Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

RISC Zero zkVM

Veridise Inc.

Jan. 30, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Shankara Pailoor
Tim Hoffman
Daniel Dominguez
Vijaykumar Singh
Benjamin Mariano
Alp Bassa

► **Contact Us:**

contact@veridise.com

► **Version History:**

Feb. 24 2025	V4
Jan. 30 2025	V3
Jan. 27 2025	V2
Jan. 16 2025	V1
Dec. 12, 2024	Initial Draft

Contents

1 Executive Summary

From July 29, 2024 to December 13, 2024, RISC Zero engaged Veridise to conduct a comprehensive security audit of their RISC-V zkVM and V2 circuit. The review was carried out in two phases: the first phase took place from Jul. 29 to Oct. 14, 2024 and covered 1) the recursive STARK-to-STARK and STARK-to-SNARK circuits, 2) their V2 RISC-V zkVM written in a custom DSL called Zirgen, and 3) receipt verification within the zkVM. The review was performed on the following GitHub repositories:

- `risc0/risczero-wip` (private repository) on commit `f7fae1d`*.
- `risc0/risc0` on commit `a6159d9`.

The second phase occurred from Nov. 3 to Dec. 13, 2024 and covered RISC Zero's host and prover implementations in the risc0 repository on commit `35c65de`. In total, Veridise conducted the assessment over 96 person-weeks, with 6 security analysts reviewing the project over 16 weeks. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts, as well as a thorough code review.

Project Summary. The RISC Zero zkVM allows one to generate proofs of execution of RISC-V programs. The high-level architecture is described in the following whitepaper. Veridise's security assessment of the RISC Zero zkVM was broken into two phases and included a mix of manual and tool-assisted analysis. The following parts of the zkVM were reviewed:

- **V2 RISC-V Circuit Implementation**. RISC Zero developed a new circuit implementation (referred to as V2) of the RISC-V CPU in their custom ZK Circuit DSL called Zirgen. Veridise analysts utilized Veridise's new implementation of Picus [1] along with manual analysis to review this code.
- **Recursion Circuits**. The RISC Zero zkVM relies on recursion to generate easily verifiable proofs of execution. RISC Zero wrote a custom STARK-to-STARK recursion circuit in an embedded DSL as well as a STARK-to-SNARK recursion circuit in Circom. The first circuit was reviewed manually, and the latter was reviewed using a combination of manual analysis and using the open source version of Picus which supports Circom.
- **ELF Decoding**. The first step of the RISC Zero prover pipeline involves decoding a RISC-V ELF binary into an executable image. The RISC Zero developers wrote a custom converter for this step and Veridise analysts utilized fuzz testing to check the decoding logic.
- **STARK Verifiers**. RISC Zero uses a STARK-based proving scheme, and the Veridise analysts manually reviewed the STARK verifier implementations.
- **RISC Zero's Prover**. RISC Zero implemented a custom prover to generate proofs of execution. The prover performs instruction decoding, instruction execution, and then generates a STARK proof. Veridise analysts utilized a mix of manual review and fuzz testing to assess this part.

* The contents of risczero-wip have been made public in the repository `risc0/zirgen`

Code Assessment. The RISC Zero developers provided Veridise analysts access to their private V2 circuit implementation written in Zirgen, their circuit DSL. The documentation of the Zirgen DSL is extremely limited, so Veridise analysts relied on the RISC Zero developers to understand the language semantics. The developers also described the high-level architecture of the RISC Zero zkVM and pointed the Veridise analysts to documentation on their website which describes the different parts of the zkVM including English descriptions of various data structures used in the implementation. Overall, Veridise analysts found parts of the documentation to be helpful, but also believe the documentation could be improved (see the Recommendations paragraph below).

The code included several test suites that exercised the different aspects of the zkVM. Veridise analysts found the test suites helpful in understanding the code and when developing harnesses to fuzz different parts of the zkVM.

Summary of Issues Detected. The security assessment uncovered 41 issues, 5 of which are assessed to be of high or critical severity by the Veridise analysts. Four of these issues are due to underconstrained bugs in the V2 circuits detected by Picus (V-RISC0-VUL-001, V-RISC0-VUL-002, V-RISC0-VUL-003, V-RISC0-VUL-004), and one is an overconstrained bug that was detected manually (V-RISC0-VUL-005). The Veridise analysts also identified 4 medium security vulnerabilities including V-RISC0-VUL-006 and V-RISC0-VUL-007 which describe underconstrained V2 circuits found by Picus in the Poseidon external call. Veridise analysts also found 1 low-severity issue, 15 warnings, and 9 informational findings. The RISC Zero developers acknowledged all issues and have fixed all critical and high severity issues along with the two medium severity circuit bugs (V-RISC0-VUL-006, V-RISC0-VUL-007).

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the overall quality of RISC Zero zkVM.

- There should be detailed documentation describing the usage scenarios and associated threat model(s) of the RISC Zero zkVM prover (see V-RISC0-VUL-009). This documentation is important for both the client and auditors to be able to assess whether code in the prover has a vulnerability or not.
- The analysts recommended that the RISC Zero developers incorporate some formal verification in the development workflow, particularly to check for underconstrained vulnerabilities. In general, it is hard to detect underconstrained bugs via traditional testing, and tools like Picus are valuable for detecting such issues. In response, the client has started incorporating Picus into their development workflow via Veridise's AuditHub.
- Veridise analysts also identified several places where documentation was outdated and assumptions were not explicitly stated in the code (see V-RISC0-VUL-033). There were also many places where code quality could be improved (V-RISC0-VUL-032).

Disclaimer. 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
risc0/risc0	`a6159d9`	Rust & C++	RISC-V zkVM
risc0/risczero-wip	`f7fae1d`	Zirgen & C++	RISC-V zkVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 29–Dec. 13, 2024	Manual & Tools	6	96 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	4	4	4
High-Severity Issues	1	1	1
Medium-Severity Issues	4	4	4
Low-Severity Issues	2	2	2
Warning-Severity Issues	19	19	2
Informational-Severity Issues	11	11	0
TOTAL	41	41	13

Table 2.4: Category Breakdown.

Name	Number
Data Validation	13
Logic Error	8
Maintainability	7
Underconstrained Circuit	6
Denial of Service	2
Overconstrained Circuit	1
Arithmetic Overflow	1
Cryptographic Vulnerability	1
Documentation	1
Gas Optimization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of the RISC Zero zkVM, specifically the V2 circuits along with parts of the Risc Zero prover and verifier. During the assessment, the security analysts aimed to answer questions such as:

- Do the V2 circuits correctly encode RISC-V semantics?
- Are any of the V2 or recursion circuits under- or over-constrained?
- Can the guest program escape RISC Zero's host execution environment?
- Are there any denial of service vulnerabilities in RISC Zero's prover?
- Does RISC Zero correctly perform the Fiat-Shamir transformation in their STARK verifier?
- Does RISC Zero properly decode ELF-binaries into executable images?
- Is RISC Zero's Rust-based finite field implementation correct? In particular, does RISC Zero's finite field implementation allow for invalid field elements to be created (e.g, out-of-field values)?
- Are the different components of RISC Zero zkVM thoroughly tested and documented?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved human experts performing an extensive manual review of the source code assisted with 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 V2 Circuits, the security analysts utilized a new, proprietary version of Picus [1]. Picus can both prove (or find counterexamples) that a circuit is deterministic using a combination of static analysis and SMT solvers.
- *Fuzzing/Property-based Testing.* Since the RISC Zero zkVM generates traces of RISC-V executions, the Veridise analysts utilized fuzz testing to ensure that the prover both conforms to the RISC-V specification (ELF binary parsing, instruction decoding, and instruction execution) and does not contain any common vulnerabilities like buffer overflows, memory leaks, double-frees, etc.

Scope. In the `risc0/risczero-wip` repository, the scope of the audit was limited to the following directories:

- `zirgen/circuit/predicates/`
- `zirgen/circuit/recursion/`
- `zirgen/circuit/rv32im/{shared/*.rs,v2/*}`
- `zirgen/circuit/verify/circom/`
- `zirgen/components/`

In the `risc0/risc0` repository, the scope was limited to the following directories:

- `risc0/binfmt/src/`
- `risc0/core/src/*.rs`
- `risc0/core/src/field/*`
- `risc0/circuit/`
- `risc0/groth16/src/*`
- `risc0/zkp/src/core/*`
- `risc0/zkp/src/core/hash/*`
- `risc0/zkp/src/hal/`
- `risc0/zkp/src/prove/*.rs`
- `risc0/zkp/src/verify/*`
- `risc0/zkvm/src/guest/*`
- `risc0/zkvm/platform/src/*.rs`
- `risc0/zkvm/platform/src/serde/`
- `risc0/zkvm/src/host/`
- `risc0/src/guest/*`
- `risc0/src/host/recursion/*`
- `risc0/src/host/receipt/*`

Methodology. Veridise security analysts reviewed the reports of previous audits for RISC Zero zkVM as found here: `https://github.com/risc0/rz-security/`, inspected the provided tests, and read the RISC Zero zkVM documentation. They then began a review of the code assisted by both fuzz-testing and formal verification.During the security assessment, the Veridise security analysts regularly met with the RISC Zero zkVM developers to ask questions about the code, and regularly communicated over a shared Slack channel.

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 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 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RISC0-VUL-001	component ExpandU32 is underconstrained	Critical	Fixed
V-RISC0-VUL-002	component DecomposeLow2 is underconstrained	Critical	Fixed
V-RISC0-VUL-003	DoDiv is underconstrained	Critical	Fixed
V-RISC0-VUL-004	Decoder is underconstrained	Critical	Fixed
V-RISC0-VUL-005	opLH is overconstrained	High	Fixed
V-RISC0-VUL-006	component PoseidonStoreOut is underconstrained	Medium	Fixed
V-RISC0-VUL-007	component PoseidonStoreState is underconstrained	Medium	Fixed
V-RISC0-VUL-008	improper usage of bytemuck::Pod	Medium	Fixed
V-RISC0-VUL-009	ASAN crash on recursion-sys FFI functions	Medium	Fixed
V-RISC0-VUL-010	Missing Data Validation in Syscall Execution	Low	Fixed
V-RISC0-VUL-011	Heap pointer overflow on large allocation	Low	Fixed
V-RISC0-VUL-012	BabyBear construction functions should validate input	Warning	Acknowledged
V-RISC0-VUL-013	BabyBear Elem operations should validate input	Warning	Acknowledged
V-RISC0-VUL-014	More rounds in Poseidon hash function	Warning	Acknowledged
V-RISC0-VUL-015	Missing pre-condition check in align_up()	Warning	Acknowledged
V-RISC0-VUL-016	send_recv_slice() is overly restrictive	Warning	Acknowledged
V-RISC0-VUL-017	Goldilocks functions should detect INVALID input	Warning	Fixed
V-RISC0-VUL-018	to_po2() is overly restrictive	Warning	Acknowledged
V-RISC0-VUL-019	hash_raw_data_slice() in guest does not match . . .	Warning	Acknowledged
V-RISC0-VUL-020	insufficient size checks in poly_interpolate()	Warning	Acknowledged
V-RISC0-VUL-021	Goldilocks construction functions should validate input	Warning	Fixed
V-RISC0-VUL-022	Segment permissions are ignored in ELF decoding	Warning	Acknowledged
V-RISC0-VUL-023	Multiple values have the same digest	Warning	Acknowledged
V-RISC0-VUL-024	Missing Overflow Check in load_elf	Warning	Acknowledged
V-RISC0-VUL-025	Invalid post state for paused	Warning	Acknowledged
V-RISC0-VUL-026	Possible poisoning of ZKR_REGISTRY mutex	Warning	Acknowledged
V-RISC0-VUL-027	Undocumented Security Assumptions	Warning	Acknowledged
V-RISC0-VUL-028	Low cycle cost for reading from user memory on . . .	Warning	Acknowledged
V-RISC0-VUL-029	User-triggered panic in syscall handling	Warning	Acknowledged
V-RISC0-VUL-030	Denial of service via child processes	Warning	Acknowledged
V-RISC0-VUL-031	Incorrect attribute application causes linking error . . .	Info	Acknowledged
V-RISC0-VUL-032	Code Quality	Info	Acknowledged
V-RISC0-VUL-033	Documentation	Info	Acknowledged
V-RISC0-VUL-034	Performance	Info	Acknowledged
V-RISC0-VUL-035	Deprecated dependencies	Info	Acknowledged
V-RISC0-VUL-036	bit_reverse() panics on trivial case	Info	Acknowledged
V-RISC0-VUL-037	Typos, unused program constructs, and other small fixes	Info	Acknowledged
V-RISC0-VUL-038	Undocumented Assumption on Deserialization . . .	Info	Acknowledged
V-RISC0-VUL-039	Overflow Leads to Lost Data on Serialization	Info	Acknowledged
V-RISC0-VUL-040	MachineContext crashes if preflight traces are empty	Info	Acknowledged
V-RISC0-VUL-041	If the power of two value is too low the witness . . .	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-RISC0-VUL-001: component ExpandU32 is underconstrained

Severity	Critical	**Commit**	f7fae1d
Type	Underconstrained Circuit	**Status**	Fixed
File(s)	`risczero-wip/zirgen/circuit/rv32im/v2/dsl/mult.zir`		
Location(s)	ExpandU32		
Confirmed Fix At	`https://github.com/risc0/risczero-wip/pull/619`, 6490b6c		

The component `ExpandU32`, shown below, takes as input a value `x` which is of type `ValU32`,along with a boolean variable `signed` which indicates whether `x` is a signed value, and returns bytes `b0, b1, b2, b3` which represent the `ith` byte of `x`. A `ValU32` type is a tuple of Felts `(low, high)` where `low` is intended to capture the lower 16 bits and `high` captures the upper 16 bits.

```
// Expand a u32 into bytes, and extract the sign bit.  We then set an additional
// 'neg' bit based on the combination of the actual sign bit and if we are
// interpreting the value as signed.
component ExpandU32(x: ValU32, signed: Val) {
  b0 := NondetU8Reg(x.low & 0xff);
  b1 := NondetU8Reg((x.low & 0xff00) / 0x100);
  b2 := NondetU8Reg(x.high & 0xff);
  // We decompose the top byte into sign bit and lower 7 bits
  // In so doing, we multiply the top 7 bits by 2 so the range check enforces
  // the fact that the value is only 7 bits
  // If the value is set to an odd number by an invalid prover, it will
  // make the verification of x.high fail to be in range of a u16
  b3Top7times2 := NondetU8Reg((x.high & 0x7f00) / 0x80);
  topBit := NondetBitReg((x.high & 0x8000) / 0x8000);
  // Now we verify that our guessed values match x
  x.low = b0 + b1 * 0x100;
  x.high = b2 + b3Top7times2 * 0x80 + topBit * 0x8000;
  // Now compute neg
  neg := topBit * signed;
  // Now we make b3, which should equal the orignal byte (and is linear).
  b3 := b3Top7times2 / 2 + 0x80 * topBit;
}
```

At a high level, the above code encodes the following constraints:

1. The first three line asserts that `b0`, `b1` and `b2` is a u8 value i.e $0 \leq b0, b1, b2 \leq 255$.
2. Next `x.low = b0 + b1 * 0x100` asserts that `b0` and `b1` correctly decompose the lower 16 bits.
3. `b3Top7times2` is constrained to be a value in $0 \leq b3Top7times2 \leq 255$ and is meant to encode $2 * k$ where k represents the bottom 7 bits of `x.high`.
4. `topBit := NondetBitReg((x.high & 0x8000) / 0x8000)` defines a felt `topBit` that is constrained to be a bit (either 0 or 1) and is meant to encode the top bit of `x`.

