Arguzz: Testing zkVMs for Soundness and Completeness Bugs

Christoph Hochrainer
christoph.hochrainer@tuwien.ac.at
TU Wien, Austria

Valentin Wüstholz
valentin.wustholz@diligence.security
Diligence Security, Austria

Maria Christakis
maria.christakis@tuwien.ac.at
TU Wien, Austria

Abstract

Zero-knowledge virtual machines (zkVMs) are increasingly deployed in decentralized applications and blockchain rollups since they enable verifiable off-chain computation. These VMs execute general-purpose programs, frequently written in Rust, and produce succinct cryptographic proofs. However, zkVMs are complex, and bugs in their constraint systems or execution logic can cause critical soundness (accepting invalid executions) or completeness (rejecting valid ones) issues.

We present Arguzz, the first automated tool for testing zkVMs for soundness and completeness bugs. To detect such bugs, Arguzz combines a novel variant of metamorphic testing with fault injection. In particular, it generates semantically equivalent program pairs, merges them into a single Rust program with a known output, and runs it inside a zkVM. By injecting faults into the VM, Arguzz mimics malicious or buggy provers to uncover overly weak constraints.

We used Arguzz to test six real-world zkVMs—RISC Zero, Nexus, Jolt, SP1, OpenVM, and Pico—and found eleven bugs in three of them. One RISC Zero bug resulted in a $50,000 bounty, despite prior audits, demonstrating the critical need for systematic testing of zkVMs.

1 Introduction

Zero-knowledge virtual machines (zkVMs) are emerging as critical infrastructure for scalable and privacy-preserving computation, especially in decentralized applications and blockchain rollups. These VMs enable general-purpose programs to be executed off-chain while producing succinct, verifiable proofs of correct execution. zkVMs are complex systems that combine compilers, execution environments, and cryptographic prover backends—components that are tightly coupled and heavily optimized for proving performance and proof size.

More specifically, typical zkVMs execute in four stages shown in Fig. 1:

1. **Preprocessing:** The program is compiled, often to a dialect of the RISC-V instruction set. This stage usually also includes the cryptographic setup for the subsequent proof generation and verification stages.
2. **Execution:** Given private and public program inputs, the execution environment runs the program and records an execution trace, called trace record. The trace record contains all information necessary to reconstruct the program behavior.
3. **Proof generation:** Given the public inputs, a cryptographic proof is produced based on the trace record and the zkVM's constraint system. Note that modern zkVMs use a universal constraint system, rather than generating a separate system for each program. Proving algorithms vary across zkVMs and are often designed to be pluggable, allowing support for multiple prover backends with different trade offs. In addition, many VMs apply proof compression to reduce the size of the generated proof, which helps minimize on-chain verification costs.
4. **Verification:** Given the public inputs, the proof is checked using the constraint system. If verification succeeds, the output is accepted as correct. Importantly, verification is decoupled from the proving process and can be carried out either by an external verifier or directly on-chain through smart contracts, enabling decentralized and transparent validation.

Given the complexity and coupling of these stages, bugs in zkVMs are both likely and difficult to detect. We focus on the two classes of bugs that developers consider most critical: *soundness* and *completeness* bugs. Soundness bugs occur when the zkVM accepts an invalid execution. These can arise when the constraint system is overly weak; for instance, the proof may be verified even when the program produces an incorrect output, thereby compromising the integrity of the system. Completeness bugs, on the other hand, happen when a valid execution is incorrectly rejected—often due to overly strict constraints that rule out legitimate behavior. This degrades user experience (since user inputs cannot be executed)

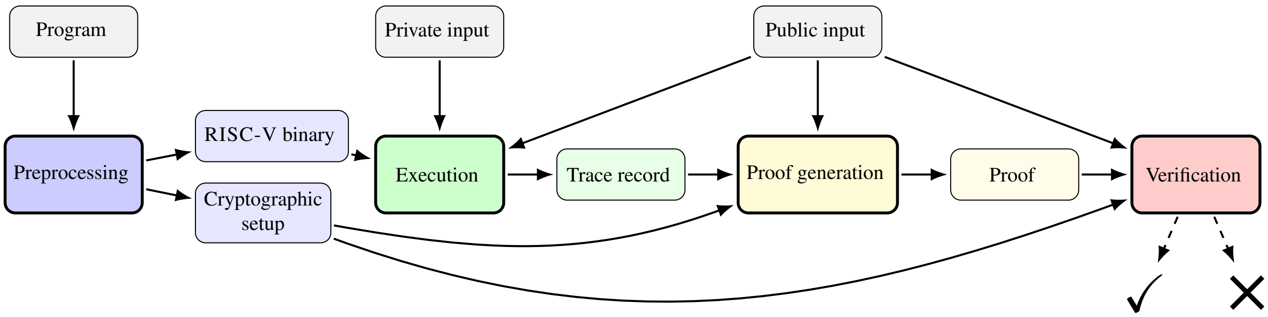

Figure 1: Overview of typical zkVM stages.

and may even lead to liveness issues. Additionally, in some cases, the constraints may simply be wrong, that is, they may misrepresent the intended semantics of the VM. Such mismatches can lead to both soundness and completeness issues, depending on whether they allow invalid executions or reject valid ones.

Both types of bugs have serious consequences. Soundness bugs can lead to fraudulent transactions being accepted in blockchain systems, violating core trust and security assumptions. Completeness bugs can block legitimate transactions or cause unexpected failures in production. In both cases, the cost of failure is high, and the complexity of zkVMs makes these bugs particularly hard to find without thorough testing.

Existing work. Our recent work [23] introduced CIRCUZZ, a fuzzer for ZK pipelines, such as CIRCOM [14], GNARK [1], and NOIR [4], which compile programs in domain-specific languages (DSLs) into constraint systems. CIRCUZZ uses *metamorphic testing* [9, 17, 40] to detect soundness and completeness bugs in these pipelines. The approach starts by generating a deterministic program, called *circuit*, in CIRCIL, an intermediate language designed to express most features of popular DSLs. Semantics-preserving transformations are applied to produce a second, equivalent circuit. Both the original and transformed circuits are then translated into a target DSL, executed on the same inputs, and their observable behaviors are compared. Divergences in behavior may indicate overly weak constraints (soundness bugs), overly strong constraints (completeness bugs), or issues that can manifest as both soundness and completeness bugs.

zkVMs share many architectural similarities with ZK pipelines but differ in two key aspects. First, they typically execute general-purpose (for instance, Rust) programs instead of DSLs. Second, in zkVMs, developers do not explicitly write constraints using assertions or DSL primitives that could cause the constraints to become unsatisfiable, and therefore, prevent proof generation. Instead, constraints are enforced automatically by the zkVM based on the semantics of the compiled Rust program, and a proof is generated. This abstraction improves usability, but also makes it harder to reason about the enforced constraints and identify bugs.

