios in practice, especially if an attacker can craft inputs which lead to this scenario. For example, if the prior instruction was an addition of two values which were inputs to the zkVM program then an attacker could specifically set those values to trigger this scenario.

Recommendation Our recommendation is to add a constraint that the most significant byte of `sr1` is 0 when `is_bb_zero` is 0.

7.1.8 V-ZKM-VUL-008: Function eval_syscall can contribute terms on the lookup argument on padding rows.

Severity	High	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/core/machine/src/syscall/[...]/air.rs:124-152`		
Confirmed Fix At	3a2cc01		

Function `eval_syscall` uses two expressions, namely `is_sys_nop` and `send_to_table`, to control what values to send to the builder's function `send_syscall` (which contributes to the lookup). However, expression `is_sys_nop` is unconstrained on padding rows (see comments marked with `[VERIDISE]` in the below snippet).

```rust
// Compute whether this syscall is SYS_NOP.
let is_sys_nop = {
    // [VERIDISE]: constraints from IsZeroOperation::<AB::F>::eval are only emitted
    IsZeroOperation::<AB::F>::eval(
        builder,
        local.syscall_id - AB::Expr::from_canonical_u32(SyscallCode::SYS_NOP.
    syscall_id()),
        local.is_sys_nop,
        local.is_real.into(),
    );
    // [VERIDISE]: local.is_sys_nop.result is unconstrained on padding rows
    local.is_sys_nop.result
};

builder.send_syscall(
    local.shard,
    local.clk,
    syscall_id.clone(),
    local.op_b_value.reduce::<AB>(),
    local.op_c_value.reduce::<AB>(),
    // [VERIDISE]: This can be non-zero on padding rows
    send_to_table - is_sys_nop,
    LookupScope::Local,
);

builder.send_syscall(
    local.shard,
    local.clk,
    local.syscall_id,
    local.op_b_value.reduce::<AB>(),
    local.op_c_value.reduce::<AB>(),
        // [VERIDISE]: This can be non-zero on padding rows
    is_sys_nop,
    LookupScope::Local,
);
```

Snippet 7.5: Snippet from `eval_syscall`

Impact This allows attackers to manipulate the lookup argument on padding rows.

Recommendation Either protect each use of `is_sys_nop` by multiplying it with `local.is_real` or enforce `is_sys_nop` to be zero on padding rows (same as `send_to_table`).

Developer Response The developers fixed this issue in commit `3a2cc01`.

7.1.9 V-ZKM-VUL-009: Panic prone code in syscalls/commit.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/syscalls/commit.rs:16`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, 443b777		

Line 16 in `commit.rs` performs an index operation that may panic if the index is out of bounds. This panic was found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

7.1.10 V-ZKM-VUL-010: Overflowing arithmetic

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/` ▶ `executor.rs:1346, 1366, 1386, 1441, 1455`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, 2275b1b		

Several locations use unchecked arithmetic which could lead to crashes or unexpected behavior due to arithmetic overflows. These arithmetic overflows can be triggered from user supplied programs, that could contain logic errors or malicious instructions that cause the overflows. These overflows were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM. Below is a list of the locations where the fuzzer found overflows, however it probably does not contain all the operations that could have this issue.

Addition overflows

- Lines 29 and 36 of `sysmmap.rs`
- Lines 1346 and 1386 of `executor.rs`

Subtraction overflows

- Lines 1366 and 1455 of `executor.rs`

Left-shift overflows

- Lines 1441 and 1455 of `executor.rs`

7.1.11 V-ZKM-VUL-011: Panic in write syscall

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/syscalls/write.rs:35, 47`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, 29d5064		

When writing text to the standard output or standard error via the `write` syscall the supplied data is encoded using UTF-8, panicking if the conversion failed. Users can crash the VM by supplying bytes not encoded in that standard either by mistake or intentionally.

Additionally, this makes the VM not compatible with other encodings, which, for example, legacy software could still be using despite UTF-8's widespread adoption.

Recommendation: Handle UTF-8 decoding errors in a more resilient manner, such as printing when encountering invalid characters.