5. Finally, `x.high = b2 + b3Top7times2 * 0x80 + topBit * 0x8000` and `b3 := b3Top7times2 / 2 + 0x80 * topBit` define the value of `b3` and **attempt** to assert that `x.high` is correctly decomposed.

The issue is that the above constraints do not properly constrain the felt `b3Top7times2` to encode the lower 7 bits of `x.high`. This is due to a missing range check that `b3` is a u8 value. As such, even though `b3Top7times2` is constrained to be a u8 value, `b3Top7times2/2` does not because if `b3Top7times2` is odd, then the resulting division (which is actually multiplication by the field inverse) will yield a value larger than a u8.

Impact As a result, a malicious prover can incorrectly decompose `x` when `signed = 0` by setting `b3Top7times2` to be an appropriate odd value. For example, suppose we are given inputs `{x = ValU32(50801, 32832), signed = 0}`. Then a prover can generate the following outputs:

```
{ b0 = 113, b1 = 198, b2 = 192, b3 = 1006633088, b3Top7times2 = 255, topBit = 1}
```

whereas the intended output is:

```
{ b0 = 113, b1 = 198, b2 = 64, b3 = 128, b3Top7times2 = 128, topBit = 0}
```

Recommendation We recommend asserting that `b3` is a u8 value just like `b0, b1, b2`.

Developer Response The developers have fixed the issue in this PR.

4.1.2 V-RISC0-VUL-002: component DecomposeLow2 is underconstrained

Severity	Critical	Commit	f7fae1d
Type	Underconstrained Circuit	Status	Fixed
File(s)	risczero-wip/zirgen/circuit/rv32im/v2/dsl/inst_ecall.zir		
Location(s)	DecomposeLow2		
Confirmed Fix At	https://github.com/risc0/zirgen/pull/140, 38a7797		

The component `DecomposeLow2`, shown below, takes as input a `Val` type `len` and breaks it up into two limbs `high` and `low2`. The intent, as seen in the code below, is for 1) the input `len` to be a valid `U16`, 2) `low2` to denote the 2 least significant bits of `len` and 3) `high` to be the remaining bits of `len` divided by 4. However, the circuit does not express this constraint and the values of `high` and `low2` are not related at all to `len`. Thus, the output values are nearly completely unconstrained.

```
component DecomposeLow2(len:Val){
// We split len into a multiple of 4, and the low 2 bits as one hot
  high := NondetReg((len&0xfffc) /4);
  low2 := NondetReg(len&0x3);
  low2Hot := OneHot<4>(low2);
  highZero := IsZero(high);
  isZero := Reg(highZero* low2Hot[0]);
  low2Zero := low2Hot[0];
  low2Nonzero := low2Hot[1] + low2Hot[2] + low2Hot[3];
}
```

This issue can be fixed by adding the following constraint: `len = 4*high + low2` and by changing `high = NondetReg((len & 0xfffc) / 4)` to `high = NondetU16Reg((len & 0xfffc) / 4)`.

Impact A malicious prover can set `isZero` to either 0 or 1 and can set any `low2Hot[i] = 1` so long as `low2Hot[j] = 0` for every `j != i` regardless of `len`.

Recommendation We recommend adding the constraint `len = 4*high + low2` and by changing the statement `high = NondetReg((len & 0xfffc) / 4)` to `high = NondetU16Reg((len & 0xfffc) / 4)`.

Developer Response The developers have fixed the issue in this PR.

4.1.3 V-RISC0-VUL-003: DoDiv is underconstrained

Severity	Critical	**Commit**	f7fae1d
Type	Underconstrained Circuit	**Status**	Fixed
File(s)	`zirgen/circuit/rv32im/v2/dsl/inst_div.zir`		
Location(s)	DoDiv		
Confirmed Fix At	https://github.com/risc0/zirgen/pull/144, cc4d252		

The circuit `DoDiv`, as seen below, takes as inputs two `U32` values `numer` and `denom` and outputs two other `U32` values `quot` and `rem` such that `numer = quot * denom + rem`.

```
component DoDiv(numer: ValU32, denom: ValU32, signed: Val, ones_comp: Val) {
  // Guess the answer
  guess := Divide(numer, denom, signed + 2 * ones_comp);
  // The quotient gets pulled into bytes during multiply anyway, so
  // no need to verify it is make of U16s
  quot_low := NondetReg(guess.quot.low);
  quot_high:= NondetReg(guess.quot.high);
  quot := ValU32(quot_low, quot_high);
  // The remainder however needs to be constrained
  rem_low := NondetU16Reg(guess.rem.low);
  rem_high:= NondetU16Reg(guess.rem.high);
  rem := ValU32(rem_low, rem_high);
  // Either all signed, or nothing signed
  settings := MultiplySettings(signed, signed, signed);
  // Do the acuumulate
  mul := MultiplyAccumulate(quot, denom, rem, settings);
  // Check the main result (numer = quot * denom + rem
  AssertEqU32(mul.outLow, numer);
  // The top bits should all be 0 or all be 1
  topBitType := NondetBitReg(1 - Isz(mul.outHigh.low));
  AssertEqU32(mul.outHigh, ValU32(0xffff * topBitType, 0xffff * topBitType));
  DivideReturn(quot, rem)
}
```

Snippet 4.1: Snippet from `DoDiv`

The intention of the above circuit is to utilize the Division Algorithm which states that given unique n and $m > 0$ there is is a unique quotient q and remainder r such that $n = m * q + r$ and $r < m$. However, the crucial check that $r < m$ has been omitted from the above code which makes the underconstrained. This is because without that check, for any q, we can set $r = n - m * q$ and satisfy the constraints.

To validate this we first created the following simple test case which set `numer = 2`, `denom = 1` and invoked `DoDiv`. We then used Picus to find two distinct values for the `quot` and `rem` that satisfied the constraints:

```
component PicusDoDiv(num : ValU32, denom: ValU32) {
    num.low = 2;
    num.high = 0;
    signed := 0;
    ones_comp := 0;
    ret := DoDiv(num, denom, signed, ones_comp);
```

```
}
```

Picus returned the following result:

```
inputs: [("num_low", 2), ("num_high", 0), ("denom_low", 1), ("denom_high3", 0)]

first output assignment: [("rem.low", 0), ("rem.high", 0), ("quot.low", 2), ("quot.high", 0)]

second output assignment: [("rem.low", 2), ("rem.high", 0), ("quot.low", 0), ("quot.high", 0)]
```

This further confirmed that the circuit was underconstrained.

Impact A malicious prover could set the value of `quot` or `rem` to whatever values they want.

Recommendation We recommend adding a constraint asserting that `rem < denom`.

Developer Response Technically, we need to conditionally check [the remainder] based on whether we are in divide-by-zero case. rv32im doesn't fault on divide-by-zero, it just returns $q = -1, r = n$, so we still have $n = m * q + r$, but not $r < m$. If we are in $m = 0$, then we need to check $r = n$.

The developers have fixed the issue in this PR.

4.1.4 V-RISC0-VUL-004: Decoder is underconstrained

Severity	Critical	**Commit**	f7fae1d
Type	Underconstrained Circuit	**Status**	Fixed
File(s)	`zirgen/circuit/rv32im/v2/dsl/decode.zir`		
Location(s)	Decoder		
Confirmed Fix At	`https://github.com/risc0/zirgen/pull/140`, 38a7797		

The `Decoder` circuit which takes a Risc-V instruction, represented as a `u32` and specifies how the different values/opcodes from the instruction jointly compose the instruction is underconstrained. The root cause is the following excerpt from the circuit:

```
_rd_0 := NondetTwitReg((inst.low & 0x0080) / 0x0080);

 // The opcode is special and is unconstrained.
 // This implies the for the decoding to be fully correct, some later
 // mechanism must in fact constrain the opcode.
 opcode := NondetU8Reg(inst.low & 0x7f); // added based on previous conversation (
   will remove soon)
 opcode2x := NondetU8Reg((inst.low & 0x7f) * 2); // added based on previous
   conversation
 opcode = opcode2x / 2; // added based on previous conversation

 // Verify the components do in fact compose into the instructions
 inst.high = _f7_6   * 0x8000 +
             _f7_45  * 0x2000 +
             _f7_23  * 0x0800 +
             _f7_01  * 0x0200 +
             _rs2_34 * 0x0080 +
             _rs2_12 * 0x0020 +
             _rs2_0  * 0x0010 +
             _rs1_34 * 0x0004 +
             _rs1_12 * 0x0001;
 inst.low = _rs1_0 * 0x8000 +
            _f3_2  * 0x4000 +
            _f3_01 * 0x1000 +
            _rd_34 * 0x0400 +
            _rd_12 * 0x0100 +
            _rd_0  * 0x0080 +
            opcode;
```

Snippet 4.2: Snippet from `Decoder()`

In particular, the correctness of the circuit relies on all the internal signals `_rs1_0`, `_f3_2`, `_f3_01`, `_rd_34`, `_rd_12` , and `_rd_0` all having the appropriate ranges whereby the Division Algorithm guarantees unique values for each of these signals. However, `_rd_0` is set to a `NondetTwitReg` which allows its value to range from 0 to 3. As such `_rd_0 * 0x0080` 's range overlaps with `_rd_12*0x0100` and so a malicious prover can (in many cases) have freedom in what values they choose for `_rd_0` and `_rd_12` for a given input.

We used our tool Picus to prove the circuit is nondeterministic. We constructed the following test case:

```
component PicusDecoderTest() {
    inst := ValU32(555, 0);
    ret := Decoder(inst);
}
```

and set `ret` to be the output value and ran Picus on it. It found that `immB.low` could be either `4096` or `4` depending on how the prover sets `_rd_0` or `_rd_12`.

Impact A malicious prover can manipulate the decoding of the instructions to values that are not intended.

Recommendation We recommend changing `_rd_0` to a `NondetBitReg` instead of a `NondetTwitReg`. We ran Picus on the fixed circuit after that change and it proved the circuit was deterministic.

Developer Response The developers have acknowledged and fixed the issue in this PR.

4.1.5 V-RISC0-VUL-005: opLH is overconstrained

Severity	High	**Commit**	f7fae1d
Type	Overconstrained Circuit	**Status**	Fixed
File(s)	`risczero-wip/zirgen/circuit/rv32im/v2/dsl/inst_mem.zir`		
Location(s)	opLH		
Confirmed Fix At	`https://github.com/risc0/risczero-wip/pull/626`, 967da01		

The component `opLH` , shown below, validates that a memory load corresponds to a RISCV LH instruction which loads a short from memory. The resulting value is equal to a sign extended `low16` . However, the below constraints over-constrain `low16` as the the lower 15 bits (multiplied by 2) are restricted to 8 bits when they should fit into 16. Thus, many valid values cannot satisfy the constraints.

```
component OpLH(input: MemLoadInput) {
  VerifyOpcodeF3(input.decoded, 0x03, 0x1);
  input.addr.low0 = 0;
  low16 := input.addr.low1 * input.data.high + (1 - input.addr.low1) * input.data.low;
  highBit := NondetBitReg((low16 & 0x8000) / 0x8000);
  low15x2 := NondetU8Reg((low16 & 0x7fff) * 2);
  low16 = highBit * 0x8000 + low15x2 / 2;
  ValU32(low16, 0xffff * highBit)
}
```

Impact Many valid executions would not satisfy the constraints because the short is in fact restricted to be a byte.

Recommendation Change `low15x2 := NondetU8Reg(...)` to `low15x2 := NondetU16Reg`.

Developer Response The developers have fixed the issue in this PR.

4.1.6 V-RISC0-VUL-006: component PoseidonStoreOut is underconstrained

Severity	Medium	**Commit**	f7fae1d
Type	Underconstrained Circuit	**Status**	Fixed
File(s)	`risczero-wip/zirgen/circuit/rv32im/v2/dsl/inst_p2.zir`		
Location(s)	PoseidonStoreOut		
Confirmed Fix At	`https://github.com/risc0/zirgen/pull/140`, 38a7797		

The component `PoseidonStoreOut` , shown below, writes the field elements from the Poseidon computation to memory. Since memory writes need to be word aligned, the code first converts the field values into `u32` values.

```
component PoseidonStoreOut(cycle: Val, prev: PoseidonState) {
  Log("Store Out");
  for i : 0..8 {
    val := prev.inner[i] * ToMontgomery();
    low := NondetU16Reg(val & 0xffff);
    high := U16Reg((val - low) / 65536);
    MemoryWrite(cycle, prev.bufOutAddr + i, ValU32(low, high));
  };
  nextState :=
    prev.hasState * StatePoseidonStoreState() +
    (1 -  prev.hasState) * StateDecode();
  PoseidonState(GetDef(prev), nextState, 0, 0, 0, prev.inner)
}
```

Snippet 4.3: Snippet from `PoseidonStoreOut`

However, the conversion of field element into a u32 component in the above code is not correct. To see why, observe that the above code generates the following constraints on `high` and `low`:

$val = (high * 65536)\%p + low \land 0 \leq high, low < 2^{16}$.

Since `high` is a `u16` , the product `high * 65536 + low` is larger than the BabyBear prime (when interpreted as integers). Thus, the computation can overflow for sufficiently large `high` and `low` values. For example, if `val = 0` then one solution is `high = 0` and `low = 0` and another solution is `high = 30720` , `low = 1`.

Impact Every malicious prover has the ability to choose which of the above two solutions they wish to write to memory and so can generate a valid proof of an incorrect hash computation.

Recommendation We recommend adding a utility to `u32.zir` which properly converts a `felt` into a `U32` and using that in this context.

Developer Response The developers have fixed this issue in the following PR.

4.1.7 V-RISC0-VUL-007: component PoseidonStoreState is underconstrained

Severity	Medium	Commit	f7fae1d
Type	Underconstrained Circuit	Status	Fixed
File(s)	risczero-wip/zirgen/circuit/rv32im/v2/dsl/inst_p2.zir		
Location(s)	PoseidonStoreState		
Confirmed Fix At	https://github.com/risc0/zirgen/pull/140, 38a7797		

The component `PoseidonStoreState` , shown below, writes the field upper 8 elements from the inner sponge of the Poseidon hash function to memory. Because memory needs to be word aligned, the field elements are converted to `U32` values before writing to memory.

```
component PoseidonStoreState(cycle: Val, prev: PoseidonState) {
  Log("Store State");
  for i : 0..8 {
    val := prev.inner[16 + i];
    low := NondetU16Reg(val & 0xffff);
    high := U16Reg((val - low) / 65536);
    MemoryWrite(cycle, prev.stateAddr+ i, ValU32(low, high));
  };
  PoseidonState(GetDef(prev), StateDecode(), 0, 0, 0, prev.inner)
}
```

Snippet 4.4: Snippet from `PoseidonStoreState`

However, the conversion of field element into a u32 component in the above code is not correct. To see why, observe that the above code generates the following constraints on `high` and `low`:

$val = (high * 65536)\%p + low \land 0 \leq high, low < 2^{16}$.