Adapting the CIRCUZZ approach to zkVMs is desirable, but faces significant challenges. First, zkVMs are significantly more computationally expensive, primarily because they must model each RISC-V instruction with precise constraints. Second, metamorphic testing cannot detect soundness bugs caused by overly weak constraints when both the original and transformed programs exhibit the same behavior. These challenges motivate the need for more efficient and targeted testing techniques tailored to zkVMs.

Our approach. In this paper, we tackle these challenges with a new approach for testing zkVMs that *integrates a novel variant of metamorphic testing with fault injection*. We implement our approach in a fuzzer called ARGUZZ. To the best of our knowledge, ARGUZZ is the first fuzzer to target zkVMs and to combine these two testing techniques.

On a high level, we design ARGUZZ as follows. First, we adapt the CIRCUZZ framework to zkVMs by translating circuits generated in CIRCIL into semantically equivalent Rust programs that can run inside zkVMs. Since CIRCIL does not support the full instruction set used by zkVMs, we extend it with custom functions that include inline assembly. This enables our circuit generator to use all instructions supported by the target VMs.

Second, to improve efficiency, we introduce a novel variant of metamorphic testing. Instead of running the original and transformed programs separately, we merge them into a single Rust program that compares their results and computes a known, expected output. This merged program is then executed inside the zkVM. Running a single combined program increases test throughput and simplifies result checking, while preserving the ability to detect behavioral inconsistencies.

Third, to detect soundness bugs due to overly weak constraints—when both the original and transformed programs exhibit the same behavior—we develop a fault-injection mechanism [7, 19, 25]. We adopt a view where the zkVM consists of two parties: the prover and the verifier. The prover corresponds to the first three stages described earlier—preprocessing, execution, and proof generation—while the verifier is the final stage that checks the proof (see Fig. 1). Given this view, we inject faults into the prover's execution

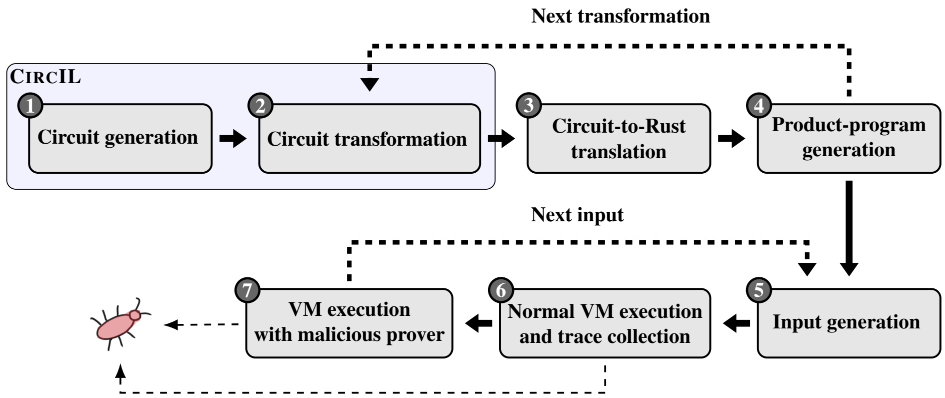

Figure 2: Overview of ARGUZZ.

logic to mimic a malicious or buggy prover—we refer to it simply as malicious in the rest of the paper. We then execute the same merged Rust program using this malicious prover and check whether the unmodified verifier accepts the resulting proof. If the program output diverges from the expected one (meaning that the fault injection successfully affected the program execution), but the verifier is deceived into accepting the proof, this indicates a soundness bug due to underconstrained behavior. Because we only report cases where a fault both changes the program's output and yields an accepted proof, this oracle rules out benign or ineffective fault injections and thereby avoids false positives.

Overall, ARGUZZ identifies three classes of bugs:

- The merged program crashes during execution on the original VM, indicating a completeness bug;
- The merged program successfully completes on the original VM but produces an unexpected output, indicating a soundness bug, a completeness bug, or an issue that can manifest as both;
- The merged program is run on a malicious prover, it produces an unexpected output, and the verifier is deceived into accepting the proof, indicating a soundness bug.

In short, ARGUZZ is the first fuzzer for zkVMs, and its design addresses two main challenges: the significantly higher computational cost of zkVMs compared to conventional ZK pipelines, and the ineffectiveness of metamorphic testing alone (as used in [23] for ZK pipelines) to reveal soundness bugs arising from weak constraints. First, ARGUZZ supports merged programs, enabling an efficient metamorphic oracle that operates within a single execution. Second, and most critically, ARGUZZ integrates a novel fault-injection mechanism at the zkVM execution level. This mechanism is explicitly designed to eliminate false positives and expose soundness bugs due to underconstrained behavior that circuit-level fuzzers, including CIRCUZZ, cannot detect.

We used ARGUZZ to test six popular zkVMs based on the RISC-V instruction set. As we discuss in our experimental evaluation, we found soundness and completeness bugs across three of these systems, namely, RISC ZERO [5], NEXUS [3], and JOLT [2, 8], for a total of *three soundness and eight completeness bugs*. One of the soundness bugs we uncovered in RISC ZERO was so severe that it earned a *$50,000 bounty* from the developers. This is significant because RISC ZERO is already deployed in production and has undergone prior security audits.

Note that we followed a responsible disclosure process for all detected issues, either privately reporting them or obtaining permission to disclose them publicly.

Contributions. Our main contributions are:

- A novel testing technique for detecting soundness and completeness bugs in zkVMs. Our technique combines an efficient variant of metamorphic testing with a fault-injection mechanism that mimics a malicious prover.
- An implementation of this technique in the open-source tool ARGUZZ[1]. ARGUZZ is designed to be modular and can be adapted to new RISC-V-based zkVMs with modest effort.
- A practical evaluation across six real-world zkVMs. ARGUZZ found eleven correctness bugs in total—three soundness and eight completeness bugs. One of the RISC ZERO soundness bugs resulted in a $50,000 bounty given its critical severity.

2 Overview

Our fuzzer tests zkVMs by combining an efficient variant of metamorphic testing and fault injection. ARGUZZ proceeds through the following seven steps, summarized in Fig. 2:

[1] Available at https://doi.org/10.5281/zenodo.16939845 and https://github.com/Rigorous-Software-Engineering/arguzz

```
1 inputs : a, b, c
2 outputs: out
3 out = (a % (b + c))
```

(a) Circuit C_1 in CircIL.

```
inputs : a, b, c
outputs: out
out = (a % ((c + 0) + b))
```

(b) Circuit C_2 in CircIL.

```
fn c1(a: u32, b: u32, c: u32) -> u32 {
  a % (b + c)
}
```

(c) Circuit C_1 as a Rust function.

```
fn c2(a: u32, b: u32, c: u32) -> u32 {
  (a % ((c + 0) + b))
}
```

(d) Circuit C_2 as a Rust function.