7.1.12 V-ZKM-VUL-012: Panic prone code in hook.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/[...]/hook.rs:120, 239, 282, 486, 514, 526`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/375`, d9ce8e0		

Several locations in `hook.rs` perform operations that may panic. The panics in this issue were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

- `hook.rs:282`, `hook.rs:239`, `hook.rs:120`, `hook.rs:486`, `hook.rs:526`: Panics if the provided inputs are smaller than expected. The panics in lines 120, 486, and 526 are not documented in their functions' doc string.
- `hook.rs:514`: Panics if the provided input is a non-quadratic residue.

7.1.13 V-ZKM-VUL-013: Panic prone code in memory.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/memory.rs:62, 113`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, ff6c568		

Several locations in `memory.rs` perform operations that may panic. The panics in this issue were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

- `memory.rs:62`, `memory:113`: Panics when obtaining the page index from a memory address and that memory is unpaged.

7.1.14 V-ZKM-VUL-014: Panic prone code in executor.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/executor.rs:1469, 1540-1544`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, f688c0c		

Several locations in `executor.rs` perform operations that may panic. The panics in this issue were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

- `executor.rs:1540-1544`: If the divisor operand is 0 then the VM crashes with a divide-by-zero error.
- `executor.rs:1469`: The handler of the `teq` instruction panics if the two operands are equal. This is close to the intended behavior of the instruction, that traps the CPU is the two values are equal. However, panicking here brings the whole VM down.

7.1.15 V-ZKM-VUL-015: Panic prone code in events/precompiles/ec.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/` ▶ `core/executor/src/events/precompiles/ec.rs:208`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/375/`, 1db0146		

Line 208 in `events/precompiles/ec.rs` performs an operation that may panic when creating a decompress event, which is caused by unwrapping an empty `CtOption` in `secp256k1.rs:97`. This panic was found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

7.1.16 V-ZKM-VUL-016: Panic prone code in syscalls/verify.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/[...]/verify.rs:21, 23, 31, 43`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366/`, e5f8456		

Several locations in `verify.rs` perform operations that may panic. The panics in this issue were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

- `syscalls/verify.rs:21`, `syscalls/verify.rs:23`, `syscalls/verify.rs:31`, `syscalls/verify.rs:43`: Several panics that can trigger from user provided inputs.

7.1.17 V-ZKM-VUL-017: Panic prone code in hint.rs

Severity	Medium	**Commit**	ad200e3
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/core/executor/src/syscalls/hint.rs:29, 37-39, 58`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/366`, 241cb02		

Several locations in `hint.rs` perform operations that may panic. The panics in this issue were found with a fuzzer that executed random MIPS instructions. A similar scenario can happen during normal execution if the provided ELF program contains logic errors or malicious instructions that result in crashing the VM.

- `syscalls/hint.rs:29`, `syscalls/hint.rs:37-39`: Several panics that can trigger from user provided inputs.
- `syscalls/hint.rs:58`: Will panic if the programs calls this syscall twice.

7.1.18 V-ZKM-VUL-018: Missing Data Validation on next_next_pc

Severity	Medium	**Commit**	ad200e3
Type	Data Validation	**Status**	Fixed
Location(s)	crates/core/machine/src/control_flow/[...]/air.rs:120-123		
Confirmed Fix At	https://github.com/ProjectZKM/Ziren/pull/328, 5900b25		

The Branch Chip validates all branch instructions in the zkVM. In particular, it makes sure that the `next_pc` and `next_next_pc` values are properly updated. While `next_pc` is properly updated, Picus finds that the columns of `next_next_pc` are underconstrained in the case where branching does not occur. In particular the following constraints relate how `next_next_pc` relates to `next_pc` when branching does not occur.

```
// Range check local.next_pc, local.next_next_pc and local.target_pc, .
// SAFETY: ‘is_real‘ is already checked to be boolean.
// The ‘KoalaBearWordRangeChecker‘ assumes that the value is checked to be a valid
    word.
// This is done when the word form is relevant, i.e. when ‘pc‘ and ‘next_pc‘ are sent
     to the ADD ALU table.
// The ADD ALU table checks the inputs are valid words, when it invokes ‘AddOperation
    ‘.
KoalaBearWordRangeChecker::<AB::F>::range_check(
    builder,
    local.next_pc,
    local.next_pc_range_checker,
    is_real.clone(),
);

KoalaBearWordRangeChecker::<AB::F>::range_check(
    builder,
    local.next_next_pc,
    local.next_next_pc_range_checker,
    is_real.clone(),
);
...
// When we are not branching, assert that local.next_pc + 4 <==> next.next_next_pc.
builder.when(is_real.clone()).when_not(local.is_branching).assert_eq(
    local.next_pc.reduce::<AB>() + AB::Expr::from_canonical_u32(4),
    local.next_next_pc.reduce::<AB>(),
);
```