Since `high` is a `u16` , the product `high * 65536 + low` is larger than the BabyBear prime (when interpreted as integers). Thus, the computation can overflow for sufficiently large `high` and `low` values. For example, if `val = 0` then one solution is `high = 0` and `low = 0` and another solution is `high = 30720` , `low = 1`.

Impact Every malicious prover has the ability to choose which of the above two solutions they wish to write to memory and so can generate a valid proof of an incorrect hash computation.

Recommendation We recommend adding a utility to `u32.zir` which properly converts a `felt` into a `U32` and using that in this context.

Developer Response The developers have fixed the issue in the following PR.

4.1.8 V-RISC0-VUL-008: improper usage of bytemuck::Pod

Severity	Medium	**Commit**	a6159d9
Type	Data Validation	**Status**	Fixed
File(s)	`risc0/core/src/field/goldilocks.rs`		
Location(s)	multiple		
Confirmed Fix At	https://github.com/risc0/risc0/pull/2857/		

The structs `Elem` and `ExtElem` in `risc0/core/src/field/goldilocks.rs` implement the trait `bytemuck::pod::Pod`.

```
#[derive(Eq, PartialEq, Clone, Copy, Debug, Pod, Zeroable)]
#[repr(transparent)]
pub struct Elem(u64);
```

Snippet 4.5: Definition of `Elem` (i.e. `GoldilocksElem`)

```
#[derive(Eq, PartialEq, Clone, Copy, Debug, Pod, Zeroable)]
#[repr(transparent)]
pub struct ExtElem([Elem; EXT_SIZE]);
```

Snippet 4.6: Definition of `ExtElem` (i.e. `GoldilocksExtElem`)

One of the safety requirements of the `Pod` trait is:

> The type must allow any bit pattern (eg: no `bool` or `char`, which have illegal bit patterns). see:
> https://docs.rs/bytemuck/1.16.1/bytemuck/trait.Pod.html#safety

The `GoldilocksElem` struct violates this requirement. Only values `<P` should be allowed. Likewise, `GoldilocksExtElem` violates this requirement because it should contain only `GoldilocksElem` with value `<P`.

Impact The `risc0_core::field::Elem::from_u32_slice()` function that both of these structs inherit allows creating `GoldilocksElem` violating the `<P` requirement and thus also `GoldilocksExtElem` containing a `GoldilocksElem` violating the `<P` requirement.

Recommendation The derived `Pod` implementation for `GoldilocksElem` and `GoldilocksExtElem` structs should be removed and they should both instead implement `CheckedBitPattern+NoUninit` where the `CheckedBitPattern` implementation allows only values `<P`.

Developer Response The developers have fixed the issue in the following PR.

4.1.9 V-RISC0-VUL-009: ASAN crash on recursion-sys FFI functions

Severity	Medium	**Commit**	a6159d9
Type	Data Validation	**Status**	Fixed
File(s)			
Location(s)			
Confirmed Fix At	`https://github.com/risc0/risc0/pull/2759`		

As part of the fuzzing campaign of the FFI functions once harness runs a sequence of function calls from the `recursion-sys` module in random order. This harness has ASAN enabled. The fuzzers found crashing inputs in several of these functions that were grouped together because they share a common error report. An example of the ASAN report can be seen below.

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==119387==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000008 (pc 0xaaaad67d68fc bp 0xfffff0fb59c0 sp 0xfffff0fb5320 T0)
==119387==The signal is caused by a READ memory access.
==119387==Hint: address points to the zero page.
    #0 0xaaaad67d68fc  (/targets/recursion-sequence+0x6868fc) (BuildId: f8f2d70ce53c6306)
    #1 0xaaaad63dc468  (/targets/recursion-sequence+0x28c468) (BuildId: f8f2d70ce53c6306)
    #2 0xaaaad6882a88  (/targets/recursion-sequence+0x732a88) (BuildId: f8f2d70ce53c6306)
    #3 0xaaaad687cdcc  (/targets/recursion-sequence+0x72cdcc) (BuildId: f8f2d70ce53c6306)
    #4 0xaaaad687ede8  (/targets/recursion-sequence+0x72ede8) (BuildId: f8f2d70ce53c6306)
    #5 0xaaaad687e924  (/targets/recursion-sequence+0x72e924) (BuildId: f8f2d70ce53c6306)
    #6 0xaaaad687cb14  (/targets/recursion-sequence+0x72cb14) (BuildId: f8f2d70ce53c6306)
    #7 0xaaaad687caf8  (/targets/recursion-sequence+0x72caf8) (BuildId: f8f2d70ce53c6306)
    #8 0xaaaad63a3c60  (/targets/recursion-sequence+0x253c60) (BuildId: f8f2d70ce53c6306)
    #9 0xaaaad687ca84  (/targets/recursion-sequence+0x72ca84) (BuildId: f8f2d70ce53c6306)
    #10 0xaaaad687ee30  (/targets/recursion-sequence+0x72ee30) (BuildId: f8f2d70ce53c6306)
    #11 0xffff969c777c  (/lib/aarch64-linux-gnu/libc.so.6+0x2777c) (BuildId: 0ebe1595b8271c41ccff2a40ccb7290647933748)
    #12 0xffff969c7854  (/lib/aarch64-linux-gnu/libc.so.6+0x27854) (BuildId: 0ebe1595b8271c41ccff2a40ccb7290647933748)
    #13 0xaaaad62cb8ac  (/targets/recursion-sequence+0x17b8ac) (BuildId: f8f2d70ce53c6306)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/targets/recursion-sequence+0x6868fc) (BuildId: f8f2d70ce53c6306)
==119387==ABORTING
Aborted
```

Snippet 4.7: ASAN report of one of the samples

The family of functions defined in `recursion-sys` accepts a pointer of pointers and an unsigned integer as the last two arguments. This a common pattern for passing by reference an array in C. The affected functions throw away the length of the array and do not check its value.

These functions are extern C functions and encapsulated in `unsafe` blocks. As such, data collections such as vectors or slices passed as arguments to the extern functions have to be converted to raw C pointers. If the collection is empty this could potentially generate a pointer to garbage data.

Since the functions do not check the length of the array represented by the pointer it assumes data has been properly initialized and reads the array without checking.

Impact These unchecked read operations result in undefined behavior and could be used as an attack vector by an attacker as part of a larger exploit.

The following functions are affected:

- `risc0_circuit_recursion_step_compute_accum`
- `risc0_circuit_recursion_step_verify_accum`
- `risc0_circuit_recursion_step_exec`
- `risc0_circuit_recursion_step_verify_mem`

Recommendation Implement a length check before the call to the inner function that contains the business logic. A possible implementation for `risc0_circuit_recursion_step_verify_mem` is as follows.

```
extern "C" uint32_t risc0_circuit_recursion_step_verify_mem(risc0_error* err,
                                                              void* ctx,
                                                              Callback callback,
                                                              size_t steps,
                                                              size_t cycle,
                                                              Fp** args_ptr,
                                                              size_t args_len) {
  return ffi_wrap<uint32_t>(err, 0, [&] {
      if (args_len < MIN_ARGS_LEN) throw "Unsufficient number of arguments";
    BridgeContext bridgeCtx{ctx, callback};
    return circuit::recursion::step_verify_mem(&bridgeCtx, bridgeCallback, steps,
    cycle, args_ptr)
        .asRaw();
  });
}
```

These exceptions would be captured by the exception handler in `ffi_wrap` and converted into a `risc0_error`, maintaining consistency with the current error handling logic.

Developer Response The developers have fixed the issue in the following PR.

4.1.10 V-RISC0-VUL-010: Missing Data Validation in Syscall Execution

Severity	Low	Commit	e6a2cb9
Type	Denial of Service	Status	Fixed
File(s)	rv32im/src/prove/emu/exec/mod.rs		
Location(s)	ecall_software		
Confirmed Fix At	https://github.com/risc0/risc0/pull/2713/		

The function `ecall_software`, shown below, is used to execute system calls like `getenv`, `read`, and `write`.

```
fn ecall_software(&mut self) -> Result<bool> {
      tracing::debug!("[{}] ecall_software", self.insn_cycles);
      println!("ecall software!");
      let into_guest_ptr = ByteAddr(self.load_register(REG_A0)?);
      println!("guest ptr: {:?}", into_guest_ptr);
      let into_guest_len = self.load_register(REG_A1)? as usize;
      if into_guest_len > 0 && !is_guest_memory(into_guest_ptr.0) {
          bail!("{into_guest_ptr:?} is an invalid guest address");
      }
      let name_ptr = self.load_guest_addr_from_register(REG_A2)?;
      let syscall_name = self.peek_string(name_ptr)?;
      let name_end = name_ptr + syscall_name.len();
      Self::check_guest_addr(name_end)?;
      tracing::trace!("ecall_software({syscall_name}, into_guest: {into_guest_len})")
    ;

      let chunks = align_up(into_guest_len, IO_CHUNK_WORDS) / IO_CHUNK_WORDS;

      let syscall = if let Some(syscall) = &self.pending.syscall {
          tracing::debug!("Replay syscall: {syscall:?}");
          syscall.clone()
      } else {
          let mut to_guest = vec![0u32; into_guest_len];

          let (a0, a1) = self
              .syscall_handler
              .syscall(&syscall_name, self, &mut to_guest)?;

          let syscall = SyscallRecord {
              to_guest,
              regs: (a0, a1),
          };
          self.pending.syscall = Some(syscall.clone());
          syscall
      };

      // The guest uses a null pointer to indicate that a transfer from host
      // to guest is not needed.
      if into_guest_len > 0 && !into_guest_ptr.is_null() {
          Self::check_guest_addr(into_guest_ptr + into_guest_len)?;
          self.store_region(into_guest_ptr, bytemuck::cast_slice(&syscall.to_guest))?
      }
```

```

        let (a0, a1) = syscall.regs;
        self.store_register(REG_A0, a0)?;
        self.store_register(REG_A1, a1)?;

        tracing::trace!("{syscall:08x?}");

        self.pending.cycles += chunks + 1; // syscallBody + syscallFini
        self.pending.pc = self.pc + WORD_SIZE;

        Ok(true)
}
```

Snippet 4.8: Snippet from `ecall_software()`

The function does the following:

1. Given a user provided `into_guest_len`, the function allocates a vector of length `into_guest_len` and assigns it to the `to_guest` field in `syscall`.
2. It then calls the syscall.
3. If the syscall succeeds, it checks whether `into_guest_ptr + into_guest_len` is a valid guest address.
4. If (3) succeeds it then writes the entire contents of `into_guest_ptr` into user memory.

There are two issues with this code:

1. The `to_guest` vector is allocated before checking that the address range fits within user memory. As such, a malicious user could allocate a `u32` vector of length `u32::MAX - 1`. Such allocations could pose a problem in the Bonzai prover depending on the system resources available.
2. The check `into_guest_ptr + into_guest_len` is not performed using safe math so the result could overflow if `into_guest_len` is sufficiently large. If the sum overflows, then the check will pass even though it shouldn't. As a result, the call `store_region` will store billions of bytes to memory one at a time.

Impact These missing checks could affect the Bonsai prover network as a malicious user could construct a program which generates syscalls that take a disproportionate amount of time to time before determining they fail.

Recommendation We recommend 1) using safe math to ensure the address range is valid, 2) Performing the allocation after the check.

Developer Response The developers have fixed the issue in this PR.

4.1.11 V-RISC0-VUL-011: Heap pointer overflow on large allocation

Severity	Low	**Commit**	e6a2cb9
Type	Arithmetic Overflow	**Status**	Fixed
File(s)	`zkvm/platfor/src/syscall.rs`		
Location(s)	sys_alloc_aligned		
Confirmed Fix At	https://github.com/risc0/risc0/pull/2778/		

The function `sys_alloc_aligned` is used to allocate new memory by returning a pointer to the latest memory on the heap and then updating the heap pointer according to the number of `bytes` requested as follows.

```
pub unsafe extern "C" fn sys_alloc_aligned(bytes: usize, align: usize) -> *mut u8 {
    ...

    // Pointer to next heap address to use, or 0 if the heap has not yet been
    // initialized.
    static mut HEAP_POS: usize = 0;

    // SAFETY: Single threaded, so nothing else can touch this while we're working.
    let mut heap_pos = unsafe { HEAP_POS };

        ...

    let ptr = heap_pos as *mut u8;
    heap_pos += bytes;

    // Check to make sure heap doesn't collide with SYSTEM memory.
    if crate::memory::SYSTEM.start() < heap_pos {
        const MSG: &[u8] = "Out of memory!".as_bytes();
        unsafe { sys_panic(MSG.as_ptr(), MSG.len()) };
    }

    unsafe { HEAP_POS = heap_pos };
    ptr
}
```

Snippet 4.9: Snippet from `sys_alloc_aligned()`

The addition `heap_pos += bytes` can overflow. Thus, if a large enough value for `bytes` is given such that the check `crate::memory::SYSTEM.start() < heap_pos` still passes, the heap pointer can be reset to already used values.

Impact A malicious user could use this to overwrite elements of the heap.

Recommendation Ensure overflow checks are enabled.

Developer Response The developers have addressed the issue by adding overflow checks.

4.1.12 V-RISC0-VUL-012: BabyBear construction functions should validate input

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Acknowledged
File(s)	risc0/core/src/field/baby_bear.rs		
Location(s)	multiple		
Confirmed Fix At	N/A		

The `risc0_core::field::baby_bear::Elem` struct provides the `new_raw()` function to create a baby bear element from a `u32` value that is already encoded in Montgomery form.