Figure 3: Example CircIL circuits and Rust functions generated by Arguzz.

(1) circuit generation, (2) circuit transformation, (3) circuit-to-Rust translation, (4) product-program generation, (5) input generation, (6) normal VM execution and trace collection, and (7) VM execution with malicious prover. We describe each step at a high level below and provide technical details in Sect. 3.

Step 1: Circuit generation. We begin by generating a random circuit in CircIL. This circuit represents a computation including typical control-flow patterns and arithmetic operations used in zero-knowledge applications. Unlike our prior work [23], Arguzz extends this step (and CircIL) to optionally include inline-assembly constructs that target specific instructions supported by target zkVMs. This ensures that even low-level or uncommon operations are exercised during testing. Fig. 3a shows an example circuit, denoted C_1, generated in this step. Lines 1–2 declare the circuit inputs and outputs, and line 3 computes the value of the output using a basic arithmetic expression.

Step 2: Circuit transformation. We use a similar set of semantics-preserving transformations as Circuzz [23]. These include transformations based on algebraic identities, such as commutativity, associativity, distributivity, and De Morgan's laws. They apply to logical, bitwise, and arithmetic operations alike. We disable transformations that are specific to field arithmetic and not applicable to general-purpose zkVMs. In addition, we enrich the transformation set with new rules to broaden coverage. A complete list of included transformations is provided in Appx. A.

In practice, we stack multiple transformations on the original circuit to produce a transformed circuit that is syntactically different but semantically equivalent. Fig. 3b shows the result of applying two transformations to the original circuit C_1. First, we apply the commutativity of addition (rule `comm-add` in Appx. A) to obtain output expression `(a % (c + b))`, followed by addition with the identity element (rule `zero-add-con` in Appx. A), which adds zero to `c`. Obviously, the resulting circuit, C_2, remains semantically equivalent to C_1.

In general, metamorphic transformations serve as an *oracle* for the expected behavior of the zkVM: the original and transformed circuits should produce the same output when executed—any divergence indicates a bug.

Step 3: Circuit-to-Rust translation. The original and transformed CircIL circuits are then independently compiled to Rust functions. This translation preserves the semantics of each circuit and includes any inline assembly specified during generation. Figs. 3c and 3d show the Rust translations of C_1 and C_2, respectively. Each function takes the circuit inputs, computes the output using standard Rust syntax, and returns the result.

Step 4: Product-program generation. Next, we merge the Rust functions into a single *product program*. This program executes the functions and checks that their outputs match. The structure is inspired by work on hyperproperty [20] reasoning [12, 13, 43], where product constructions are used to reason about relationships between multiple program executions. In our setting, the product program computes a known, expected output only if both function executions behave identically. This enables Arguzz to detect behavioral mismatches while avoiding the overhead of executing each function as a separate program.

Fig. 4 shows the product program (in Rust) generated by Arguzz using the functions of Figs. 3c and 3d. On lines 17–18, it calls the functions, and on line 21, it compares their outputs. If they differ, the program returns a special value `OOPS` (line 22); otherwise, it returns a `SUCCESS` value (line 24).

Step 5: Input generation. We generate random private and public inputs for the product program. These inputs are used for subsequently executing the product program in the VM. For example, for the program of Fig. 4, Arguzz might randomly generate the values `7` for `a`, `3` for `b`, and `2` for `c`.

Step 6: Normal VM execution and trace collection. Given the generated inputs, the product program is executed inside

```rust
const OOPS: u32 = 0x0;
const SUCCESS: u32 = 0xC0FFEE;

// circuit c1 as Rust function
fn c1(a: u32, b: u32, c: u32) -> u32 {
  a % (b + c)
}

// circuit c2 as Rust function
fn c2(a: u32, b: u32, c: u32) -> u32 {
  (a % ((c + 0) + b))
}

// VM entry point
[zkvm::entry(main)]
fn main(a: u32, b: u32, c: u32) -> u32 {
  let c1_out = c1(a, b, c);
  let c2_out = c2(a, b, c);

  // check if violation occurred
  if c1_out != c2_out {
    OOPS    // unexpected result
  } else {
    SUCCESS // expected result
  }
}
```

Figure 4: Product program in Rust generated by ARGUZZ using the functions of Figs. 3c and 3d.

the unmodified zkVM. If it crashes or produces an output different from the expected one, we flag this as a potential soundness or completeness bug. During this execution, we also collect the trace of the product program, which records the sequence of executed instructions. This trace is used in the next step to guide fault injection: by identifying which instructions the VM executed, we can target faults at the corresponding points in the VM's instruction-handling logic.

For example, assume that, for input values `7`, `3`, and `2`, the product program in Fig. 4 crashes; this indicates a completeness bug. Now, assume that it returns `OOPS`; this may indicate a soundness bug, a completeness bug, or an issue that manifests as both. Finally, assume that it returns `SUCCESS`; no bug is detected. If no bug is detected, we observe that the RISC-V `remu` instruction (unsigned remainder) is executed twice (as part of computing the return value of each Rust function). Knowing this, allows us to target the VM's implementation of `remu` in a fault-injected run, increasing the chances of exposing soundness bugs related to that specific operation.

Step 7: VM execution with malicious prover. Finally, we re-run the product program with the same inputs and mimic prover misbehavior by injecting faults directly into the zkVM. This process is guided by the trace collected in the previous step: we target the VM's handling of instructions that were actually executed. For example, we may modify their operands or output values. This allows us to explore how the verifier behaves in the presence of a malicious prover. If the fault injection causes the product program to return an incorrect output (i.e., `OOPS`) but the proof still verifies successfully, ARGUZZ reports a soundness bug. The verifier accepting an invalid trace indicates that the constraints are underspecified. Crucially, we require both a demonstrable deviation in execution and a successful verification, which guarantees that only truly underconstrained behavior is flagged and that ineffective injections cannot lead to false alarms.

```rust
  if c1_out != c2_out {
    OOPS // unexpected result
  else if c2_out != c3_out {
    OOPS // unexpected result
  } else {
    SUCCESS // expected result
  }
}
```

Figure 5: Part of product program in Rust generated from three semantically equivalent circuits.