The safety of this code depends on the range checks above which are supposed to constrain `next_pc` and `next_next_pc` to be words in the KoalaBear field. However, the safety of this check depends on limbs of the two words being valid bytes. This check is missing. The check is not as important for `next_pc` since only the reduced form is kept in the lookup table. However, `next_next_pc` appears to be represented as columns and so its well-formedness is not preserved.

Impact The values of `next_next_pc` are underconstrained, allowing an attacker to set them to be any value which sums to `next_pc.reduce::AB<>() + 4`.

Recommendation Constrain the limbs of `next_next_pc` to be bytes.

Developer Response The developers fixed this issue in commit `5900b25`.

7.1.19 V-ZKM-VUL-019: Issue with converting BigUint with large digits to field elements

Severity	Low	**Commit**	ad200e3
Type	Data Validation	**Status**	Acknowledged
Location(s)	`crates/core/machine/src/operations/field/util.rs:11`		
Confirmed Fix At	N/A		

The function `biguint_to_field` converts `BigUint` elements to KoalaBear field elements by converting digits in a base `2^32` expansion to the corresponding field elements using `from_canonical_u32` and subsequent weighted addition:

```rust
fn biguint_to_field<F: PrimeField32>(num: BigUint) -> F {
    let mut x = F::ZERO;
    let mut power = F::from_canonical_u32(1u32);
    let base = F::from_canonical_u64((1 << 32) % F::ORDER_U64);
    let digits = num.iter_u32_digits();
    for digit in digits.into_iter() {
        x += F::from_canonical_u32(digit) * power;
        power *= base;
    }
    x
}
```

The behavior of `from_canonical_u32` when the input exceeds the field characteristic is undefined. In the case of the KoalaBear field, for each `u32` digit there is a 1/2 chance of falling into this range with undefined behavior.

Impact For `BigUint` elements exceeding the field characteristic the conversion might not yield the true value.

Recommendation Reduce digits modulo the prime order before passing to `from_canonical_u32`.

Developer Response The developers acknowledged this issue but they are not planning to fix it at the moment due to the minimal impact on the codebase.

7.1.20 V-ZKM-VUL-020: Inconsistent constraints on syscall_id in eval_syscall

Severity	Low	**Commit**	ad200e3
Type	Logic Error	**Status**	Fixed
Location(s)	crates/core/machine/src/syscall/[...]/air.rs:116-169		
Confirmed Fix At	4080c23		

Function `eval_syscall` constrains column `local.syscall_id` differently, depending on the type of the actual system call. Specifically, for all instructions except for `SyscallCode::ENTER_UNCONSTRAINED` and `SyscallCode::SYS_NOP`, the function relies on the two bytes passed to `local.prev_a_value` to constraint column `local.syscall_id`. Whereas for the `ENTER_UNCONSTRAINED` and `SYS_NOP` cases, the function relies on the witness to properly constraint `local.syscall_id` as shown by the following snippet.

```
// Compute whether this syscall is SYS_NOP.
let is_sys_nop = {
    IsZeroOperation::<AB::F>::eval(
        builder,
        local.syscall_id - AB::Expr::from_canonical_u32(SyscallCode::SYS_NOP.
    syscall_id()),
        local.is_sys_nop,
        local.is_real.into(),
    );
    local.is_sys_nop.result
};

// [VERIDISE]: ... omitted for brevity.

builder
   .when(local.is_real)
   .when_not(is_enter_unconstrained + is_sys_nop)
   .assert_eq(local.syscall_id, syscall_id.clone());
```