```
/// Create a new [BabyBear] from a Montgomery form representation
///
/// Requires that 'x' comes pre-encoded in Montgomery form.
pub const fn new_raw(x: u32) -> Self {
    Self(x)
}
```

Snippet 4.10: Definition of `new_raw()`

The correctness of many operations implemented within `risc0_core::field::baby_bear::Elem` depend on the property that all `Elem` values are modulo `P` (the baby bear prime) and encoded in Montgomery form.

For example, `Elem::add` is implemented by calling the following function on the `u32` values wrapped within the two operands:

```
fn add(lhs: u32, rhs: u32) -> u32 {
    let x = lhs.wrapping_add(rhs);
    if x >= P {
        x - P
    } else {
        x
    }
}
```

Snippet 4.11: Implementation of addition on the `u32` values backing the `Elem` struct

If `lhs = rhs = P+1`, then `x = 2P+2` so the `if` condition is true and the function returns `x-P = 2P+2-P = P+2` which is not a valid baby bear element.

Secondly, `Elem::from_u32_words()` provides the same functionality and likewise lacks input validation. This function is implemented for the `risc0_core::field::Elem` trait. It seems to the auditors that the intention of that function is to take raw input that is already "valid" and "reduced" for the specific implementation of the trait. In this case, the function documentation should state that values must be pre-encoded in Montgomery form.

Finally, `ExtElem::from_u32_words()` allows creating an extension field element from a slice of `u32`. However, it performs no validation on the input, instead directly wrapping each `u32` in `Elem`. This function is implemented for the `risc0_core::field::Elem` trait. See previous paragraph.

Impact These functions allow creating `risc0_core::field::baby_bear::Elem` instances which are not in Montgomery form reduced modulo the baby bear prime and performing operations such as `add` with such a value will produce incorrect results without producing an error.

Additionally, without the necessary checks on `ExtElem::from_u32_words()`, this function can be used to violate the assumption stated on `ExtElem::is_valid()`: "assume that if our first subelement is valid, the whole thing is valid."

Recommendation Add `assert!(x < P)` to `Elem::new_raw()` to express the documented requirement in code.

Add `assert!(x < P)` to `Elem::from_u32_words()` and document the requirement on the function.

`ExtElem::from_u32_words()` should use `Elem::new_raw(*word)` instead of `Elem(*word)`.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.13 V-RISC0-VUL-013: BabyBear Elem operations should validate input

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Acknowledged
File(s)	`risc0/core/src/field/baby_bear.rs`		
Location(s)	multiple		
Confirmed Fix At	N/A		

The `u32` value within the `Elem` parameter is directly passed to `decode()` without ensuring the `Elem` is valid. Occurs in the following locations:

1. Within `impl From<Elem> for u32`
2. Within `Elem::as_u32()`

Impact These functions produce the same result for `Elem::INVALID` and `Elem::new(1069547522)` which could cause uses of these functions to treat `INVALID` as a valid value.

Recommendation Add `x.ensure_valid()` to both functions.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.14 V-RISC0-VUL-014: More rounds in Poseidon hash function

Severity	Warning	**Commit**	a6159d9
Type	Maintainability	**Status**	Acknowledged
File(s)	`risc0/zkp/src/core/hash/poseidon/mod.rsrisc0/zkp/src/core/hash/poseidon/consts.rs`		
Location(s)			
Confirmed Fix At	N/A		

The Poseidon hash function is implemented with 8 full rounds and 21 partial rounds.

In the file `risc0/zkp/src/core/hash/poseidon/mod.rs`, it is claimed that the Poseidon hash function achieves 128-bit security. However, considering an interpolation attack, see here, the actual security can be estimated as: $d \log(d) \left(\log(d) + \log(p)\right) \log(\log(d))$,

where $d = t^{r-2}$, with $t = 7$ (the degree of the S-box) and r representing the total number of rounds. The following python code gives 91 bits of security.

```python

import math
r = 8 + 21
d = 7**(r - 2)
p = 15 * 2**27 + 1
log_d, log_p = math.log2(d), math.log2(p)
log_result = math.log2(d * log_d * (log_d + log_p) * math.log2(log_d))
print(log_result)
```

Impact This reduces the security of the hash function, and its consequent usage. In the latest commit, this Poseidon implementation is no longer used.

Recommendation Increase the number of rounds, or fix the number of security bits for this hash function.

Developer Response The developers have informed us that this implementation is being deprecated in favor of Poseidon2 and will be removed from the code base.

4.1.15 V-RISC0-VUL-015: Missing pre-condition check in align_up()

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Acknowledged
File(s)	`risc0/zkvm/platform/src/lib.rs`		
Location(s)	align_up()		
Confirmed Fix At	N/A		

The `align_up(addr,align)` function returns "the smallest `x` with alignment `align` so that `x >= addr`." This is an optimized version of `addr.next_multiple_of(align)` with the assumption that `align` is a power of 2.

```
pub const fn align_up(addr: usize, align: usize) -> usize {
    let mask = align - 1;
    (addr + mask) & !mask
}
```

Snippet 4.12: Implementation of `align_up()`

When `align` is not a power of 2, the function returns without panic but the returned value is not equivalent to `addr.next_multiple_of(align)`. This pre-condition is documented on the function but not checked with an assert statement.

The table below demonstrates values returned for various inputs.

Sample inputs and outputs

`addr`	`align`	`align_up(addr, align)`	`addr.next_multiple_of(align)`
294	5	298	295
295	5	299	295
296	5	296	300
297	5	297	300

Impact All uses of `align_up()` within the scope of this audit satisfy the pre-condition so there is no problem currently. However, if future additions to the audited code add a use of `align_up()` that does not satisfy the pre-condition, the function will return a meaningless value.

Recommendation Add the following assertions at the beginning of the `align_up()` function to ensure `align` is a power of 2:

1. `debug_assert!(align > 0)`
2. `debug_assert!(align & (align - 1) == 0)`

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.16 V-RISC0-VUL-016: send_recv_slice() is overly restrictive

Severity	Warning	**Commit**	a6159d9
Type	Logic Error	**Status**	Acknowledged
File(s)	`risc0/zkvm/src/guest/env.rs`		
Location(s)	send_recv_slice()		
Confirmed Fix At	N/A		

The zkvm guest provides the function `send_recv_slice()` to exchange plain old data with the host. The function makes two syscalls to the host with the first obtaining the size of the response data in bytes and the second containing the response data itself as an array of words.

```
pub fn send_recv_slice<T: Pod, U: Pod>(syscall_name: SyscallName, to_host: &[T]) -> &'static [U] {
  let syscall::Return(nbytes, _) = syscall(syscall_name, bytemuck::cast_slice(to_host), &mut []);
  let nwords = align_up(nbytes as usize, WORD_SIZE) / WORD_SIZE;
  let from_host_buf = unsafe { core::slice::from_raw_parts_mut(sys_alloc_words(nwords), nwords) };
  syscall(syscall_name, &[], from_host_buf);
  &bytemuck::cast_slice(from_host_buf)[..nbytes as usize / core::mem::size_of::<U>()]
}
```

Snippet 4.13: Implementation of `send_recv_slice()`

The return expression in this function uses `bytemuck::cast_slice(from_host_buf)` to reinterpret the `&[u32]` buffer as a slice `&[U]` and then uses `[..nbytes as usize / size_of::<U>()]` to truncate the final `U` if there were not enough bytes to construct a complete value of type `U`. Note, `size_of::<U>()` returns the size of a type in bytes.

According to `bytemuck::cast_slice()` documentation, the function will panic if the entire input slice cannot be split into a multiple of output type elements exactly. Therefore, this use of `cast_slice()` introduces the requirement that an integer `x` exists such that `nwords * WORD_SIZE = size_of::<U>() * x`, otherwise the `cast_slice()` will panic. Note, the constant `WORD_SIZE` is `4`.

In the following scenarios, the `send_recv_slice()` function successfully returns as many elements of type `U` that can be formed from the received data, truncating excess bytes:

1. Assume `size_of::<U>() = 8` and `nbytes = 5`. Thus `nwords = 2`, and 2*4=8 is a multiple of 8 so `cast_slice()` returns a slice with length 1. Finally that result is sliced as `[..5/8]` which returns an empty slice from the function.
2. Assume `size_of::<U>() = 8` and `nbytes = 13`. Thus `nwords = 4`, and 4*4=16 is a multiple of 8 so `cast_slice()` returns a slice with length 2. Finally that result is sliced as `[..13/8]` which returns a slice with length 1 from the function.

In the following scenarios, the `send_recv_slice()` function will panic instead of returning those elements of type `U` that can be formed from the received data:

1. Assume `size_of::<U>() = 8` and `nbytes = 12`. Thus `nwords = 3`, but 3*4=12 is not a multiple of 8 so `cast_slice()` panics and the function does not return a single element of type `U` as expected.

2. Assume `size_of::<U>() = 6` and `nbytes = 6`. Thus `nwords = 2`, but 2*4=8 is not a multiple of 6 so `cast_slice()` panics and the function does not return a single element of type `U` as expected.

Impact In certain scenarios where the number of bytes received from the host is not a multiple of `size_of::<U>()` the `send_recv_slice()` function will panic rather than returning as many elements of type `U` as possible.

Recommendation To handle all scenarios mentioned above, the return expression in the `send_recv_slice()` function should be modified so that the truncation occurs on the byte array and preserves the largest number of bytes equal to a multiple of `size_of::<U>()`. Replacing the return expression with the following would suffice.

```
let nitems = nbytes as usize / core::mem::size_of::<U>();
bytemuck::cast_slice(
    &bytemuck::cast_slice::<_, u8>(from_host_buf)[..nitems * core::mem::size_of::<U
    >()],
)
```

Snippet 4.14: Fixed return expression

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.17 V-RISC0-VUL-017: Goldilocks functions should detect INVALID input

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Fixed
File(s)	`risc0/core/src/field/goldilocks.rs`		
Location(s)	multiple		
Confirmed Fix At	`https://github.com/risc0/risc0/pull/2857/`		

The operations on `Elem` and `ExtElem` within `risc0/core/src/field/goldilocks.rs` must assert that their inputs are not `Self::INVALID` or else meaningless results may be computed without producing an error.

Calls to `ensure_valid()` should be added in the following locations:

1. On `self` and `rhs` in `Elem::add()`
2. On `self` and `rhs` in `Elem::add_assign()`
3. On `self` and `rhs` in `Elem::sub()`
4. On `self` and `rhs` in `Elem::sub_assign()`
5. On `self` and `rhs` in `Elem::mul()`
6. On `self` and `rhs` in `Elem::mul_assign()`
7. `PartialEq` is currently a derived trait for `Elem` and `ExtElem` but it should be implemented explicitly to add `ensure_valid()` checks (see the implementations of `PartialEq` in `risc0/core/src/field/baby_bear.rs`)

Impact These functions will produce meaningless results without producing an error if these parameters are `Self::INVALID`. When performing computations with `INVALID`, there is no guarantee that the results will even be within the range of the finite field.

Recommendation Add calls to `ensure_valid()` in all locations mentioned above.

Additionally, the auditors recommend changing the `debug_assert` within `ensure_valid()` to a release `assert` to prevent silent propagation of elements that are `INVALID` or otherwise outside the range of the finite field implementation.

Developer Response The developers have fixed the issue in the following PR.

4.1.18 V-RISC0-VUL-018: to_po2() is overly restrictive

Severity	Warning	**Commit**	a6159d9
Type	Logic Error	**Status**	Acknowledged
File(s)	`risc0/zkp/src/core/mod.rs`		
Location(s)	to_po2()		
Confirmed Fix At	N/A		

The `to_po2()` function in `risc0/zkp/src/core/mod.rs` computes the largest power of 2 (`po2`) such that `(1 << po2) <= x` for input `x`. Both the input and output of this function have type `usize` but an intermediate step in the implementation truncates the input to `u32` type. As a result, this can cause the following incorrect behavior for certain values larger than `u32::MAX`:

1. Values that truncate to 0 via the `as u32` cast cause panic with message "attempt to subtract with overflow" because the value 0 has 32 leading 0's.
2. Other values larger than `u32::MAX` produce a result, `po2`, that is not the *largest* value such that `(1 << po2) <= x`. One example is `x = 3911141260033417769` which returns the value `31`, but should return `61`.

Impact The function can panic or return unexpected values on inputs larger than `u32::MAX`. This is not documented and possibly not the intended behavior.

Recommendation Implement the function as `(usize::BITS - 1 - x.leading_zeros()) as usize` to handle the full range of input parameters to the function.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.19 V-RISC0-VUL-019: hash_raw_data_slice() in guest does not match documentation

Severity	Warning	**Commit**	a6159d9
Type	Logic Error	**Status**	Acknowledged
File(s)	`risc0/zkp/src/core/hash/sha/guest.rs`		
Location(s)	hash_raw_data_slice()		
Confirmed Fix At	N/A		

The `hash_raw_data_slice()` function in `risc0/zkp/src/core/hash/sha/guest.rs` is implemented for the `risc0_zkp::core::hash::sha::Sha256` trait. The documentation for this function in the trait definition says the function must "Generate a hash from a slice of anything that can be represented as a slice of *bytes*." However, the implementation in `risc0/zkp/src/core/hash/sha/guest.rs` enforces a stronger restriction by converting it to a slice of `u32` *words*.