Recall that, in our example, the trace collected in the previous step shows that instruction `remu rd, rs1, rs2` is executed twice, where `rd` is the destination register, `rs1` the dividend, and `rs2` the divisor. For our input values, `rs1 = 7`, `rs2 = 5`, `rd` is assigned the value `7 % 5`. In this step, we re-run the product program with the same inputs, and to mimic a malicious prover, we automatically choose to inject a fault that modifies the behavior of one of the executed `remu` instructions. Specifically, we replace the divisor `rs2` with `rs1`, causing the VM to compute `remu rd, rs1, rs1` instead. This yields `7 % 7 = 0` as the return value of one of the Rust functions, and in turn, the product program returns `OOPS`. However, due to a missing constraint in RISC ZERO, the generated proof still verifies successfully. After minimizing the product program, we are able to verify that `7 % 5 = 0`!

We reported this soundness bug to the RISC ZERO developers, who classified the issue as critical and awarded a $50,000 bounty. Importantly, the bug was not limited to the `remu` instruction; it affected any instruction that used three register operands, such as `divu`, due to missing checks in the constraint system. ARGUZZ exposed multiple such cases. The issue was subsequently patched with changes to eleven files in ZIRGEN[2], RISC ZERO's constraint-system implementation, and 32 files in the RISC ZERO[3] zkVM implementation. A new release was issued, and all clients were migrated to the updated version.

Note that ARGUZZ performs two loops, as shown in Fig. 2.

[2] https://github.com/risc0/zirgen/pull/238
[3] https://github.com/risc0/risc0/pull/3181

The first one (steps 2–4) generates new transformed circuits to exercise the constraint system in different ways. The corresponding Rust functions are all merged into the same product program for comparison. For example, we could generate a third transformed circuit and corresponding Rust function `c3` and merge it in the product program as shown Fig. 5. This allows us to compare multiple semantically equivalent functions in a single execution and detect inconsistencies across any of them. The other loop (steps 5–7) tests each product program across different inputs and fault injections. In general, these loops increase the likelihood of finding bugs with each product program.

3 Approach

As outlined in the previous section, our method combines two complementary ideas: a new variant of metamorphic testing and fault injection. Our metamorphic-testing variant provides an effective strategy for generating product programs with a known output; this allows detecting soundness and completeness bugs. On the other hand, fault injection simulates misbehavior by the prover to detect soundness bugs due to overly weak constraints. Note, however, that while our design leverages the synergy of these two ideas, they are technically orthogonal. In particular, any technique that produces programs with known outputs could be used in place of our product-program generator; however, such a substitution would eliminate metamorphic equivalence checks and therefore miss bugs that can only be exposed through metamorphic testing.

ARGUZZ workflow. As discussed, the ARGUZZ workflow consists of seven steps. In this section, we focus on the three more technically involved components of the approach–(1) circuit generation, (4) product-program generation, and (7) VM execution with malicious prover. The remaining steps are conceptually straightforward and summarized below:

- **Step 2: Circuit transformation.** This step applies semantics-preserving rewrites to the original circuit to produce a transformed variant. We reuse most transformations from CIRCUZZ, omitting those tailored to field arithmetic (irrelevant for zkVMs), and enriching the set with new rules (see Appx. A).
- **Step 3: Circuit-to-Rust translation.** Each CIRCIL circuit (original and transformed) is compiled into a standalone Rust function. This translation is direct and preserves circuit semantics, including inline assembly.
- **Step 5: Input generation.** Inputs are generated using a blackbox-fuzzing strategy guided by the type signatures of the Rust functions. We maintain a configurable set of constants that include interesting boundary values (e.g., 0, 1, −1, maximum or minimum integers, etc.) to increase the likelihood of triggering edge cases.
- **Step 6: Normal VM execution and trace collection.** The product program is executed inside the unmodified zkVM. We collect the trace record to determine which parts of the VM are exercised. This information is used to guide fault injection in the next step.

```
inputs : a, b, c
outputs: out
out = mulhsu(a, (b + c))
```

(a) Circuit *C* in CIRCIL.

```
1  fn c(a: u32, b: u32, c: u32) -> u32 {
2    macro_rules! mulhsu {
3      ($a:expr, $b:expr) => {{
4        let result: u32;
5        unsafe {
6          core::arch::asm!(
7            "mulhsu {result}, {a}, {b}",
8            result = out(reg) result,
9            a = in(reg) $a,
10           b = in(reg) $b,
11         );
12       }
13       result
14     }}
15   }
16   mulhsu!(a, (b + c))
17 }
```

(b) Circuit *C* as a Rust function.

Figure 6: Example CIRCIL circuit and Rust function generated by ARGUZZ using the inline-assembly extension.

3.1 Circuit Generation

ARGUZZ begins by generating a circuit expressed in CIRCIL, the intermediate language introduced in CIRCUZZ [23] for testing ZK pipelines. This circuit forms the basis for metamorphic transformations and subsequent execution in the zkVM.

To systematically explore the VM's behavior, we extend CIRCIL with custom functions that emit inline RISC-V assembly. This allows ARGUZZ to explicitly include specific instructions—such as `mulhsu`, which computes the upper half of the product of two integers—in the generated circuit. Such an extension is critical for ensuring broad coverage across the instruction set supported by each zkVM. As an example, the CIRCIL circuit in Fig. 6a calls the custom `mulhsu` function. Its Rust translation, shown in Fig. 6b, implements this function using a macro and inline assembly on lines 2–15.

3.2 Product-Program Generation

Our fault-injection mechanism requires programs with known outputs to reliably detect soundness bugs. However, generating programs of configurable complexity with predictable outcomes is challenging, especially when targeting a diverse set of low-level instructions supported by zkVMs.

Metamorphic testing. Metamorphic testing [17] provides an effective solution to this problem. It is a well established technique used for testing complex software systems, including compilers [16], program analyzers (e.g., [22, 29, 33–35, 38, 44, 48–50]), and ZK pipelines [23]. In such contexts, metamorphic testing generates two semantically equivalent yet syntactically different programs whose outputs must match. Specifically, the CIRCUZZ fuzzer for ZK pipelines generates an original circuit in CIRCIL and derives a transformed variant by applying a random sequence of semantics-preserving rewrites.

While metamorphic testing guarantees output equivalence between the original and transformed programs, their actual outputs are not known a priori. Therefore, traditional metamorphic testing executes both programs separately and compares their outputs externally.

Metamorphic oracles as product programs. In ARGUZZ, we generate a single product program from the original and transformed Rust functions generated in the previous step. Instead of running separate executions and performing external checks, our product program internally computes the outputs of all functions and directly compares them. Specifically, it produces one of two outcomes (see Fig. 4 for an example): `SUCCESS`, indicating all outputs match, or `OOPS`, indicating a mismatch. Thus, we effectively encode the metamorphic oracle directly into the product program. This often eliminates execution overhead and elegantly solves the requirement of knowing the output beforehand—the expected output is always `SUCCESS` unless the zkVM under test is buggy.

