Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

RISC Zero Bigint2 Precompile

Veridise Inc.

Mar 25, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Shankara Pailoor
Alp Bassa
Ben Mariano
Daniel Dominguez

► **Contact Us:**

contact@veridise.com

► **Version History:**

Mar. 25, 2025 V1

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 4

3 Security Assessment Goals and Scope 5
- 3.1 Security Assessment Goals . . . 5
- 3.2 Security Assessment Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 6

4 Vulnerability Report 7
- 4.1 Detailed Description of Issues . . . 8
  - 4.1.1 V-RISC0_R2-VUL-001: No Documentation on Bigint2 . . . 8
  - 4.1.2 V-RISC0_R2-VUL-002: Incorrect terms added to ensure positivity . . . 9
  - 4.1.3 V-RISC0_R2-VUL-003: Magic Constants . . . 11

Glossary 13

1 Executive Summary

From Dec 2, 2024 to Dec 27, 2024, RISC Zero engaged Veridise to conduct a follow up security assessment of their ZKVM. The previous audit performed by Veridise covered most of their ZKVM implementation including parts of the prover, verifier, recursion circuits, default host implementation, and their new arithmetization of the RISC-V CPU in their V2 circuit DSL. The follow up engagement was intended to review a new implementation of the RISC-V big integer precompile (referred to as Bigint2). Veridise conducted the assessment over 8 person-weeks, with 2 security analysts reviewing the project over 4 weeks. The auditors reviewed the Bigint2 implementation on the repository `https://github.com/risc0/zirgen` on commit `c60423`.

Bigint2 Summary. Most ZKVMs utilize precompiles (co-processors) to make certain types of computation more efficient to both prove and verify. For example, given a complex hash function H, one could use a ZKVM to prove that $H(x) = y$ for given input x and output y by coding the computation in a language like Rust, compiling the program to RISC-V, and then generating a proof. However, the proof generation and verification times for such programs can be prohibitively expensive as they will require a lot of cycles to execute. To make such computation accessible to users, ZKVM developers write *precompile circuits* which express such complex computation directly in a circuit and make that computation accessible to end users via the system call interface.

The RISC Zero developers wrote a new precompile circuit (in their V1 eDSL) that allows users to perform arbitrary precision arithmetic (commonly referred to as big integer arithmetic). Such computation is especially useful for clients that are performing cryptographic computations like elliptic curve arithmetic or RSA cryptography as they typically require integers that are much larger than most machine integers (256-bit or 4096-bit integers).

At a high level, RISC Zero allows users to construct big integers of any bitwidth that is a multiple of 128. In particular, every big integer of bitwidth $128 * n$ can be expressed as follows: $\sum_{i=0}^{n-1} c_i * 256^{16*i}$. Each c_i in the representation is referred to as a chunk and is a 128-bit integer. It is important to note that each chunk represents a value that cannot be expressed in the native field used by the ZKVM (BabyBear). As such, each chunk c_i is represented as a list of bytes $[a_0, \ldots, a_{15}]$ such $c_i = \sum_{j=0}^{15} a_j * 256^j$ and each $a_j \in [0, 256)$.

The Bigint2 precompile exposes micro instructions for performing arithmetic operations over bigints. The precompile maintains a "current" chunk which is expressed as an array of 16 bytes along with 3 virtual registers: `poly`, `term`, `total`. Before describing the operations and what these registers do, it is important to note that the precompile treats each chunk $[a_0, \ldots, a_{15}]$ as a degree-15 *polynomial* $p(x) = \sum_{i=0}^{15} a_i x^i$. Note that p has the nice property that $p(256)$ equals the represented integer. Thus multiplication and addition of these chunks is reduced to polynomial addition and multiplication. In more detail, `poly` stores a big integer under construction, `total` is an accumulator register that keeps track of the running value of a sequence of multiplications and additions over big integers, and `term` stores a big integer that should be multiplied with `poly`. In addition to these registers, the precompile exposes 3 different memory operations over bigints:

1. `MemRead` - Read a chunk from memory into the current chunk.
2. `MemWrite` - Write the current chunk to memory.
3. `MemNop` - Nop (Don't do anything).

and 7 different arithmetic operations over bigints:

1. `kNop` - Nop.
2. `KOpShift` - Given a chunk c representing a polynomial $\sum_{i=0}^{15} a_i x^{16*k+i}$, the shift operator multiplies c by x^{16} generating the polynomial $\sum_{i=0}^{15} a_i x^{16*(k+1)+i}$. Intuitively this operation is used to perform computation with an inner chunk of a bigint that uses multiple chunks.
3. `kOpSetTerm` - Intuitively, this is used to move the value in `poly` to `term`.
4. `kOpAddTot` - Essentially, this operations multiplies `poly` and `term` and adds it to the current value in `total` and stores the result back into `total`.
5. `kOpCarry1`, `kOpCarry2` - The bigint operations can result in the coefficients growing larger than a byte. These operations are used to propagate carry values and normalize the resulting bigint representation.
6. `kOpEqz` - Asserts that the normalized value in `total` is equal to zero.

Such micro instructions might be challenging for end users to use directly, so the Zirgen compiler provides high level macro instructions which multiply, divide, and add bigints and will compile those instructions into several Bigint2 micro instructions.

It is important to note that the Bigint2 precompile checks the correctness of these operations using the Schwartz-Zippel Lemma. For instance, to check that $p = r$ where p and r are polynomials of low degree, then for a randomly sampled point x in the field, $p(x) - r(x) = 0$ if and only if $p = r$ with high probability. The random point is selected via the Fiat-Shamir transform on the committed data-trace over a degree-4 extension of the BabyBear field. The precise implementation of the Fiat-Shamir transform was out of scope for the audit, so the auditors assumed it was implemented correctly.

Code Assessment. The RISC Zero ZKVM code is generally well documented with comments throughout the code describing the implementation and intended behavior. However, the design and implementation of the Bigint2 precompiles were not described anywhere and there were little-to-no comments in the code outlining the intended behavior (see V-RISC0_R2-VUL-001). The Veridise auditors felt this documentation was important to have as these operations are exposed to end-users and have implicit correctness assumptions that are not stated anywhere.

Summary of Issues Detected. The security assessment uncovered 3 issues, none of which are assessed to be of high or critical severity by the Veridise analysts. Veridise auditors also identified a number of places where documentation could be improved (see V-RISC0_R2-VUL-001), where magic constants are used (see V-RISC0_R2-VUL-003), and where code did not match the comments (see V-RISC0_R2-VUL-002).

Recommendations. After conducting the assessment of the protocol, the security the analysts suggest that the RISC Zero developers document their implementation of the Bigint2 precompile (ideally both a README and with comments in the code) and replace all magic constants in their implementation with meaningful variable names.

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/zirgen	`c60423`	C++	RISC-V ZKVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Dec 2–Dec 27, 2024	Manual	2	8 person-weeks

Table 2.3: Vulnerability Summary.

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

Table 2.4: Category Breakdown.

Name	Number
Documentation	1
Logic Error	1
Magic Constants	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of RISC Zero's bigint2 precompile (which included programs that used the precompile) along with the syscall implementation of the Risc0 ZKVM host. During the assessment, the security analysts aimed to answer questions such as:

- Are there any underconstrained vulnerabilities in the Bigint2 instruction decoding constraints?
- Do the programs using the Bigint2 operations adhere to any implicit assumptions made by the operations?
- Are the out-of-circuit checks on the Bigint2 operations correct? Are there any missing checks?
- Are there any potential denial of service attacks or escape-to-host attacks using the syscall interface?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved human experts manually reviewing code.

Scope. In the `risc0/zirgen` repository, the scope of the audit was limited to the following files:

- `zirgen/circuit/bigint/bibc-exec.cpp`
- `zirgen/circuit/bigint/bigint2c.cpp`
- `zirgen/circuit/bigint/elliptic_curve.cpp`
- `zirgen/circuit/bigint/elliptic_curve.h`
- `zirgen/circuit/bigint/bibc-exec.cpp`
- `zirgen/circuit/bigint/rsa.cpp`
- `zirgen/circuit/bigint/rsa.h`
- `zirgen/circuit/rv32im/v1/edsl/bigint2.cpp`

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

- `risc0/circuit/rv32im/src/prove/emu/exec/mod.rs`
- `risc0/circuit/rv32im/src/prove/emu/preflight/mod.rs`
- `risc0/circuit/rv32im/src/prove/emu/bibc.rs`
- `risc0/circuit/rv32im/src/prove/emu/preflight/bigint2.rs`
- `risc0/zkvm/src/host/server/exec/syscall.rs`
- `risc0/zkvm/src/host/server/exec/syscall/*.rs`

Methodology. Veridise security analysts reviewed the reports of previous audits for RISC Zero, inspected the provided tests, and read the RISC Zero documentation. They then began an extensive manual review of the code. During the security assessment, the Veridise security analysts regularly met with the RISC Zero developers to ask questions about the code. The RISC Zero developers were very helpful answering the auditor's questions both in-person and over Slack.