```
fn hash_raw_data_slice<T: bytemuck::NoUninit>(data: &[T]) -> Self::DigestPtr {
    let digest = alloc_uninit_digest();
    let words: &[u32] = bytemuck::cast_slice(data);
    update_u32(digest, &SHA256_INIT, words, WithoutTrailer);
    // Now that digest is initialized, we can convert it to a reference.
    unsafe { &mut *digest }
}
```

Snippet 4.15: Definition of `hash_raw_data_slice()` in `risc0/zkp/src/core/hash/sha/guest.rs`

Impact All audited uses of this function have `Field::Elem` or `Field::ExtElem` as input elements and all audited implementations of those are (transparently) `u32`, `u64`, or array of one of those which means it will fit in the word boundary. However, future modification to the code base may cause unexpected panics since this implementation does not align with the trait documentation.

Recommendation Change the type of the `words` local to `&[u8]` and replace the call to `update_u32()` with `update_u8()`.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.20 V-RISC0-VUL-020: insufficient size checks in poly_interpolate()

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Acknowledged
File(s)	`risc0/zkp/src/core/poly.rs`		
Location(s)	poly_interpolate()		
Confirmed Fix At	N/A		

In the `poly_interpolate()` function from `risc0/zkp/src/core/poly.rs`, additional checks using the `size` parameter are needed in two places:

1. Before computing the output, the `poly_interpolate()` function clears the output slice by looping over the entire output slice and assigning the value `E::ZERO` at every position. All other computations in this function use only a prefix of the slice parameters with length `size`.
2. Secondly, the parameter `x` is not guaranteed to have at least `size` values. If there are fewer than `size` values in `x`, the expression `x.iter().enumerate().take(size)` would yield only the elements in `x` leaving a suffix of the `ft` array filled with 0 values. This scenario would not cause a panic but would end up computing a value that is probably not the intended result.
3. Thirdly, it should be checked that all the elements of `x: &[E]`, are distinct.

Impact

1. Clearing the output beyond the intended range may cause unexpected results to be computed by the callers of this function. However, within the scope of the current audit, this function is called only by `risc0_zkp::prove::prover::Prover::finalize()` and in that caller, these excess zero assignments have no impact other than performance.
2. If the parameter `x` has fewer than `size` values, the function will compute a value for `out` that is incorrect or meaningless without producing an error.
3. If $x[i] = x[j]$, no valid interpolated polynomial should exist. However, `poly_interpolate()` will still produce a polynomial that evaluates to $f(x[i]) + f(x[j])$.

Recommendation

1. Change the loop that clears the output so that it iterates over `out[0..size]` instead of `*out`.
2. Add the assertion `assert!(x.len() >= size)`. The auditors additionally recommend adding `debug_assert!(out.len() >= size)` and `debug_assert!(fx.len() >= size)` to make all constraints on the input explicit. It is safe to use debug-only assertions for these two cases because both `out` and `fx` have accesses later in the function that would panic if the assertions were false.
3. Add an assertion to check that the elements of x are distinct.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.21 V-RISC0-VUL-021: Goldilocks construction functions should validate input

Severity	Warning	**Commit**	a6159d9
Type	Data Validation	**Status**	Fixed
File(s)	`risc0/core/src/field/goldilocks.rs`		
Location(s)	multiple		
Confirmed Fix At	`https://github.com/risc0/risc0/pull/2857/`		

1. The `GoldilocksElem` struct implements the `from_u32_words()` function from the `risc0_core::field::Elem` trait. It seems to the auditors that the intention of that function is to take raw input that is already "valid" and "reduced" for the specific implementation of the trait. However, the `GoldilocksElem::from_u32_words()` function does not validate that the input is "valid" and "reduced" so it allows creating a `GoldilocksElem` whose value is not within the goldilocks field.
2. There are multiple functions that allow constructing `GoldilocksExtElem` using `INVALID` for one or more of the inner `GoldilocksElem`. These are `new()`, `from_fp()`, `from_subfield()`, `from_subelems()`, `impl From<[Elem; EXT_SIZE]> for ExtElem`, and `impl From<Elem> for ExtElem`. There is no need to construct `GoldilocksExtElem` containing `INVALID` because the `GoldilocksExtElem::INVALID` constant already exists. Furthermore, `GoldilocksExtElem` constructed via the functions mentioned above may pass the `is_valid()` check because that function checks strict equality with `GoldilocksExtElem::INVALID` that has both component `Elem` as the `INVALID` instance.

Impact These functions allow creating `GoldilocksElem` instances whose value is not within the goldilocks field. The correctness of many operations implemented within `risc0/core/src/field/goldilocks.rs` depend on the property that all `GoldilocksElem` instances are less than `P` (the goldilocks prime). They also allow creating `GoldilocksExtElem` that contain `INVALID` but pass the `is_valid()` check.

Recommendation

1. Make the following changes:
   a) Add `assert!(val < P)` after the assignment `let val: u64 = ...` in `GoldilocksElem::from_u32_words()`
   b) Document the requirement mentioned above on the `from_u32_words()` function in `risc0/core/src/field/mod.rs`.
2. Make the following changes:
   a) Add calls to `ensure_valid()` on both parameters in `ExtElem::new()`
   b) All of the other functions listed above must use `ExtElem::new()` to create an instance rather than doing so directly
   c) Update the definition of `GoldilocksExtElem::is_valid()` to check only the first index for `INVALID` (like the BabyBear implementation does; because there should now be no way to create a `GoldilocksExtElem` instance containing `Elem::INVALID` other than the `ExtElem::INVALID` instance itself).

Developer Response The developers fixed the issue in the following PR.

4.1.22 V-RISC0-VUL-022: Segment permissions are ignored in ELF decoding

Severity	Warning	**Commit**	e6a2cb9
Type	Data Validation	**Status**	Acknowledged
File(s)	`binfmt/src/elf.rs`		
Location(s)	load_elf		
Confirmed Fix At	N/A		

The first step in the ZKVM usage workflow is extracting an *image* from an ELF. The image describes the initial memory layout for the program and the entrypoint for execution. The decoding essentially transcribes all `PT_LOAD` segments into their internal image data structure. The `PT_LOAD` segments describe the physical and virtual addresses of the program, the size of the segment in the ELF file, alignment details, etc. It may contain executable code, data, or both, depending on its attributes.

The `PT_LOAD` segments have a `p_flags` attribute that describes access permission . These flags can take on values such as `PF_X` (executable), `PF_W` (writable), and `PF_R` (readable), and the combination of these flags determines the segment's permissions. The Risc0 decoder ignores these flags when decoding, implicitly allowing all segments to be writable. As such, the executions of images within the Risc-V ZKVM can differ from Risc-V programs on other platforms. For example, suppose we have a Risc-V program P where developers intentionally marked a segment S in memory as read-only and suppose under input i, P attempts to write to S. Then the Risc Zero ZKVM this write will succeed when it should fail under common execution environments like Linux.

This deviation from common execution environments is not documented anywhere and could affect end-users as it would allow verifiable proofs to be created of infeasible executions.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.23 V-RISC0-VUL-023: Multiple values have the same digest

Severity	Warning	**Commit**	e6a2cb9
Type	Cryptographic Vulnerabilit	**Status**	Acknowledged
File(s)	`risc0/binfmt/src/hash.rs`		
Location(s)	See description.		
Confirmed Fix At	N/A		

The value `Digest::ZERO` is used as a special constant value for digests computed from certain values, including `None`, empty arrays, and empty tagged iterations.

```
pub fn tagged_iter<S: Sha256>(
    tag: &str,
    iter: impl DoubleEndedIterator<Item = impl Borrow<Digest>>,
) -> Digest {
    iter.rfold(Digest::ZERO, |list_digest, elem| {
        tagged_list_cons::<S>(tag, elem.borrow(), &list_digest)
    })
}
```

Snippet 4.16: Implementation of `tagged_iter()`

Impact Having multiple values with the same digest could lead to collisions in unsafe ways. For example, the `tagged_iter` function above will produce the same `Digest::Zero` digest for all tagged iterations, regardless of the tag provided.

Recommendation Make sure all values have a (highly probably) unique digest.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.24 V-RISC0-VUL-024: Missing Overflow Check in load_elf

Severity	Warning	**Commit**	e6a2cb9
Type	Data Validation	**Status**	Acknowledged
File(s)	`binfmt/src/elf.rs`		
Location(s)	load_elf		
Confirmed Fix At	N/A		

The `load_elf` function loads a RISC-V ELF file into memory. To load a segment, the loader examines the `offset` field in the segment header which specifies the offset in the ELF file where the segment is located. The loader then reads the segment from the file at the offset as shown in the following code snippet

```
for i in (0..mem_size).step_by(WORD_SIZE) {
    let addr = vaddr.checked_add(i).context("Invalid segment vaddr")?;
    if addr >= max_mem {
        bail!("Address [0x{addr:08x}] exceeds maximum address for guest programs [0
    x{max_mem:08x}]");
    }
    if i >= file_size {
        // Past the file size, all zeros.
        image.insert(addr, 0);
    } else {
        let mut word = 0;
        // Don't read past the end of the file.
        let len = core::cmp::min(file_size - i, WORD_SIZE as u32);
        for j in 0..len {
            let offset = (offset + i + j) as usize;
            let byte = input.get(offset).context("Invalid segment offset")?;
            word |= (*byte as u32) << (j * 8);
        }
        image.insert(addr, word);
    }
}
```

Snippet 4.17: Snippet from load_elf()

The issue is that the statement `let offset = (offset + i + j) as usize` does not check whether the sums overflow. If compiled with the `release` flag, if `offset` is sufficiently large, the sum could overflow, causing the calculated offset to point somewhere at the beginning of the file. This could allow an invalid segment to be parsed into an image without throwing an error.

Impact `load_elf` should succeed if and only if the binary is a well formed Risc-V ELF. Violating this specification can have unexpected behavior for downstream users.

Recommendation Most likely the impact of this vulnerability will be minor but we recommend performing overflow checks via `checked_add` just to be safe.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.25 V-RISC0-VUL-025: Invalid post state for paused

Severity	Warning	**Commit**	e6a2cb9
Type	Logic Error	**Status**	Acknowledged
File(s)	zkvm/src/receipt_claim.rs		
Location(s)	ok, paused		
Confirmed Fix At	N/A		

The functions `ok` and `paused` are both used to construct a receipt claim - their only difference is that `ok` produces one with an exist code `Halted` while `paused` produces one with an exit code `Paused`. Both set the following post state value.

```
post: MaybePruned::Value(SystemState {
    pc: 0,
    merkle_root: Digest::ZERO,
})
```

Snippet 4.18: Post state value from `ok` and `paused`

This post state is just a placeholder and does not represent the actual state after execution.

Impact Depending on how these post state values are used, it could lead to false conclusions about the state at various points.

Recommendation Set the post state to the actual state after execution.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.26 V-RISC0-VUL-026: Possible poisoning of ZKR_REGISTRY mutex

Severity	Warning	Commit	e6a2cb9
Type	Denial of Service	Status	Acknowledged
File(s)	zkvm/src/hsot/recursion/prove/mod.rs		
Location(s)	get_registered_zkr		
Confirmed Fix At	N/A		

The ZKR_REGISTRY is a mapping that is used to store a function that fetches a recursion program from it's control ID.

```
pub(crate) static ZKR_REGISTRY: Mutex<ZkrRegistry> = Mutex::new(BTreeMap::new());
```

Snippet 4.19: Definition of ZKR_REGISTRY

The value is defined with a mutex which is locked in both register_zkr and get_registered_zkr. In get_registered_zkr, the following logic is used to fetch the associated function and call it to get the actual recursion program for a given control ID.

```
pub fn get_registered_zkr(control_id: &Digest) -> Result<Program> {
    let registry = ZKR_REGISTRY.lock().unwrap();
    registry
        .get(control_id)
        .map(|f| f())
        .unwrap_or_else(|| bail!("Control id {control_id} unregistered"))
}
```

Snippet 4.20: Implementation of get_registered_zkr

If the call to f() panics, the mutex on ZKR_REGISTRY could be "poisoned", meaning any other thread that tries to call ZKR_REGISTRY.lock().unwrap() will panic.

Impact If one thread intentionally or unintentionally panicked and poisoned the mutex, it could potentially block all other threads from accessing the ZKR_REGISTRY.

Recommendation Alter the code to avoid the possibility of panicking when holding the mutex or remove the possibility of issue by keeping the protocol single threaded.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.27 V-RISC0-VUL-027: Undocumented Security Assumptions

Severity	Warning	**Commit**	e6a2cb9
Type	Documentation	**Status**	Acknowledged
File(s)	See issue description		
Location(s)			
Confirmed Fix At	N/A		

The RiscZero VM consists of several components that all serve different purposes. For example:

1. ELF Decoder (Decodes an ELF into an image)
2. Risc-V execution engine (Executes the RISC-V image and builds segments)
3. Risc-V Circuits (used to generate the prover and verifier)
4. Syscall interface (host and guest components)
5. Host (the environment used to execute Risc-V programs)
6. Guest (the program being executed)

Each of these components have different security assumptions which also depends on how they are used. For example the security assumption of the host and guest code differ depending on whether it is being used in *local proving mode* or within *Bonsai*.

As such, the important usage scenarios for each of these components need to be documented along with the high level security properties that need to be preserved. For example, within the Bonsai proving mode, it needs to be clear:

1. What is the trust model between users and Risc Zero? Is it a whitelisted model or can anybody easily access the prover?
2. How are users being charged for using the prover?

Currently these are not explicitly documented and this makes it difficult to 1) determine if a piece of code is a bug, 2) if it is a bug, correctly assessing the severity of it. We recommend documenting the threat model and security assumptions associated with the major components of the zkvm. This would help both users of the VM as well as other auditors.

Impact Undocumented security assumptions make it difficult for users and auditors to determine whether code is secure.

Recommendation We recommend adding explicit threat models and security assumptions for the major components of the VM.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.28 V-RISC0-VUL-028: Low cycle cost for reading from user memory on system calls

Severity	Warning	**Commit**	e6a2cb9
Type	Logic Error	**Status**	Acknowledged
File(s)	`zkvm/src/host/server/exec/syscall.rs`		
Location(s)	See issue description.		
Confirmed Fix At	N/A		

For many system calls, bytes are passed in from the guest directly and are read in via the `load_region` function pictured below.

```
fn load_region(&mut self, addr: ByteAddr, size: u32) -> Result<Vec<u8>> {
    let mut region = Vec::new();
    for i in 0..size {
        region.push(self.load_u8(addr + i)?);
    }
    Ok(region)
}
```

Snippet 4.21: Implementation of `load_region`

The `load_u8` function ends up calling the `peek_u8` function which in turn calls the pager `peek` function. This function loads 0 bytes into a page if that page has not yet been written to the page table or otherwise loads the relevant data from the page cache. In both cases, no cycle updates are recorded.

Impact A malicious user could make system calls that read the entirety of guest memory (there are checks that ensure reads from guest memory are valid guest addresses). The cycles associated with these reads will not be accounted for.

Recommendation Record cycles associated with system call loads.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.29 V-RISC0-VUL-029: User-triggered panic in syscall handling

Severity	Warning	**Commit**	e6a2cb9
Type	Logic Error	**Status**	Acknowledged
File(s)	`zkvm/src/host/server/exec/syscall.rs`		
Location(s)	syscall		
Confirmed Fix At	N/A		

For read system calls (i.e., `SysRead`), the handling for the system call includes the following check.

```
assert!(
    nbytes >= to_guest.len() * WORD_SIZE,
    "Word-aligned read buffer must be fully filled"
);
```

Snippet 4.22: Snippet from `syscall()` for `SysRead`

The assertion checks that the number of bytes provided to read (which are read from a register) must be greater than or equal to the length of the guest buffer. A user can provide any value for `nbytes`, causing the execution to panic.

Impact Allowing the user to cause the handler code to panic is less than ideal, as it could complicate the ability to properly track execution costs and recover from errors.

Recommendation For most system call errors, an `Error` is returned rather than panicking. We suggest doing that here as well.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.30 V-RISC0-VUL-030: Denial of service via child processes

Severity	Warning	**Commit**	e6a2cb9
Type	Logic Error	**Status**	Acknowledged
File(s)	`zkvm/src/host/server/exec/syscall/fork.rs`		
Location(s)	run		
Confirmed Fix At	N/A		

The function `run` for child processes resulting from a fork has the following definition.

```
pub fn run(&mut self) -> Result<()> {
    let mut emu = Emulator::new();
    let mut cycles = 1;
    while !self.exit {
        emu.step(self)?;
        cycles += 1;
    }
    tracing::info!("unconstrained cycles: {cycles}");
    Ok(())
}
```

Snippet 4.23: Implementation of `run`

As one can see, it creates a new emulator and steps on that emulator until the exit flag is flipped. Notably, the cycles associated with this execution are tracked locally but (1) are not used to limit the iterations of the loop and (2) are not recorded anywhere outside of this function.

Impact An attacker could exploit this to cause a remote prover to hang on a program execution indefinitely.

Recommendation Add in some limits on child process executions.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.31 V-RISC0-VUL-031: Incorrect attribute application causes linking error when link time DCE is disabled

Severity	Info	**Commit**	a6159d9
Type	Maintainability	**Status**	Acknowledged
File(s)	`risc0/zkvm/platform/src/syscall.rs`		
Location(s)	sys_argv , sys_alloc_words		
Confirmed Fix At	N/A		

Functions `sys_argv` and `sys_alloc_words` are guarded by a feature flag named `export-syscalls` along with the function `sys_alloc_aligned`. This set of functions is intended to be exported to clients of the VM, hence the feature guard.

A misconfiguration in the attributes, as seen in the snippet below, can cause compilation errors under certain configurations. The former functions are not properly guarded and are not removed during compilation. The latter is properly guarded and removed if the feature is off. In normal circumstances the former functions are considered dead code since other components that do not require the functions are not going to have the feature enabled. The default LTO configuration will remove the functions in a DCE pass.

If the compiler is configured to not run the DCE pass then the former functions will remain in the output binary but not the latter, causing a linking error.