Regarding execution overhead, product programs often improve test throughput, mostly by reducing prover costs. Proving dominates runtime and typically scales super-linearly with trace size. Moreover, many zkVMs pad traces to the nearest power of two, which amplifies costs. If two executions are run separately, each incurs its own padding overhead. With a product program, padding is applied only once. For instance, a single trace of size 500 (padded to 512) is cheaper to prove than running two separate traces of sizes 280 and 180, which would be padded to 512 and 256 respectively. This effect becomes even more pronounced when bundling more than two Rust functions into a single product program, since multiple padding overheads are avoided.

The concept of a product program was originally introduced in the context of hyperproperty [20] reasoning [12, 13, 43]. Rather than invoking multiple program variants externally to check a hyperproperty, product programs encode the multiple executions internally and perform the necessary comparisons within a single combined program. This internalization enables hyperproperty checking without requiring repeated calls to an external oracle. For example, such reductions allow standard program-verification tools—designed for single-execution properties—to be applied to relational or multi-trace properties.

Metamorphic testing can be viewed as a form of hyperproperty reasoning, typically targeting 2-safety properties—for such properties, a failing test consists of two executions. Note that, in ARGUZZ, metamorphic testing checks k-safety properties because of the first loop in Fig. 2 (steps 2–4), which produces and compares k transformed circuits within the same product program. While product programs have a rich history in program verification, we are not aware of prior work that applies them directly to metamorphic testing.

3.3 VM Execution with Malicious Prover

Fault injection is a well established technique for uncovering bugs in software systems. The core idea is that injected faults should trigger an appropriate system response—ideally failing fast and gracefully, rather than causing silent or progressive system corruption. For example, chaos engineering, popularized by Netflix, tests the resilience of distributed systems by introducing random faults such as network outages.

The zkVM threat model. The threat model for zkVMs mirrors that of other zero-knowledge systems: the verifier is the only trusted component, while the prover is potentially adversarial. In other words, an attacker may use a malicious execution environment to generate an invalid trace and a corresponding proof, and it is the verifier's responsibility to detect and reject such proofs.

Designing effective fault injection. To test this trust boundary, we present the first fault-injection mechanism for zkVMs. In particular, we simulate a malicious prover by injecting faults into the VM's execution logic. If the product program returns an unexpected output but the verifier is successfully deceived into accepting the resulting proof, this indicates a soundness bug.

Our fault-injection mechanism deliberately encourages a "ripple effect" by introducing faults that propagate naturally through the execution trace. We inject a fault into the execution of a single instruction, e.g., by adding 1 to the result of a multiplication. The modified value is written to a register, and any subsequent instructions that read from that register propagate the altered result. This allows the fault to cascade through the trace in a way that respects normal data dependencies, increasing the likelihood that the resulting proof verifies.

With this design, the execution trace remains valid up to the point of injection. Immediately after the fault, a small,

```
1 fn execute(/* ... */) -> /* ... */ {
2   let mut i: Instruction;
3
4   // < ZKVM INSTRUCTION DECODING >
5
6   // instruction-modification injection
7   if is_injection_enabled() &&
8      is_injection_type("INSTR_MOD") &&
9      is_injection_step()
10  {
11    let new_i = fuzzer::new_instr(&i);
12    i = new_i;
13  }
14
15  // < ZKVM INSTRUCTION EXECUTION >
16
17  // increments the injection step counter
18  fuzzer::step();
19
20  // ...
21 }
```

Figure 7: Generic instruction-modification injection performed by ARGUZZ.

localized inconsistency may be introduced, but the remainder of the trace becomes consistent again—this time with respect to the faulty state. Such a localized inconsistency is more likely to evade detection if the corresponding constraints are insufficiently precise for those specific points in the trace.

An alternative design we initially considered was to generate a valid trace record and then randomly flip bits to test whether the verifier would still accept the corresponding proof. However, bit flips are not guaranteed to produce invalid traces. For example, consider a trace computing `42 * 0`: flipping the operand from `42` to `43` would still yield the same valid result of zero.

By contrast, our fault-injection mechanism considers both the program's expected output and the verifier's output to determine whether a verified proof corresponds to an invalid trace. A bug report is only generated when the injected fault actually changes the program's output—showing that execution was meaningfully altered (unlike in the above example of changing `42 * 0` to `43 * 0`)—yet the verifier still accepts the resulting proof. Faults that do not affect the program output or that are rejected by the verifier are not reported. This strict reporting rule ensures that only genuine underconstrained behavior is flagged and no false positives arise.

Fault-injection types. The fault-injection component is implemented by augmenting the zkVM's execution stage with custom logic. We define several injection *types* that describe how a fault is applied. For example, an instruction-modification injection may change the operation, the output value, or a register operand—such as altering the divisor register in a `remu` instruction (see Sect. 2)—while a memory-modification injection writes arbitrary values into memory.

Since some injection types are tailored to the internal design of a specific VM, we focus our discussion on a generic type that can be applied universally across a wide range of zkVMs, namely, the instruction-modification injection. Fig. 7 shows how we modify the execution stage of a zkVM to implement this injection type. At a high level, an instruction is decoded (line 4), the instruction is replaced by a new, fuzzed variant if certain conditions are met (lines 6–13), and the instruction is executed (line 15).

To decide where to inject, we currently use a custom, global step counter that increments with each executed instruction and triggers the injection when it matches a target step. The target step is selected by a fault-injection scheduler, described later in this section. In the figure, the step counter is checked on line 9. This mechanism is not fundamental: it can be replaced with other targeting schemes, such as using the cycle count or program counter. After the instruction execution, the counter is incremented on line 18.

To avoid accidental injections during normal execution, all injected code is guarded by a global injection flag, which is set only for fault-injection runs (line 7). A second check verifies that the injection type matches the target one (line 8)—the target injection type is chosen randomly by the fuzzer. Finally, a third check ensures that the current step matches the target injection point (line 9) as discussed earlier.

Note that certain injection types rely on randomized values—for example, generating a random operation, value, or operand—to increase behavioral diversity across test runs. These values are provided by the fuzzer, e.g., line 11 randomly fuzzes the current instruction to generate a new variant.

While the details of fault-injection code vary across zkVMs due to internal architectural differences, the instruction-modification injection type (shown in Fig. 7) is implemented in all the VMs we tested. Other injection types include modifying the program counter or altering the output of an operation before it is written to a register or memory. In some cases, we developed custom injection types tailored to a specific VM. For instance, OPENVM adopts a chip-based design, where each chip is a modular execution unit responsible for individual operations or families of operations; in this setting, we created injection types targeted at particular chips. Notably, all the soundness bugs uncovered by ARGUZZ were triggered by the instruction-modification injection. A complete list of the injection types in ARGUZZ can be found in [24].