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_R2-VUL-001	No Documentation on Bigint2	Medium	Acknowledged
V-RISC0_R2-VUL-002	Incorrect terms added to ensure positivity	Warning	Fixed
V-RISC0_R2-VUL-003	Magic Constants	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-RISC0_R2-VUL-001: No Documentation on Bigint2

Severity	Medium	**Commit**	c604231
Type	Documentation	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	multiple		
Confirmed Fix At	N/A		

The `bigint2` circuit implements a set of 7 of operations on big integers where each big integer is represented by the coefficients of a degree 16 polynomial. The operations are listed below:

1. `Nop`
2. `kOpCarry1`
3. `kOpCarry2`
4. `kOpEqz`
5. `kOpShift`
6. `kOpSetTerm`
7. `kOpEqz`

These operations also occur over 4 different register values:

1. `poly`
2. `term`
3. `tot`
4. `tmp`

The specification, motivation, assumptions, and overall design behind the implementation of `bigint2` is completely undocumented. The lack of such description makes it:

1. Challenging to assess the security of the protocol as the high level specification is not provided.
2. Challenging for future users/developers of these circuits because the assumptions and design are omitted.

Impact The lack of a specification behind the design and implementation of `bigint2` makes it 1) challenging to assess its security and 2) difficult for users/future developers to utilize safely.

Recommendation We recommend adding detailed documentation for `bigint2` including assumptions, examples, correctness arguments.

Developer Response The developers have acknowledged the issue.

4.1.2 V-RISC0_R2-VUL-002: Incorrect terms added to ensure positivity

Severity	Warning	**Commit**	c604231
Type	Logic Error	**Status**	Fixed
File(s)	zirgen/circuit/bigint/elliptic_curves.cpp		
Location(s)			
Confirmed Fix At	https://github.com/risc0/zirgen/pull/122		

As `BigInt::NondetQuotOp` and `BigInt::NondetRemOp` require non-negative inputs, suitable multiples of the prime `p` of the base field of the elliptic curve are added to enforce this. At various parts the choice of the multiple does not seem to be correct:

- Line 113: When computing `nu = yP - lambda * xP`, the term `prime` is added to ensure positivity. As we are subtracting the product `lambda * xP`, the correct term should be `prime * prime`

```
1  nu = builder.create<BigInt::AddOp>(
2      loc, nu, prime); // Quot/Rem needs nonnegative inputs, so enforce
     positivity
```

Snippet 4.1: Line 113 from elliptic_curves.cpp

- Line 129: When computing `yR = -(lambda * xR + nu)`
  - for the term `lambda * xR` we would need to add `prime * prime`
  - for the term `nu = yP - lambda * xP + prime * prime`, assuming we have added `prime * prime` to `nu` on Line 113, we would need to add `prime` (for `yP`) and `prime * prime` for the term `prime * prime`.

  Making a total `2 * prime * prime + prime`.

```
1  yR = builder.create<BigInt::AddOp>(loc, yR, prime);
2  Value prime_sqr = builder.create<BigInt::MulOp>(loc, prime, prime);
3  yR = builder.create<BigInt::AddOp>(
4      loc, yR, prime_sqr); // The prime^2 term is for the original lambda * xR
```

Snippet 4.2: Line 129 from elliptic_curves.cpp

- Line 154-157: Given the changes above, the correction terms here also need to be adjusted.

```
1  y_check_other = builder.create<BigInt::AddOp>(loc, y_check_other, prime);
2  y_check_other = builder.create<BigInt::AddOp>(loc, y_check_other, prime);
3  y_check_other = builder.create<BigInt::AddOp>(loc, y_check_other, prime_sqr);
```

Snippet 4.3: Line 154 from elliptic_curves.cpp

Impact Adding incorrect multiples of `prime` might result in negative arguments passed to `BigInt::NondetQuotOp` and `BigInt::NondetRemOp`, which would result in unexpected behavior. This will cause completeness issues and could potentially be exploited for breaking soundness.