```
#[cfg_attr(feature = "export-syscalls", no_mangle)]
pub unsafe extern "C" fn sys_argv(
    out_words: *mut u32,
    out_nwords: usize,
    arg_index: usize,
) -> usize {
    let Return(a0, _) = syscall_1(nr::SYS_ARGV, out_words, out_nwords, arg_index as
    u32);
    a0 as usize
}

#[cfg_attr(feature = "export-syscalls", no_mangle)]
pub extern "C" fn sys_alloc_words(nwords: usize) -> *mut u32 {
    unsafe { sys_alloc_aligned(WORD_SIZE * nwords, WORD_SIZE) as *mut u32 }
}

#[cfg(feature = "export-syscalls")]
#[no_mangle]
pub unsafe extern "C" fn sys_alloc_aligned(bytes: usize, align: usize) -> *mut u8 {
```

Snippet 4.24: Snippet from `syscall.rs`

The attribute `cfg_attr` in the upper functions will apply the `no_mangle` attribute if the `export-syscalls`feature is on. `cfg` on the other hand will include the function if the feature is on.

The following compilation command will trigger the link error.

```
RUSTFLAGS="-C codegen-units=1 -C link-dead-code" cargo +nightly build --release -v
```

Output:

= note: Undefined symbols for architecture arm64: "_sys_alloc_aligned", referenced from: risc0_zkvm_platform::syscall::sys_alloc_words::h076a4834bc7c4f52 in librisc0_zkvm_platform-f9e7bddc9c948a84.rlib3 ld: symbol(s) not found for architecture arm64 clang: error: linker command failed with exit code 1 (use -v to see invocation)

error: could not compile `risc0-zkvm-methods` (build script) due to 1 previous error

Impact We do not identify a direct security risk from this misconfiguration. It can lead to cryptic linking errors and unintended behavior.

Recommendation Replace the following line

```
#[cfg_attr(feature = "export-syscalls", no_mangle)]
```

With the following two lines

```
#[cfg(feature = "export-syscalls")]
#[no_mangle]
```

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.32 V-RISC0-VUL-032: Code Quality

Severity	Info	Commit	a6159d9
Type	Maintainability	Status	Acknowledged
File(s)	multiple		
Location(s)	multiple		
Confirmed Fix At	N/A		

1. `risc0/core/src/field/mod.rs`
   a) The definition of the `Elem` trait lists redundant supertraits: `core::clone::Clone` and `Clone` are the same, likewise `core::marker::Copy` and `Copy` are the same.

```
pub trait Elem:
    ops::Mul<Output = Self>
    + ops::MulAssign
    + ops::Add<Output = Self>
    + ops::AddAssign
    + ops::Neg
    + ops::Sub<Output = Self>
    + ops::SubAssign
    + cmp::PartialEq
    + cmp::Eq
    + core::clone::Clone
    + core::marker::Copy
    + Sized
    + bytemuck::NoUninit
    + bytemuck::CheckedBitPattern
    + core::default::Default
    + Clone
    + Copy
    + Send
    + Sync
    + Debug
    + 'static
{
...
```

Snippet 4.25: Definition of `risc0_core::field::Elem`

   b) Once fixes are applied for V-RISC0-VUL-012 and V-RISC0-VUL-021 , the functions `ensure_reduced()` and `is_reduced()` and be removed. The former is currently unused both will no longer be necessary once all instances of `Elem` are "reduced" by construction.

2. `risc0/core/src/field/baby_bear.rs`
   a) The function `Elem::random()` performs the cast `P as u64` rather than using the pre-defined constant `P_U64`.
   b) The function `to_u32_words()` is defined as `Vec::<u32>::from([self.0])` but the auditors recommend `[self.0].to_vec()` instead because it is more idiomatic Rust and may have slightly better performance.
   c) The field element and extension field element types are defined along with aliases as follows:

```
pub struct Elem(u32);
pub type BabyBearElem = Elem;

pub struct ExtElem([Elem; EXT_SIZE]);
pub type BabyBearExtElem = ExtElem;
```

This auditors recommend directly using the names `BabyBearElem` and `BabyBearExtElem` for the struct definitions and removing the aliases. This avoids confusion with the `Elem` and `ExtElem` traits defined in `risc0/core/src/field/mod.rs`.

d) The `from_fp()` function performs the same operation as `from_subfield()` with the only differences being that it allows `Elem::INVALID` which is unlikely to be a valid use case anyway and that it uses `Elem::new(0)` instead of `Elem::ZERO` which may have worse performance. The `from_fp()` function should be removed and its uses replaces with `from_subfield()`.

e) The fields `ExtElem::ZERO` and `ExtElem::ONE` are initialized via helper functions that call a series of other functions. It would be equivalent and more straightforward to implement them as follows:

```
const ZERO: Self = ExtElem([Elem::ZERO, Elem::ZERO, Elem::ZERO, Elem::ZERO
    ]);
const ONE: Self = ExtElem([Elem::ONE, Elem::ZERO, Elem::ZERO, Elem::ZERO]);
```

Snippet 4.26: Recommended implementation of `ExtElem::ZERO` and `ExtElem::ONE`

f) The `ExtElem::const_part()` function is not called anywhere so it should be removed.

3. `risc0/core/src/field/goldilocks.rs`

   a) The field element and extension field element types are defined along with aliases as follows:

```
pub struct Elem(u64);
pub type GoldilocksElem = Elem;

pub struct ExtElem([Elem; EXT_SIZE]);
pub type GoldilocksExtElem = ExtElem;
```

This auditors recommend directly using the names `GoldilocksElem` and `GoldilocksExtElem` for the struct definitions and removing the aliases. This avoids confusion with the `Elem` and `ExtElem` traits defined in `risc0/core/src/field/mod.rs`.

   b) The fields `ExtElem::INVALID`, `ExtElem::ZERO`, and `ExtElem::ONE` are initialized via helper functions that call a series of other functions. It would be equivalent and more straightforward to implement them as follows:

```
const INVALID: Self = ExtElem([Elem::INVALID, Elem::INVALID]);
const ZERO: Self = ExtElem([Elem::ZERO, Elem::ZERO]);
const ONE: Self = ExtElem([Elem::ONE, Elem::ZERO]);
```

Snippet 4.27: Recommended implementation of `GoldilocksExtElem` constants

   c) The `ExtElem::const_part()` function is not called anywhere so it should be removed.

   d) The `from_fp()` function performs the same operation as `from_subfield()` with the only difference being that it uses `Elem::new(0)` instead of `Elem::ZERO` which may have worse performance. The `from_fp()` function should be removed and its uses replaces with `from_subfield()`.

e) The function `to_u32_words()` is defined as Vec::<u32>::from([self.0 as u32, (self.0 >> 32) as u32]) but the auditors recommend `[self.0 as u32, (self.0 >> 32) as u32].to_vec()` instead because it is more idiomatic Rust and may have slightly better performance.

f) Uses of `Elem::new(0)` should be replaced with the constant `Elem::ZERO`. This occurs in `from_fp()`, `from_u64()`, and `const from_u64()`.

g) The `ExtElem::from_u32_words()` function exhibits three different behaviors when given an input slice whose length is not 4.

   i. When input length is >5, it panics with a clear and helpful error message "Extra elements passed to create element in extension field"
   ii. When input length equals 5, there is no error, it simply ignores the final value in the slice
   iii. When input length <4, it panics with "called `Option::unwrap()` on a `None` value"

   The auditors recommend adding an explicit assertion with a clear error message for length <4 and adding an assertion that fails when length equals 5.

4. `risc0/groth16/src/seal_to_json.rs`

   a) Use of magic constant in `to_json()`. The line `pos += 8` should be `pos += DIGEST_WORDS`

5. The `poly_eval()` function is defined in `risc0/zkp/src/verify/mod.rs` but is already implemented in `risc0/zkp/src/core/poly.rs`.
6. In `poly_interpolate()` from `risc0/zkp/src/core/poly.rs` there is a parameter named `x` and two local loop variables named `x`, shadowing the parameter within the scopes of those local variables.
7. In `risc0/zkp/src/core/mod.rs`, the `Random` trait and its implementation are unused. They should be removed.
8. There are two functions implemented with the following block of code (with different variable names changed to `x` here) that is more complicated than necessary.

```
match bytemuck::try_cast(data) {
    Ok(x) => x,
    Err(PodCastError::TargetAlignmentGreaterAndInputNotAligned) => {
        bytemuck::pod_read_unaligned(&data)
    }
    Err(e) => unreachable!(/*omitted*/),
}
```

Snippet 4.28: Overly complicated block of code

The documentation of `bytemuck::try_cast()` specifically states "alignment isn't a factor" (see `https://docs.rs/bytemuck/1.16.1/bytemuck/fn.try_cast.html`) and a careful examination of the implementation reveals that the `TargetAlignmentGreaterAndInputNotAligned` error kind cannot occur. Thus, it would be sufficient to implement both of these function as simply `bytemuck::cast(data)`. The functions are:

- `impl From<[u8; DIGEST_BYTES]> for Digest` in `risc0/zkp/src/core/digest.rs`
- `impl From<[u8; BLOCK_BYTES]> for Block` in `risc0/zkp/src/core/hash/sha/mod.rs`

9. There are two locations that use `bytemuck::checked::cast_slice()` to cast `&[u32]` into `&[BabyBearElem]`. However, `BabyBearElem::from_u32_slice()` provides the same

functionality and should be used instead to hide implementation details. The functions are:

- `decode_receipt_claim_from_seal()` in `risc0/zkvm/src/receipt/segment.rs`
- `verify_integrity_with_context()` in `risc0/zkvm/src/receipt/succinct.rs`

10. `risc0/zkp/src/core/hash/sha/guest.rs`
    a) The function `update_u32()` is not used outside of this file so the `pub(crate)` access modifier should be removed.
11. `risc0/circuit/recursion/src/zkr.rs`
    a) The `get_all_zkrs()` function could use `with_context()` like the `get_zkr()` function does. Additionally, consider extracting common functionality of these two functions into a helper.
12. `risc0/zkp/src/core/ntt.rs`
    a) The `bit_rev_32()` function duplicates `u32::reverse_bits()`. It should be removed and all uses replaced with `u32::reverse_bits()`.
13. `risc0/zkp/src/hal/cpu.rs`
    a) The `batch_expand_into_evaluate_ntt()` has a parameter named `expand_bits` and also defines a local within the first bracketed code block named `expand_bits`.
14. `risc0/zkp/src/core/hash/sha/cpu.rs`
    a) The `hash_bytes()` function converts the `&[u8]` slice into `[u32; DIGEST_WORDS]` to use as parameter to `Digest::from()`. However, there is another implementation of `Digest::from()` that takes `[u8; DIGEST_BYTES]` so the simpler implementation `<[u8; DIGEST_BYTES]>::try_from(digest.as_slice()).unwrap()` can be used to create a `Digest` instance.
15. In `risc0/zkp/src/core/hash/sha/cpu.rs`, the `hash_raw_data_slice()`, `compress()`, and `compress_slice()` functions use `word.to_be()` to ensure the words in `state` (which come from a `Digest` instance and thus have big-endian byte order) are converted to native byte order before calling `sha2::compress256()`. However, this conversion should use `u32::from_be(*word)` which is equivalent but makes it clear that the conversion is from big-endian to native byte order.
16. In `risc0/zkp/src/core/digest.rs`, the `Digest::from_bytes()` function uses `u32::from_be(word)` to convert the given `u32` word from native byte order to big-endian byte order. However, this conversion should use `word.to_be()` which is equivalent but makes it clear that the conversion is from native to big-endian byte order.

Impact The issues mentioned make the code harder to understand and maintain.

Recommendation

1. Remove the redundant traits and unnecessary functions.
2. Apply the recommendations mentioned.
3. Apply the recommendations mentioned.

4. Apply the recommendations mentioned.
5. Remove the `poly_eval()` function from `risc0/zkp/src/verify/mod.rs` and replace all uses with the one from `risc0/zkp/src/core/poly.rs`
6. Rename the two loop variables that are named `x` to a name that does not shadow any other variable in scope.
7. Remove the unused trait and implementation.
8. Replace the body of both functions mentioned with simply `bytemuck::cast(data)`.
9. Replace the use of `bytemuck::checked::cast_slice()` with `BabyBearElem::from_u32_slice()` in both locations.
10. Apply the recommendation mentioned.
11. Apply the recommendations mentioned.
12. Apply the recommendation mentioned.
13. Rename either the parameter or the local variable so that no shadowing of names occurs.
14. Apply the recommendation mentioned.
15. In each function, replace the first use of `word.to_be()` with `u32::from_be(*word)`.
16. Replace `u32::from_be(word)` with `word.to_be()`.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.33 V-RISC0-VUL-033: Documentation

Severity	Info	**Commit**	a6159d9
Type	Maintainability	**Status**	Acknowledged
File(s)	multiple		
Location(s)	multiple		
Confirmed Fix At	N/A		

1. In `baby_bear.rs` the constant `const INVALID: Self = Elem(0xffffffff)` and in `goldilock.rs` the constant `const INVALID: Self = Elem(0xffffffff_ffffffff)` are used to define the `is_valid()` function that distinguishes between initialized and uninitialized data. This is not documented on the `is_valid()` function.
2. In `risc0/zkp/src/core/mod.rs`, the `to_po2()` function documentation is inaccurate.

```
/// For x = (1 << po2), given x, find po2.
```

Snippet 4.29: Documentation of `to_po2()`

The stated equality only holds when the input `x` is a power of 2. However, the examples show that function is intended to handle input that is not a power of 2, and the implementation matches that intention.

3. In `risc0/zkp/src/core/mod.rs`
   a) In `to_po2()`, the auditors recommend adding examples for the minimum and maximum inputs for the function:
      i. `assert_eq!(to_po2(1), 0); // min input`
      ii. `assert_eq!(to_po2(usize::MAX), 63); // max input`
   b) In `log2_ceil()`, the auditors recommend adding examples for the minimum and maximum inputs for the function:
      i. `assert_eq!(log2_ceil(0), 0); // min input`
      ii. `assert_eq!(log2_ceil(1<<63), 63); // max input`
4. In `risc0/zkp/src/prove/write_iop.rs`, the `write_pod_slice()` function will panic if the input is not aligned to `u32` word boundary and an exact multiple of words in length because the call to `bytemuck::cast_slice()` in that function is producing `&[u32]`. All audited uses of this function use `&[Digest]` as input so this panic will not occur but this restriction is not documented on the function.
5. The `Digest` struct defined in `risc0/zkp/src/core/digest.rs` is composed of `u32` words. According to commit 3224c3c and uses of `Digest` such as in `risc0/zkp/src/core/hash/sha/cpu.rs`, the auditors conclude that these words should always be in big-endian byte order. However, that requirement is not clearly document on the `Digest` struct.
6. In `risc0/zkp/src/core/hash/sha/cpu.rs`, the `compress()` function converts the compressed `state` variable from native byte order into big-endian byte order before storing it in a `Digest` instance. However, the documentation states the conversion is the opposite direction.
7. In `risc0/zkp/src/core/hash/sha/rng.rs`, the `ShaRng::new()` function is documented as "Create a new [ShaRng] from a given [Sha256]" but there is no `Sha256` parameter here. Instead, it only uses the implementation from `risc0/zkp/src/core/hash/sha/cpu.rs`.

Recommendation

1. Add documentation on `is_valid()`
2. The function documentation should be modified to "Returns the largest `po2` such that `(1 << po2) <= x`." The auditors also recommend adding the assertion `debug_assert_ne!(x, 0)`. The function will already panic when the input is `0` but this assertion makes it clear that failure is intended in that case.
3. Apply the recommendations mentioned.
4. Document the restriction that input to the `write_pod_slice()` function must be aligned to the `u32` word boundary and an exact multiple of `u32` words in length.
5. Add documentation to the `Digest` struct stating that the `u32` words should always be in big-endian byte order.
6. Change comment on the final loop in `compress()` to "Convert the native byte order result to big-endian."
7. Documentation should be corrected.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.34 V-RISC0-VUL-034: Performance

Severity	Info	**Commit**	a6159d9
Type	Gas Optimization	**Status**	Acknowledged
File(s)	multiple		
Location(s)	multiple		
Confirmed Fix At	N/A		

1. The current polynomial evaluation in `risc0/zkp/src/core/poly.rs` is inefficient, requiring $O(n^2)$ multiplications and $O(n)$ additions.