Ensuring that injected faults actually take effect is crucial. Some faults may otherwise be blocked by safety checks in the prover's code. To prevent this, we replace built-in assertion and panic macros in the VM with custom versions. When the injection flag is set, these macros disable selected runtime checks, allowing injections to proceed without being prematurely aborted and to propagate through execution.

Fault-injection scheduler. A naive version of our fault-injection strategy could inject faults at random points in the execution trace. However, in real programs, certain instructions—such as memory reads or additions—occur far more frequently than others. As a result, purely random injection would disproportionately target common instructions, leaving rarer instructions undertested.

To address this imbalance, we implement a fairer fault-injection scheduler that aims to uniformly cover all available RISC-V instructions. The fuzzer maintains a count of how often each instruction has been targeted for injection. When analyzing the trace collected during normal execution, we identify the least frequently injected instructions and randomly select one of them. The scheduler then injects a fault at that instruction. If the selected instruction appears multiple times in the trace, the injection point is chosen uniformly at random among its occurrences.

Challenges addressed. Our fault-injection mechanism is designed to address two central challenges: precisely detecting soundness bugs due to weak constraints and ensuring that injections are directed at both common and rare instructions.

To avoid generating false positives, ARGUZZ does not report a bug solely because the verifier accepts a proof after fault injection. We additionally require that the product-program output changes from `SUCCESS` to `OOPS`. If the program crashes or produces the expected output, we cannot draw any definitive conclusions. The crash may have been caused by an unrelated side effect of the injection, or the fault may have failed to influence the execution. For instance, we observed that some injections—such as adding 2 to the first operand of a modulo operation with 2—do not affect the output or trace in a detectable way. By requiring a change in the output of the product program, we ensure that the injected fault has a concrete, observable impact on execution—one that the verifier should reject.

Moreover, ARGUZZ uses a scheduling strategy that distributes injections across instructions based on their observed frequencies in the execution trace. This prevents common instructions from being oversampled while rare instructions remain undertested. The scheduler therefore enables ARGUZZ to target unusual or low-frequency operations that are otherwise difficult to exercise, which we empirically confirm in our RQ5 experiments.

Conceptually, this fault-injection approach is not tied to any specific zkVM. Beyond zkVMs, it may generalize to ZK pipelines that expose a similar prover-verifier structure, though we leave empirical validation of this broader applicability to future work.

4 Experimental Evaluation

We evaluate ARGUZZ by testing six popular zkVMs. In our evaluation, we address the following research questions:

RQ1: How effective is ARGUZZ in detecting soundness and completeness bugs in zkVMs?

RQ2: What are characteristics of the detected bugs?

RQ3: How efficient is ARGUZZ?

RQ4: How effective is the inline-assembly extension?

RQ5: How effective is the fault-injection scheduler?

RQ6: What is the impact of the instruction-modification fault-injection type?

4.1 Experimental Setup

zkVM selection. To ensure that our evaluation focuses on zkVMs with real-world impact, we consulted with the Ethereum Foundation, who provided us with a list of mature zkVM implementations. From this list, we selected six zkVMs that are actively maintained and representative of the state of the art. This selection balances diversity of design choices with practical relevance.

Experiments in the wild. The primary goal of these experiments was to assess ARGUZZ's effectiveness in discovering previously unknown bugs in mature zkVMs. We began testing RISC ZERO and NEXUS in March 2025 and gradually extended ARGUZZ to support additional zkVMs. Support for SP1 was added in May 2025, followed by JOLT, OPENVM, and PICO in June 2025. Once a bug was discovered in a zkVM, we typically paused fuzzing that system until the issue was fixed to avoid reporting duplicates. For all zkVMs, we tested their respective main branches.

Controlled experiments. In our controlled experiments, we evaluated ARGUZZ using two configurations.

The first focused on *bug refinding*, i.e., assessing whether the reported bugs can be rediscovered. In this setup, we launched 5 fuzzing campaigns, each with a different numeric seed to initialize the fuzzer's random-number generator. Each campaign ran on 4 CPUs and was limited to 24h. To obtain transformed circuits, we applied between 1 and 4 stacked metamorphic transformations. Each product program bundled between 2 and 10 Rust functions, corresponding to the loop in steps 2–4 of Fig. 2. We executed each product program under 3 different combinations of inputs and fault injections, corresponding to the loop in steps 5–7 of Fig. 2. For tractability of the experiments, we enabled only the instruction-modification injection, which is responsible for all detected soundness bugs. To confirm that a bug found with this configuration was the same as the one originally reported, we applied the fix provided by the developers and checked that the bug disappeared. All bug-refinding experiments were performed on the original, buggy versions of the respective zkVMs, i.e., the versions predating the fixes provided by each development team.

The second configuration focused on *feature evaluation*, i.e., quantifying the benefits of ARGUZZ's design contributions such as the inline-assembly extension, the fault-injection scheduler, and other components. This setup used the same parameters as the bug-refinding configuration, with the exception that each campaign was run with one random seed. In contrast to the bug-refinding setup, all feature-evaluation experiments were conducted on the latest fixed versions of each zkVM, incorporating the developer patches for the bugs reported in this work.

Hardware. We performed all experiments on a machine with an AMD EPYC 9474F CPU @ 3.60GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm).

4.2 Experimental Results

RQ1: Effectiveness of ARGUZZ. Tab. 1 summarizes all previously unknown, unique bugs uncovered by ARGUZZ in our in-the-wild experiments. The first column assigns each bug an identifier (ID) and, where available, links to the corresponding public bug report. The second column lists the zkVM in which the bug was discovered—importantly, we found issues in three separate zkVMs. The third column links to the pull requests containing the corresponding patches. The fourth column classifies each bug by its impact, distinguishing whether it affected soundness, completeness, or both. The fifth column specifies the oracle that exposed the bug: "MT" (metamorphic testing) indicates that the product program either crashed when executed on the unmodified VM or produced an unexpected output; "FI" (fault injection) means that the product program was executed on a malicious prover, produced an unexpected output, and yet the verifier was deceived into accepting the resulting proof. The last column provides a brief description of each bug.

In total, ARGUZZ *uncovered eleven unique, previously unknown bugs across three distinct zkVMs.* This result is significant given that these are mature zkVMs, routinely subjected to audits and developed under rigorous engineering practices. Notably, both bugs identified in RISC ZERO were acknowledged with bounties: bug 1 was awarded $50,000 and bug 2 $1,000. In contrast, NEXUS and JOLT did not operate bounty programs, though they promptly addressed the reported vulnerabilities. *Of the found bugs, the eight completeness bugs were detected with metamorphic testing, while the three soundness bugs were revealed through fault injection.* This highlights the complementary strengths of the two techniques: metamorphic testing was more prolific in uncovering completeness bugs, whereas fault injection revealed the most critical soundness vulnerabilities.