Snippet 7.6: Snippet from `eval_syscall`

However, as shown below, the same function uses a local expression called `syscall_id`, which is a function of `local.prev_a_value` to compare against the `SyscallCode::ENTER_UNCONSTRAINED` case.

```
// The syscall code is the read-in value of op_a at the start of the instruction.
let syscall_code = local.prev_a_value;

// We interpret the syscall_code as little-endian bytes and interpret each byte as a
    u8
// with different information.
let syscall_id = syscall_code[0] + syscall_code[1] * AB::Expr::from_canonical_u32
    (256);

// [VERIDISE]: ... omitted for brevity.

// Compute whether this syscall is ENTER_UNCONSTRAINED.
let is_enter_unconstrained = {
    IsZeroOperation::<AB::F>::eval(
        builder,
        syscall_id.clone()
```

```
            - AB::Expr::from_canonical_u32(SyscallCode::ENTER_UNCONSTRAINED.
    syscall_id()),
        local.is_enter_unconstrained,
        local.is_real.into(),
    );
    local.is_enter_unconstrained.result
};
```

Snippet 7.7: Snippet from `eval_syscall`

Impact This leaves column `local.syscall_id` unconstrained for the `ENTER_UNCONSTRAINED` system call. However, this column is not being used in any other meaningful constraint, so the impact is minimal.

Recommendation Handle both ways uniformly or remove column `local.syscall_id` all together and rely on `local.prev_a_value` for all system call identifiers.

Developer Response The developers fixed the issue in commits `4080c23` and `3a2cc01`.

7.1.21 V-ZKM-VUL-021: Maintainability concerns

Severity	Warning	**Commit**	ad200e3
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/core/machine/src/` ▶ `bytes/air.rs:31-69` ▶ `memory/` • `global.rs:254, 278` • `instructions/` ∗ `air.rs:144` ∗ `columns.rs:58-60` ▶ `operations/` • `field/` ∗ `field_inner_product.rs:26` ∗ `range.rs:21, 64` ∗ `util.rs:41` • `global_lookup.rs:178, 181` ▶ `program/mod.rs:42` ▶ `syscall/instructions/` • `air.rs:334-337, 363-366`		
Confirmed Fix At	24c9912		

This issue aggregates several maintainability concerns currently present in the codebase. These issues do not pose any immediate risk to the zkVM, but they might increase the probability of introducing issues in future versions of the code.

1. Cases of inconsistent or outdated documentation:
   a) `crates/core/machine/src/operations/global_lookup.rs:181`: The comment should be 2^30 - 2^23 rather than 2^30 - 2^24. The implementation is correct.
   b) `crates/core/machine/src/operations/field/range.rs:21`: The comment on that line and other places in the implementation of `FieldLtCols` assume that the `rhs` of the comparison is the modulus, which is inconsistent with the top-level documentation of the struct.
   c) `crates/core/machine/src/operations/field/field_inner_product.rs:L26`: `a` and `b` are vectors of field elements, not field elements as the comment suggests.
   d) `crates/core/machine/src/syscall/instructions/air.rs:363-366`: The comments do not mention `is_exit_group`.
   e) `crates/core/machine/src/syscall/instructions/columns.rs:69`: The comments mention the babybear prime instead of koalabear.
   f) `crates/core/machine/src/operations/field/util.rs:41`: The comment refers to a single hardcoded value of `offset`, which is a function parameter.
   g) `crates/core/machine/src/program/mod.rs:42`: The comment is inconsistent with the chip.
   h) `crates/core/machine/src/memory/instructions/air.rs:144`: The comment mentions the babybear prime instead of koalabear.
   i) `crates/core/machine/src/memory/instructions/columns.rs:58-60`: `addr_offset` has been renamed to `addr_ls_two_bits`.
2. `crates/core/machine/src/operations/global_lookup.rs:178`: Variable name `x3_2x_26z5` is inconsistent with the name of the curve used by Ziren. It should be referring to x^3 + 3z*x -3.

3. `crates/core/machine/src/operations/field/range.rs:64`: Function `eval` assumes that all limbs are range checked by the users of `FieldLtCols`. It is recommended to document this assumption.
4. `crates/core/machine/src/syscall/instructions/air.rs:334-337`: This code snippet is repeated in multiple places in this file. It is recommended to extract this logic to a function so the snippets will not become out of sync.
5. `crates/core/machine/src/memory/global.rs:254,278`: Variable `values` is never used.
6. `crates/core/machine/src/bytes/air.rs:31-69`: This loops assumes that `ByteOpcode::all` returns the opcodes in the same order that their defined in the enum. Indexing `multiplicities` with `i` is a bit dangerous because if this assumption gets violated in the future, the receives will not be correct. It will be safer to index `multiplicities` with the `opcode`.