```
pub fn poly_eval<E: ExtElem>(coeffs: &[E], x: E) -> E {
    let mut mul = E::ONE;
    let mut tot = E::ZERO;
    for coeff in coeffs {
        tot += *coeff * mul;
        mul *= x;
    }
    tot
}
```

Snippet 4.30: Definition of `poly_eval()` in `poly.rs`

Horner's method computes $P(x) = (((a_n x + a_{n-1})\, x + a_{n-2}) \ldots)\, x + a_1)\, x + a_0$, reducing the complexity to $O(n)$ for both multiplications and additions. A significant portion of computation in FRI relies on polynomial evaluation, so improving its efficiency will enhance overall performance.

2. The current polynomial interpolation in `risc0/zkp/src/core/poly.rs` is $O(d^2)$ but could be performed in $O(d\ log\ d)$ by using FFTs or recursion.
3. The `ExtElem::pow()` functions in `risc0/core/src/field/baby_bear.rs` and `risc0/core/src/field/goldilocks.rs` should use squaring instead of self-multiplication in the line `x *= x`. Direct squaring is faster in field extensions, compared to using self-multiplication for squaring. This will speed-up the exponentiation in `pow()`. Additionally, in `goldilocks.rs`, using `ExtElem::from(1)` is suboptimal compared to using the pre-defined constant `ONE`.
4. `Elem::pow()` in `risc0/core/src/field/mod.rs` could have a slight performance improvement if implemented like the Rust lib version: d3a393932e/library/core/src/num/int_macros.rs (L2731-L2751). Note that the overflow issue mentioned in the Rust lib is not relevant here since multiplication is modulo `P`.
5. The `map_pow()` function defined in `risc0/core/src/field/mod.rs` uses `copied()` on the iterator but there is no need to perform the copy, just dereference the values.
6. There are several redundant `is_valid()` checks in `risc0/core/src/field/baby_bear.rs`:

   a) `ExtElem::from_subfield()` uses `ensure_valid()` on an `Elem` instance that is then passed into `ExtElem::from([Elem; EXT_SIZE])` which also calls `ensure_valid()` on each `Elem` instance. The `ensure_valid()` in `ExtElem::from_subfield()` can be removed.

   b) `ExtElem::from_subelems()` uses `ensure_valid()` on each `Elem` instance and then they are all passed into `ExtElem::from([Elem; EXT_SIZE])` which also calls `ensure_valid()` on each `Elem` instance. All `ensure_valid()` calls in `ExtElem::from_subelems()` can be removed.

7. The ExtElem::from_u32() function risc0/core/src/field/baby_bear.rs has multiple calls to Elem::new(0) rather than using the pre-defined constant Elem::ZERO.
8. In risc0/core/src/field/baby_bear.rs, the impl From<u64> for Elem implementation performs the x%P operation and then the call to Elem::new() repeats that operation. To avoid the redundant modulus operation, this function can be implemented as Elem::new_raw(encode((x % P_U64) as u32)).
9. The add_input_digest() function in risc0/zkvm/src/host/recursion/prove/mod.rs uses .copied() on the &digest.as_words() iterator but it is not necessary. Remove the .copied() call.
10. In zirgen/circuit/verify/poly.cpp, the poly_eval() function should use Horner's method (as in issue 1 above).
11. In risc0/zkp/src/core/hash/sha/mod.rs, there are multiple uses of bytemuck APIs where minor performance improvements could be achieved.

    a) In impl AsRef<[u8; BLOCK_BYTES]> for Block, using self as the parameter instead of &self.0 has equivalent behavior and avoids the field access and reference operation.
    b) In impl AsMut<[u8; BLOCK_BYTES]> for Block, using self as the parameter instead of &mut self.0 has equivalent behavior and avoids the field access and reference operation.
    c) In Block::as_bytes(), implementing the function as bytemuck::cast_ref::<_, [u8; BLOCK_BYTES]>(self) is equivalent since self is a fixed-size array and the cast_ref() function should have better performance than cast_slice().
    d) In Block::as_mut_bytes(), since self is a fixed-size array, implementing the function as bytemuck::cast_mut::<_, [u8; BLOCK_BYTES]>(self) is equivalent and the cast_mut() function should have better performance than cast_slice_mut().
    e) In Block::as_half_blocks(), since self is a fixed-size array, using bytemuck::cast_ref::<_, [Digest; 2]>(self) is equivalent and the cast_ref() function should have better performance than cast_slice(). The entire function can be implemented more simply as:

```
let [half_block1, half_block2] = bytemuck::cast_ref::<_, [Digest; 2]>(self)
    ;
(half_block1, half_block2)
```

Snippet 4.31: Recommended implementation of Block::as_half_blocks()

12. In risc0/zkp/src/core/digest.rs, there are multiple uses of bytemuck APIs where minor performance improvements could be achieved.

    a) In impl AsRef<[u8; DIGEST_BYTES]> for Digest, using self as the parameter instead of &self.0 has equivalent behavior and avoids the field access and reference operation.
    b) In impl AsMut<[u8; DIGEST_BYTES]> for Digest, using self as the parameter instead of &mut self.0 has equivalent behavior and avoids the field access and reference operation.
    c) In Digest::as_bytes(), implementing the function as bytemuck::cast_ref::<_, [u8; DIGEST_BYTES]>(self) is equivalent since self is a fixed-size array and the cast_ref() function should have better performance than

cast_slice().

d) In Digest::as_mut_bytes(), since self is a fixed-size array, implementing the function as bytemuck::cast_mut::<_, [u8; DIGEST_BYTES]>(self) is equivalent and the cast_mut() function should have better performance than cast_slice_mut().

13. risc0/zkp/src/core/hash/sha/cpu.rs

   a) The hash_raw_data_slice() function makes a call to bytemuck::cast_slice_mut() that is not necessary. The result type from this call is &[u8] but the parameter type of the call is already &[u8] so no conversion is necessary. The call to bytemuck::cast_slice_mut() should be removed, just using its parameter in its place.

   b) The compress() function makes multiple calls to block.as_mut_slice() within a loop. This call should be moved outside the loop and assigned to a new local that is used in place of the calls within the loop.

14. In risc0/zkp/src/core/ntt.rs, the interpolate_ntt() function has a loop at the end that multiplies the values in io by norm.

```
for x in io.iter_mut().take(size) {
    *x = *x * norm;
}
```

Snippet 4.32: Snippet from interpolate_ntt()

The take(size) is not necessary because size == io.len().

15. In risc0/zkp/src/core/hash/sha/rust_crypto.rs, the finalize_variable_core() function performs multiple calls to b.as_half_blocks() and retrieves a different indexed field from the tuple for each call. Multiple calls return the same result so only one call to b.as_half_blocks() is necessary per code block.

Recommendation

1. Use Horner's method for poly_eval(), as follows:

```
pub fn poly_eval<E: ExtElem>(coeffs: &[E], x: E) -> E {
    let mut tot = E::ZERO;
    for &coeff in coeffs.iter().rev() {
        tot = tot * x + coeff;
    }
    tot
}
```

2. Refer to these lecture notes, https://people.csail.mit.edu/madhu/ST12/scribe/lect06.pdf, page 6-3 for a more efficient implementation.
3. Refer to this paper, https://eprint.iacr.org/2006/471.pdf, using section 3 (for quadratic extension) in goldilocks.rs and section 5 (for quartic extension) in baby_bear.rs. Additionally, in goldilocks.rs replace ExtElem::from(1) with Self::ONE.
4. Implement like the Rust lib version.
5. Remove copied() and use * as necessary.
6. Apply the recommendations mentioned.
7. Uses of Elem::new(0) should be replaced with Elem::ZERO.
8. Apply the recommendation mentioned.

9. Apply the recommendation mentioned.
10. Apply the recommendation mentioned.
11. Apply the recommendations mentioned.
12. Apply the recommendations mentioned.
13. Apply the recommendations mentioned.
14. Remove `take(size)`
15. In both branches of the match statement, add the assignment `let (h0, h1) = b.as_half_blocks();` and then use the `h*` locals directly instead of making multiple calls to `b.as_half_blocks()`.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.35 V-RISC0-VUL-035: Deprecated dependencies

Severity	Info	**Commit**	a6159d9
Type	Maintainability	**Status**	Acknowledged
File(s)	`Cargo.toml`		
Location(s)			
Confirmed Fix At	N/A		

Running `cargo audit` reveals that the risc0 project uses versions of two dependencies that have been yanked from crates.io:

```
Crate:     bytemuck
Version:   1.16.1
Warning:   yanked

Crate:     bytes
Version:   1.6.0
Warning:   yanked
```

Snippet 4.33: Snippet from `cargo audit` output

Impact Although no reason can be specified when yanking a crate from crates.io, that auditors assume that some bug or vulnerability exists in the crate that prompted its developers to yank it.

Recommendation Upgrade `bytemuck` to version `1.16.2` or newer, per `https://crates.io/crates/bytemuck`.

Upgrade `bytes` to version `1.6.1` or newer, per `https://crates.io/crates/bytes/versions`.

Add `cargo audit` as a regular step in your build process.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.36 V-RISC0-VUL-036: bit_reverse() panics on trivial case

Severity	Info	**Commit**	a6159d9
Type	Logic Error	**Status**	Acknowledged
File(s)	`risc0/zkp/src/core/ntt.rs`		
Location(s)	bit_reverse()		
Confirmed Fix At	N/A		

The `bit_reverse()` function permutes the values in the input slice, swapping values at `i` and `i'`, computed by reversing the bits of `i`. The function computes the reversed array index by reversing the index as a 32 bit integer and then right-shifting the result right down to the number of bits required to represent the max index in the input array.

```
let rev_idx = (bit_rev_32(i as u32) >> (32 - n)) as usize;
```

Snippet 4.34: Snippet from `bit_reverse()`

When the input slice contains a single element, `n=0` which causes a panic "attempt to shift right with overflow." However, the operation performed by the function is trivial when the input slice contains a single element. No modification of the slice is needed in that case.

Impact Unexpected panic.

Recommendation Add `if n > 0 {..}` around the loop.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.37 V-RISC0-VUL-037: Typos, unused program constructs, and other small fixes

Severity	Info	Commit	e6a2cb9
Type	Maintainability	Status	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description The following program constructs are unused:

- `zkvm/src/receipt_claim.rs`:
  - function `paused()`
- `zkvm/src/serde/err.rs`:
  - enum `Error::SerializeBufferFull`

The following additional small changes should be considered:

- `zkvm/src/receipt_claim.rs`:
  - `impl Digest for Unknown`:
    - function `digest<S: Sha256>()`:
      - Use the `unreachable!` macro to panic on truly unreachable code.
- `binfmt/src/elf.rs`:
  - function `load_elf()`:
    - The check `segments.len() > 256` is used to check that there are at most 256 ELF segments - we suggest defining a constant explicitly to capture the "magic number" `256`.
- `zkvm/platform/src/lib.rs`:
  - function `align_up`:
    - This implementation relies on the fact that the `align` argument is a power of `2` - if it is not, the implementation is not correct. Because this is a necessary assumption for correctness, it should be added as an explicit assertion.
- `zkvm/platform/src/sycall.rs`:
  - const `USER` and `MACHINE` are not used.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.38 V-RISC0-VUL-038: Undocumented Assumption on Deserialization Could Drop Data

Severity	Info	**Commit**	e6a2cb9
Type	Maintainability	**Status**	Acknowledged
File(s)	`zkvm/src/serde/deserializer.rs`		
Location(s)	read_padded_bytes		
Confirmed Fix At	N/A		

The function `read_padded_bytes()` is used to read individual bytes from an array of `u32` values. The implementation of this function is shown below.

```
fn read_padded_bytes(&mut self, out: &mut [u8]) -> Result<()> {
    let bytes: &[u8] = bytemuck::cast_slice(self);
    if out.len() > bytes.len() {
        Err(Error::DeserializeUnexpectedEnd)
    } else {
        out.clone_from_slice(&bytes[..out.len()]);
        (_, *self) = self.split_at(align_up(out.len(), WORD_SIZE) / WORD_SIZE);
        Ok(())
    }
}
```

Snippet 4.35: Implementation of `read_padded_bytes()`

The line `(_, *self) = self.split_at(align_up(out.len(), WORD_SIZE) / WORD_SIZE);` is intended to remove the first `out.len()` bytes from the array, aligned up until they closest word-aligned byte *after* `out.len()`.

Impact If the length of `out.len()` is not word-aligned, it means that some bytes will be deleted from the current array of `u32` values that are never written to the output.

Recommendation Based on the places in the code where this function is used, we believe the behavior described above is intended. We recommend documentation this intended behavior along with the function so that mistakes are not made with future use of this function.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.39 V-RISC0-VUL-039: Overflow Leads to Lost Data on Serialization

Severity	Info	Commit	e6a2cb9
Type	Data Validation	Status	Acknowledged
File(s)	zkvm/src/serde/serializer.rs		
Location(s)	serialize_str, serialize_bytes		
Confirmed Fix At	N/A		

The function `serialize_str()` is used to serialize a string as an array of words as follows.