Note that certain completeness bugs manifest as crashes, making metamorphic oracles unnecessary for detecting them. However, metamorphic testing remains essential for identifying soundness bugs where the zkVM produces an unexpected output without the involvement of a malicious prover. While the zkVMs we evaluate did not exhibit such soundness bugs, CIRCUZZ relied on this technique to discover all of its soundness bugs in ZK pipelines. In ARGUZZ, we additionally rely on metamorphic oracles to eliminate false positives during fault injection by identifying executions that deviate from the expected behavior. Finally, metamorphic transformations broaden program diversity through equivalent rewrites.

In addition, note that ARGUZZ operates over three input dimensions: the randomly generated programs, the randomly chosen inputs to those programs, and the fault-injection points, which are guided by our scheduler (via lightweight instruction-coverage feedback). Although the first two dimensions are purely random, they are sufficient to expose a wide range of completeness and soundness bugs in practice. Blackbox fuzzing remains a widely used and effective strategy when testing complex systems, e.g., compilers [47], database systems [39], or SMT solvers [34]. Such fuzzers generate structurally valid inputs by construction (e.g., compiling programs, valid SQL queries, valid SMT queries), which already achieve high coverage with relatively few tests. Coverage instrumentation, on the other hand, can impose substantial runtime overhead that reduces throughput, especially for zkVMs where proving dominates execution time. Nevertheless, exploring a lightweight form of coverage feedback tailored to zkVMs is an interesting direction for future work.

Taken together, these components make ARGUZZ highly effective at uncovering both soundness and completeness bugs in zkVMs. Metamorphic testing can surface both types of bugs by exercising diverse but semantically equivalent program variants, while product-program execution enables efficient equivalence checking. Fault injection complements this by exposing underconstrained behavior that manifests under adversarial executions. Moreover, the scheduler ensures broad instruction coverage, increasing the likelihood of triggering subtle constraint bugs. Combined with lightweight random program and input generation, these elements allow ARGUZZ to cover a substantial portion of the zkVM state space in practice, which is demonstrated by its ability to discover numerous correctness bugs across multiple mature implementations.

RQ2: Detected bugs. To better understand the types of issues exposed by ARGUZZ, we now examine several of the discovered bugs in more detail.

Bug 1, which we already introduced in Sect. 2, was present in RISC ZERO versions 2.0.0, 2.0.1, and 2.0.2. It affected all instructions with three register operands, such as `divu` and `remu`. The root cause was a missing constraint that caused the

Table 1: Unique bugs detected by ARGUZZ.

Bug ID	zkVM	Fix ID	Type	Oracle	Description
1	RISC ZERO	A, B	soundness	FI	Missing constraint in three-register instructions
2	RISC ZERO	C	completeness	MT	Off-by-one error in cycle-counting logic
3	NEXUS	D	soundness	FI	Unconstrained store operand in load-store instructions
4	NEXUS	E	completeness	MT	Out-of-bounds panic due to memory size misestimation
5	NEXUS	F	completeness	MT	Carry overflow in multiplication extension
6	JOLT	G	soundness	FI	Unconstrained immediate operand in `lui`
7	JOLT	H	completeness	MT	Incorrect RAM size calculation
8	JOLT	I	completeness	MT	Out-of-bounds panic for high-address bytecode
9	JOLT	J	completeness	MT	Dory-commitment failure for traces shorter than 256 cycles
10	JOLT	K	completeness	MT	Sumcheck-verification failure for `mulhsu`
11	JOLT	L, M	completeness	MT	Sumcheck-verification failure for inline `div` and `rem`

VM to fail to distinguish between the values of the first and second operand registers. As a result, proofs generated by a malicious prover could go undetected by the verifier, leading to a critical soundness bug. Addressing this bug required coordinated fixes across two repositories: not only the main RISC ZERO zkVM implementation, but also the ZIRGEN repository, which contains RISC ZERO's constraint-system implementation.

Bug 2 was due to a subtle off-by-one error in a prover component responsible for counting execution cycles: the final processing step was not included in the total. At first, this mistake was masked by an unrelated boundary-condition issue, preventing immediate failures. However, the incorrect count eventually propagated to later stages, where it caused downstream validation to fail. ARGUZZ uncovered this bug by fuzzing programs of varying sizes, which triggered the specific conditions needed for the error to manifest. The developers noted they were impressed that fuzzing revealed it, and we received a $1,000 bounty for the report.

Bug 3 was a missing constraint in NEXUS's handling of load-store instructions, which allowed a malicious prover to exploit an unconstrained memory write. Specifically, the lower bits of the register holding the store value were not properly constrained for the RISC-V store instructions `sw`, `sh`, and `sb`. As a result, a prover could alter these bits without detection. ARGUZZ uncovered this issue with the instruction-modification injection (see Fig. 7). In this case, the injection changed the second operand—the store value—to reference an arbitrary register. The flaw surfaced once the value was read back from memory and enabled unsoundly verifying $2^3 \oplus 2^3 = 1$.

Bug 4 was caused by an incorrect estimation of the size of touched or initialized memory, which led to an out-of-bounds panic inside the prover for certain programs. After our report, the developers promptly issued a dedicated patch in NEXUS version 0.3.1 to resolve the issue. Once again, the diverse range of programs generated by ARGUZZ revealed an untested edge case, exposing a completeness bug.

Bug 5 occurred after the introduction of RISC-V's multiplication extension in NEXUS. It manifested as a panic indicating that carry-flag bounds had been exceeded, and it affected multiplication, division, and remainder operations. In this case, ARGUZZ was essential not only for generating a suitable program but also for producing the input arguments required to trigger the bug. Fixing the issue required a rework of the carry logic across all RISC-V multiplication operations in NEXUS.

Bug 6 was caused by a missing constraint in the RISC-V `lui` instruction in JOLT (v0.1.0). The `lui` instruction is supposed to load a 16-bit immediate value into the upper 16 bits of a target register. Due to a missing constraint, a malicious prover could arbitrarily manipulate the immediate operand and thereby control the instruction's output. Similar to bug 3, ARGUZZ discovered this issue through an instruction-modification injection (see Fig. 7), which in this case manipulated the immediate operand. Notably, all three soundness bugs discovered by ARGUZZ were detected using the instruction-modification injection type.

In addition to the previously discussed issues, ARGUZZ uncovered five further completeness bugs in JOLT (v0.2.0). Bug 7 stemmed from an incorrect RAM size calculation that prevented certain program bytecode from fitting in memory. Bug 8 triggered an out-of-bounds panic whenever program bytecode was placed at higher memory addresses than those accessed during execution. Bug 9 was caused by the new Dory commitment [31] implementation, which failed on short traces containing fewer than 256 cycles. Bug 10 revealed a sumcheck-verification failure for the `mulhsu` instruction; note that CIRCIL circuits contained this instruction thanks to our inline-assembly extension (see Sect. 3.1). Finally, bug 11 caused a sumcheck failure when verifying inline `div` and `rem` instructions.