Recommendation Address all maintainability concerns mentioned above.

7.1.22 V-ZKM-VUL-022: Constraint on send_to_table has multiple solutions

Severity	Warning	**Commit**	ad200e3
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/core/machine/src/syscall/[...]/air.rs:117-121`		
Confirmed Fix At	05e8ad8		

Function `eval_syscall` uses a local expression named `send_to_table` as the multiplicity argument of calls to `send_syscall` (which contributes to the lookup argument). The intention is for all terms of `send_to_table` to be zero on padding rows, as demonstrated by the following code snippet:

```
let send_to_table = syscall_code[2] + local.is_sys_linux;

// SAFETY: Assert that for non real row, the send_to_table value is 0 so that the `
    send_syscall`
// interaction is not activated.
builder.when(AB::Expr::ONE - local.is_real).assert_zero(send_to_table.clone());
```

Snippet 7.8: Snippet from `eval_syscall`

However, the equation `syscall_code[2] + local.is_sys_linux = 0` can have multiple solutions because both terms of the equation are not constrained to be booleans. Even though the zkVM is implicitly trusting guest programs to honestly set `syscall_code[2]` as a boolean, the same does not hold for `local.is_sys_linux`

Impact Currently, there is no security impact of the above issue, since all other uses of `local.is_sys_linux` in constrained are guarded by `local.is_real` (i.e., non-padding rows). However, this might change in future versions of the codebase.

Recommendation Constrain column `local.is_sys_linux` (at least) to be a boolean.

Developer Response The developers fixed this issue in commit `05e8ad8`.

7.1.23 V-ZKM-VUL-023: Unnecessary constraints

Severity	Info	**Commit**	ad200e3
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/core/machine/src/` ▶ `air/program.rs:20-28` ▶ `memory/instructions/air.rs:496-501`		
Confirmed Fix At	`https://github.com/ProjectZKM/Ziren/pull/328`		

The following locations contain unnecessary constraints or redundant values in lookups:

1. `crate/core/machine/src/memory/instructions/air.rs:469-501`: This constraint is trivially satisfied.
2. `crates/core/machine/src/air/program.rs:20-28`: The instruction opcode appears twice in the lookup values.
3. `crates/core/machine/src/operations/add.rs:74-76`: These constraints are implied by constraints on line 90-92, 80-82, 85-87 and hence can be dropped.
4. `crates/core/machine/src/operations/add.rs:45`:
   `debug_assert_eq!` is used to check that `overflow` is either `0` or `256` by multiplying `overflow` with `overflow-256`. As multiplication is performed modulo `2^32` the product can be zero without any of the factor being zero (`overflow=2^31` gives `2^31 * (2^31 - 256)` which will be `0` modulo `2^32`). The assert does not do the intended check.

Impact There are no security implications due to the above, they might only make proving slightly more expensive.

Recommendation Implement the above optimizations.

Glossary

AIR constraints Aalgebraic rules that ensure each step of a computation follows the correct machine behavior. If all constraints hold across the trace, the computation is proven valid . 1

LLZK A ZK circuit IR built in MLIR. See https://github.com/Veridise/llzk-lib. 9

MIPS MIPS is a computer architecture that stands for "Microprocessor without Interlocked Pipelined Stages". See https://en.wikipedia.org/wiki/MIPS_architecture for more details. 1

smt Satisfiability Modulo Theories. The problem of determining whether a certain mathematical statement has any solutions. SMT solvers attempt to do this automatically. See https://en.wikipedia.org/wiki/Satisfiability_modulo_theories to learn more. 5

SNARK Succinct Non-interactive ARguments of Knowledge. Cryptographic proofs that let a verifier check the correctness of a computation using a very small proof and minimal verification time . 1

STARK Scalable Transparent ARguments of Knowledge. A cryptographic proof system that verifies the correctness of computations by expressing their execution as algebraic constraints over a trace. 1

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. 1, 42

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