Recommendation Make amendments as indicated above.

Developer Response The developers have fixed the issue in the following PR. However, they note that the issue does not break soundness or completeness as `nu` is not used in a `NondetQuotOp` or `NondetRemOp`.

4.1.3 V-RISC0_R2-VUL-003: Magic Constants

Severity	Info	**Commit**	c604231
Type	Magic Constants	**Status**	Acknowledged
File(s)	See issue description		
Location(s)			
Confirmed Fix At	N/A		

There are several cases where magic constants are used and should be replaced by meaningful globals:

- `zirgen/circuit/rv32im/v1/edsl/bigint2.h`

```
  RamBody ram;
  RamReg readInst;
  RamReg readRegAddr;
  std::array<RamReg, 4> io;
  OneHot<7> polyOp;
  OneHot<3> memOp;
  Reg isLast;
  Reg offset;
  Reg instWordAddr;
  std::array<Bit, 5> checkReg;
  std::array<Bit, 3> checkCoeff;
  std::array<ByteReg, 13> bytes;
  std::array<TwitByteReg, 3> twitBytes;
  FpExtReg mix;
  FpExtReg poly;
  FpExtReg term;
  FpExtReg tot;
  FpExtReg tmp;
```

Snippet 4.4: Snippet from `Bigint2CycleImpl`

The constant template parameters used in the declaration of the `io`, `checkReg`, `checkCoeff`, `bytes` and `twitBytes` fields should be replaced with meaningful constant names

- `zirgen/circuit/rv32im/v1/edsl/bigint2.cpp` (`Bigint2CycleImpl`)

```
BigInt2CycleImpl::BigInt2CycleImpl(RamHeader ramHeader)
: ram(ramHeader, 6), mix("mix"), poly("accum"), term("accum"), tot("accum"), tmp("
    accum") {
this->registerCallback("compute\_accum", \&BigInt2CycleImpl::onAccum);
}
```

Snippet 4.5: Bigint2 constructor

The constant `6` should be replaced by a meaningful name.

- `zirgen/circuit/rv32im/v1/edsl/bigint2.cpp` (`getByte`)

```
Val BigInt2CycleImpl::getByte(size_t i) {
  if (i < 13) {
    return bytes[i]->get();
```

```
  } else {
    return twitBytes[i - 13]->get();
  }
}
```

Snippet 4.6: Excerpt from `getByte`

The constant `13` should be set a meaningful variable name. In particular, the variable name should match the one chosen for the declaration of `bytes`.

- `zirgen/circuit/rv32im/v1/edsl/bigint2.cpp` (`set`)

```
    polyOp->set(0);
    memOp->set(2);
```

Snippet 4.7: Excerpt from `set`

The values of the ops being set should be changed to enum constants.

- `zirgen/circuit/rv32im/v1/edsl/bigint2.cpp` (`set`)

```
for (size_t i = 0; i < 5; i++) {
      checkReg[i]->set(0);
}
for (size_t i = 0; i < 3; i++) {
  checkCoeff[i]->set(0);
}
```

Snippet 4.8: Excerpt of `set` from bigint2

The values for `5` and `3` could be changed to `checkReg.size()` and `checkCoeff.size()` respectively.

Impact Magic constants can cause issues when refactoring implementations as every occurrence of the constant needs to be changed. As such, it is better to replace such constants with globals.

Recommendation See above.

Developer Response The developers have acknowledged the issue.

Glossary

Fiat-Shamir A well-known method for converting interactive proofs to non-interactive ones [1]. See `https://en.wikipedia.org/wiki/Fiat-Shamir_heuristic` to learn more. 2

RSA cryptography RSA is a public-key cryptosystem whose security relies on the practical difficulty of factoring the product of two large prime numbers. See `https://en.wikipedia.org/wiki/RSA_(cryptosystem)` for more. 1

Schwartz-Zippel Lemma At a high level, the Schwartz-Zippel Lemma says that for a low degree polynomial over a field , then for a random point in the field, the polynomial will be non-zero at that point with high probability. See `https://en.wikipedia.org/wiki/Schwartz-Zippel_lemma` for more. 2

Bibliography

[1] 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 13).