```rust
fn serialize_str(self, v: &str) -> Result<()> {
    let bytes = v.as_bytes();
    self.serialize_u32(bytes.len() as u32)?;
    self.stream.write_padded_bytes(bytes)
}
```

Snippet 4.36: Implementation of `serialize_str()`

The function serializes the length of the string first in a single world and then writes the actual string data as bytes after that. The potential issue here is that it is possible that the expression `bytes.len()` can overflow a `u32`, and thus the conversion for the length of the string could lead to unexpected behavior.

The same issue holds for `serialize_bytes()`.

Impact This unchecked overflow could cause strange behavior on serialization where a string is not serialized as expected. This could lead to people thinking they are passing one input into the guest program but actually passing something very different.

Recommendation Add a check for overflow that panics in the case that the length of the string is greater than the max `u32`.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.40 V-RISC0-VUL-040: MachineContext crashes if preflight traces are empty

Severity	Info	**Commit**	e6a2cb9
Type	Data Validation	**Status**	Acknowledged
File(s)	`circuit/rv32im/src/prove/engine/witgen.rs, circuit/rv32im/src/prove/engine/machine.rs`		
Location(s)	MachineContext::get_cycle		
Confirmed Fix At	N/A		

The method `MachineContext::get_cycle` will read from an empty array if the preflight trace is empty and the `StepMode` is not equal to `StepMode::SeqForward`. There is no validation of the minimum number of traces required for the witness generation.

`WitnessGenerator::new` will call `MachineContext::inject_exec_backs` if the `StepMode` is not `StepMode::SeqForward`. Previously, the instance of `MachineContext` is initialized with the `trace` parameter of `WitnessGenerator::new`. As part of its internal logic `MachineContext::inject_exec_backs` will call `MachineContext::get_cycle` which will read from the `cycles` field of either `trace.pre` or `trace.body`. Panicking if either of these arrays is empty.

Recommendation Add validation that the number of elements in the traces is at least the minimum number of elements required and fail if not enough values are provided.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

4.1.41 V-RISC0-VUL-041: If the power of two value is too low the witness generator crashes

Severity	Info	**Commit**	e6a2cb9
Type	Data Validation	**Status**	Acknowledged
File(s)	`circuit/rv32im/src/prove/engine/witgen.rs, circuit/rv32im/src/prove/engine/loader.rs`		
Location(s)	Loader::add_cycle, Loader::body		
Confirmed Fix At	N/A		

The witness generation logic doesn't seem to be checking that the power of two supplied is large enough to create enough cycles for the circuit.

The `po2` argument in `WitnessGenerator::new` affects the number of maximum cycles the generator will use. This value is used to compute `let steps = 1 << po2;` and that variable is then used as the `max_cycles` parameter of `Loader::new`. This value affects two computations that will panic if the value is too low and the code was compile in debug mode. If the code was compiled in release mode the second location will not crash and will silently overflow.

Power of 2 less than 11 The `Loader` struct defines an array field named `ctrl` of size `max_cycles * 16`. If the power of two is less than 11 it crashes with an overflow writing into `ctrl` inside the loop in `Loader::bytes_setup` when adding a cycle with `Loader::add_cycle`. The affected function is as follows.

```
fn add_cycle(&mut self, row: CtrlCycle) {
    self.ctrl[self.cycle] = BabyBearElem::new(self.cycle as u32);
    for i in 1..row.0.len() {
        // Out-of-bounds write happens here
        self.ctrl[self.max_cycles * i + self.cycle] = row.0[i];
    }
    self.cycle += 1;
}
```

On the last iteration of the loop the function attempts to write into the index equal to the size of the `ctrl` array. For example, if the power of two is 4 then `ctrl` will be 256 elements long and in the last iteration it would attempt to write into index 256.

Given max_cycles = 16 i = 15 cycle = 16 Then max_cycles * i + cycle = 16 * 15 + 16 = 256

The number of cycles required to complete the initialization step is 1592, which in binary is `0110 0011 1000`. Any power of two less than 11 will be too small to fit all the required cycles.

Power of 2 equal to 11 If the value is equal to 11 then the loader crashes in `Loader::body` trying to compute the number of cycles dedicated to the body. The implementation of the function is as follows.

```
fn body(&mut self) {
        // Values if po2 = 11
        //        -1544 = 2048            - 1592        - 6            - 1994
    let body_cycles = self.max_cycles - self.cycle - FINI_CYCLES - ZK_CYCLES;
```

```
    tracing::debug!("[{}] BODY: {body_cycles}", self.cycle);
    for _ in 0..body_cycles {
            // Will cause another out-of-bounds crash here
        self.add_cycle(CtrlCycle::body());
    }
}
```

The `body_cycles` variable is unsigned, causing an underflow. This causes a panic in debug mode. However this kind of overflow errors are not checked at runtime if the code is compiled in release mode for performance reasons. In release mode the value will be computed normally. Since `body_cycles` is a `usize` this will set its value to `18446744073709550072` in a 64-bit machine.

The following snippet shows the result of the computation. A link to a Rust playground with the code is provided below.

```
#[allow(arithmetic_overflow)]
fn main() {
    let x: usize = 2048 - 1592 - 6 - 1994;
    println!("{}", x);
}
```

This large value will then be used to add cycles, causing a similar crash to the first since `ctrl` won't have enough elements.

Impact If an attacker can control the value of `po2` when requesting a proof they may be able to crash the system by providing a low enough value that triggers the panics described above.

Recommendation Add validation to the witness generation logic that fails the generation if the power of two value passed as argument is not large enough to fill the necessary cycles.

Developer Response RISC Zero has acknowledged this warning and will opportunistically resolve this issue in development sprints during throughout 2025 calendar year.

5 Picus

5.1 Overview

This section describes how the Veridise analysts used a new version of Picus to check the determinism of the V2 circuits.

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

Veridise's new version of Picus takes as input circuits expressed in its own DSL called the Picus Constraint Language (PCL for short). As such, the V2 circuits could not directly be provided as input as they are expressed in their own language called Zirgen. To work around this, the Veridise analysts wrote a transpiler to convert circuits written in Zirgen to circuits in PCL. One key challenge in performing this conversion was identifying the output signals for each circuit. Veridise analysts worked with RISC Zero developers to determine which signals were outputs as opposed to intermediate signals.

5.4 Results Summary

The V2 Circuit consists of 123 sub-circuits, all of which were translated into PCL. Picus was able to successfully prove 117/123 (95%) were deterministic found underconstrained issues in the remaining 6 (V-RISC0-VUL-001, V-RISC0-VUL-002, V-RISC0-VUL-003, V-RISC0-VUL-004, V-RISC0-VUL-005, V-RISC0-VUL-006). All 6 issues were acknowledged by the RISC Zero developers and have been fixed (additionally confirmed using Picus).

6 Fuzz Testing

6.1 Methodology

There were several components of the RISC Zero zkVM that the security analysts believed were suitable for fuzz testing including the ELF-binary decoding, the instruction decoding, image execution, and prover's FFI component. In particular, for the ELF parsing, instruction decoding, and image execution we performed differential fuzzing to make sure RISC Zero's ZKVM conformed to the RISC-V semantics. For the FFI component, the analysts fuzzed for common vulnerabilities such as use-after-free, buffer overflows, etc. The details of the fuzzing approach, along with the fuzzers used, can be found in the fuzzing specs in this section.

6.2 Properties Fuzzed

Table 6.1 describes the fuzz-tested invariants. The second column describes the invariant informally in English, and the third shows the total amount of compute time spent fuzzing this property. The last column indicates the number of bugs identified when fuzzing the invariant.

The Veridise team devoted a total of 384 compute-hours to fuzzing this protocol, identifying a total of 1 bug (V-RISC0-VUL-009) which was acknowledged and fixed by the developers.

Table 6.1: Invariants Fuzzed.

Specification	Invariant	Minutes Fuzzed	Bugs Found
V-RISC0-SPEC-001	ELF-Parsing Crash Detection	4320	0
V-RISC0-SPEC-002	FFI CVD Fuzzing	10080	1
V-RISC0-SPEC-003	Instruction Decoding Differential Fuzzing	4320	0
V-RISC0-SPEC-004	Instruction Execution	4320	0

6.3 Detailed Description of Fuzzed Specifications

6.3.1 V-RISC0-SPEC-001: ELF-Parsing Crash Detection

Minutes Fuzzed	4320	Bugs Found	0

Description The target of these campaign is the module that parses ELF files. The goal was to trigger unexpected crashes by feeding the harness malformed ELF files. Any error that is purposefully handled is ignored.

Harness The entrypoint of the harness is the `risc0_binfmt::Program::load_elf((input: &[u8], max_mem: u32) -> Result<Program>` method and the following harness was defined

```
fuzz!(|data| {
    if data.len() < size_of::<u32>() {
        return;
    }

    let max_mem = u32::from_le_bytes([data[0], data[1], data[2], data[3]]);
    let input = &data[4..];

    let _ = Program::load_elf(input, max_mem);
});
```

This harness is compatible with both Libfuzzer and AFL.

Campaign This campaign has two steps: 1) corpus generation with a generational fuzzer, and 2) coverage guided fuzzing with Libfuzzer and AFL in different configurations.

Phase 1 In this phase, Veridise analysts generated a corpus of ELF files mutated with targeted mutations designed for the ELF file format. They then used the generational fuzzer Melkor to generate 12000 mutated samples from different types of ELF files (executables, libraries, etc)

This corpus was then minimized using the harness above with AFL's cmin, keeping any potential crashing input as a crash.

Phase 2 With the minimized corpus, the target was fuzzed for 3 days using the following combinations of fuzzers and sanitizers:

1. 4 instances of AFL w/ no sanitizers.
2. 4 instances of Libfuzzer w/ AddressSanitizer enabled.
3. 4 instances of Libfuzzer wi/ LeakSanitizer enabled.

This phase generated 1576 out-of-memory crashes out of which 0 could be replicated in a vacuum.

6.3.2 V-RISC0-SPEC-002: FFI CVD Fuzzing

Minutes Fuzzed	10080	Bugs Found	1

The FFI functions exposed by `rv32im-sys` and `recursion-sys` were fuzzed independently and in random sequences.

Description The crates `rv32im-sys` and `recursion-sys` expose a series of functions implemented in C++ to the rest of the Rust codebase. To fuzz them two strategies were used:

- For each function generate a random input and call the function with it.
- Generate a sequence of function calls that share some random initial data and call them in order.

The aim of the first strategy is to look for issues that affect the function on isolation while the aim for the second strategy is to look for issues that may happen as a consequence of several functions interaction with each other.

The following functions were targeted:

- `risc0_circuit_recursion_poly_fp`
- `risc0_circuit_recursion_step_compute_accum`
- `risc0_circuit_recursion_step_exec`
- `risc0_circuit_recursion_step_verify_accum`
- `risc0_circuit_recursion_step_verify_bytes`
- `risc0_circuit_recursion_step_verify_mem`
- `risc0_circuit_rv32im_calc_prefix_products`
- `risc0_circuit_rv32im_poly_fp`
- `risc0_circuit_rv32im_step_compute_accum`
- `risc0_circuit_rv32im_step_verify_accum`

6.3.3 V-RISC0-SPEC-003: Instruction Decoding Differential Fuzzing

Minutes Fuzzed	4320	Bugs Found	0

Description The goal of this campaign was twofold; first, find unexpected crashes during the decoding of RISC-V instructions, and second, perform differential testing against an oracle on the correctness of the decoding.

Harness The entrypoints for this target are the following methods in the `risc0_circuit_rv32im::prove::emu::rv32im` module:

- `DecodedInstruction::new(insn: u32) -> DecodedInstruction`
- `FastDecodeTable::lookup(&self, decoded: &DecodedInstruction) -> Instruction`

A special double-purpose ALF-based harness was created for this campaign. This harness takes two functions (`input` and `oracle`) that return the same type and compares the results, crashing if they are not structurally equal. The comparison can be deactivated at runtime by setting the `NO_ORACLE=1` environment variable. This reconfigures the harness to run only the `input` function and discards the results. For more information about this harness refer to `fuzzing_support/src/oracle/mod.rs` in the provided tarball.

Veridise analysts chose Capstone as the oracle.

To perform the comparison, the output of both components was transformed into a list of summaries. These summaries are defined as follows:

```
type OpSet = HashSet<u32>;

struct InsSummary {
    id: u32,      // The instruction ID as defined by Capstone
    regs: OpSet,  // The set of registers used by the instruction
    imms: OpSet,  // The set of immediate values defined by the structure
}
```

This summary is compared via structural equivalence i.e, two instructions are equal if they have the same opcode, operate on the same set of registers, and define the same immediate values.

To enable this comparison a conversion function $(DecodedInstruction, Instruction) \mapsto InsSummary$ was implemented that extracted the data of the instruction. The `id` of the summary was extracted from the `InstKind` of the `Instruction` since it is an almost 1:1 mapping with the capstone defined `id`. The summarization logic for registers and immediate values was based on The RISC-V Instruction Set Manual.

Campaign This campaign was performed in two phases, first with the oracle disabled, and second performing differential fuzzing against the oracle. The goal of the first phase was to generate a corpus of files that cover a substantial amount of code and to search for unexpected crashes. The second phase's goal is to find inputs that give different outputs between the input and oracle functions.

Phase 1 The first step of phase 1 was to generate an initial corpus. A fresh dataset of 10000 random inputs of 1000 bytes each was generated and then minimized with `cmin`. This minimization yielded a corpus of 12 files.

The minimized corpus was then fuzzed for 24 hours with the configuration below. No crashes were found during the fuzzing and only one more corpus sample was found.

Phase 2 The second phase enables the oracle and runs the fuzzers in differential mode. Uses the corpus as left by the previous phase.

This phase was run for 48 hours (not complete at the time of writing) and reported 65 (could be more) unique crashes.

Fuzzer configuration Both phases of the campaign performed fuzzing using the following combination of fuzzers and sanitizers

1. 9 instances of AFL w/ no sanitizers.
2. 5 instances of AFL w/ AddressSanitizer enabled.

Triage process After the execution of phase 2 the fuzzers found a total of 784 crashes. This total was minimized removing duplicates by content and then minimized by coverage using AFL's `cmin` tool. Each crash in the resulting output was individually minimized with AFL's `tmin`.

6.3.4 V-RISC0-SPEC-004: Instruction Execution

Minutes Fuzzed	4320	Bugs Found	0

As an extension of the Instruction decoding fuzzing campaign the instruction execution was compared against an oracle. For this task Unicorn, a QEMU based library, was selected.

Description The instruction execution logic in `rv32im` was exercised using a sequence of random bytes as input. The same input was executed in Unicorn and the results compared.

Glossary

Fiat-Shamir A well-known method for converting interactive proofs to non-interactive ones [2]. See `https://en.wikipedia.org/wiki/FiatâĂŞShamir_heuristic` to learn more. 4

RISC-V RISC-V is a computer architecture that stands for "Reduced Instruction Set Computer version 5". See `https://en.wikipedia.org/wiki/RISC-V` for more information. 1

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. 73

SMT Satisfiability Modulo Theories. 4

SNARK Stands for Succinct Non-Interactive Argument of Knowledge. See `https://en.wikipedia.org/wiki/Non-interactive_zero-knowledge_proof` for an overview.. 1

STARK Stands for Scalable Transparent Argument of Knowledge. See `https://starkware.co/wp-content/uploads/2022/05/STARK-paper.pdf` for more details.. 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. 73

ZK Circuit zero-knowledge circuit. 1

Bibliography

[1] Shankara Pailoor et al. ‘Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs’. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023) (cited on pages 1, 4).

[2] Amos Fiat and Adi Shamir. ‘How To Prove Yourself: Practical Solutions to Identification and Signature Problems’. In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194 (cited on page 73).