Beyond these zkVM bugs, ARGUZZ also uncovered a Rust compiler bug as a by-product while testing RISC ZERO. The issue was present in Rust version 1.80, which RISC ZERO used to compile input programs. A miscompilation caused a boolean expression to evaluate incorrectly, leading to both

Table 2: Number of product programs executed in each zkVM with and without fault injection enabled.

zkVM	Programs (injection on)	Programs (injection off)
RISC ZERO	1468	2294
NEXUS	788	1376
JOLT	377	503
SP1	494	880
OPENVM	1267	1591
PICO	549	1089

soundness and completeness issues. Further details about the affected expression can be found in the corresponding RISC ZERO bug report[4].

RQ3: Efficiency of ARGUZZ. In this research question, we evaluate the efficiency of ARGUZZ along three dimensions.

We first conducted a bug-refinding experiment to measure how quickly ARGUZZ can rediscover previously reported bugs. Bug 11 was not included as it was fixed after the paper-submission deadline. Out of the ten remaining bugs, ARGUZZ successfully rediscovered six (bugs 1, 3, 5, 6, 9, and 10), including all soundness bugs, within a median time of 13h. In fact, two of these were refound in under 20min and five within 6h. The remaining completeness bugs require longer (days) as they depend on very specific conditions to manifest (see RQ2). Detailed per-bug results are reported in [24]. These results are encouraging: all soundness bugs, which are the most critical from a security perspective, would have been discovered with only a few hours of fuzzing despite the size, complexity, and slow execution speeds of zkVMs. Completeness bugs 2, 4, 7, and 8 are inherently harder to hit, as they only arise under narrowly defined circumstances.

Second, we measure the runtime overhead of enabling fault injection in ARGUZZ, shown in Tab. 2. For this experiment, we use the feature-evaluation configuration and run ARGUZZ both with and without fault injection enabled. The first column lists the zkVM, the second shows the number of tested product programs with injection enabled, and the third shows the number of tested programs with injection disabled (i.e., omitting step 7 in Fig. 2). As the table shows, throughput increases substantially when disabling fault injection, ranging from 25.6% for OPENVM to 98.4% for PICO. The increase does not reach a full 2x, however, despite omitting the additional VM run in step 7. This is expected for two reasons: (1) the earlier ARGUZZ steps introduce common overhead, and (2) injected runs often terminate early (see RQ6), reducing their runtime.

Third, we evaluate the impact of product-program size on VM execution time, since running the VM is the slowest component in our workflow. For this, we use the feature-evaluation configuration and disable fault injection to avoid confounding effects on runtime. We observed that execution times typically fall within a very narrow range, even though program sizes vary widely. In particular, we did not observe a linear relationship between program size and execution time. For this reason, the number of semantically equivalent functions bundled into a product program has little effect on runtime. This is also why our metamorphic-testing variant with product programs tends to be more efficient than traditional metamorphic testing. [24] provides plots illustrating this trend.

RQ4: Effectiveness of inline-assembly extension. We now evaluate the effectiveness of our inline-assembly extension (see Sect. 3.1). We use the feature-evaluation configuration (see Sect. 4.1) and compare ARGUZZ with and without the inline-assembly extension enabled. Our metric is the percentage increase in instruction coverage, measured as the share of instructions included in the binaries generated from our product programs when the extension is enabled.

Across the tested zkVMs, we observed consistent improvements. Specifically, the inline-assembly extension increased instruction coverage by 38.2% in RISC ZERO, 15.0% in NEXUS, 45.2% in JOLT, 44.0% in SP1, 34.5% in OPENVM, and 44.0% in PICO. These gains demonstrate that the extension is effective at systematically incorporating rarely used instructions that would otherwise remain uncovered.

To provide deeper insight, [24] includes bar charts showing the instruction-frequency distributions in binaries generated from ARGUZZ programs, with and without the inline-assembly extension.

RQ5: Effectiveness of fault-injection scheduler. In this research question, we evaluate the effectiveness of the fault-injection scheduler, whose goal is to apply injections uniformly across all available RISC-V instructions (see Sect. 3.3). For this experiment, we use the feature-evaluation configuration (see Sect. 4.1) and compare ARGUZZ with and without the scheduler enabled. Without the scheduler, injections are placed at random points in the trace, which disproportionately targets common instructions and leaves rare ones underexplored. With the scheduler, injections are guided by instruction frequency to ensure more uniform coverage.

Fig. 8 shows the results as box plots. The x-axis denotes the zkVMs, while the y-axis shows how often each instruction was selected for injection. For each zkVM, the left box represents ARGUZZ without the scheduler, and the right box ARGUZZ with the scheduler. As the figure illustrates, the scheduler prevents imbalance by distributing injections evenly, thereby avoiding undertesting of rare instructions.

RQ6: Impact of instruction-modification injection. Here, we measure the impact of the instruction-modification injection (see Sect. 3.3), which is the only injection type imple-

[4] https://github.com/risc0/risc0/issues/2878

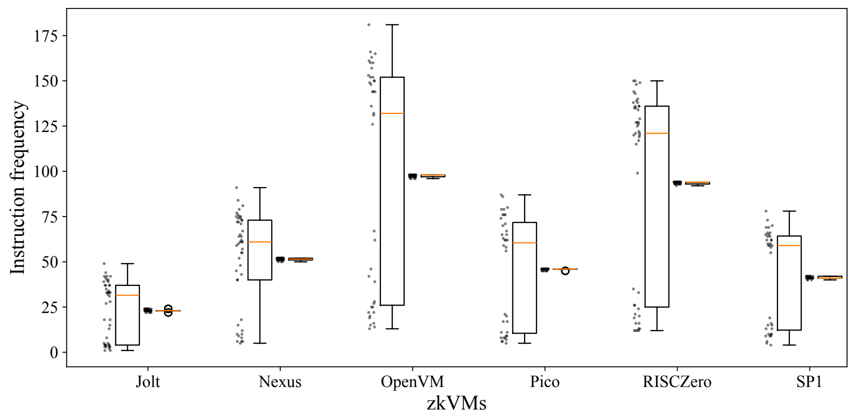

Figure 8: Distribution of injection frequencies across instructions for each zkVM, comparing ARGUZZ without (left box) and with (right box) the fault-injection scheduler.

mented across all zkVMs we tested and the one responsible for uncovering all of our soundness bugs. For this experiment, we used the feature-evaluation configuration (see Sect. 4.1).

Tab. 3 summarizes the results. The first column lists each zkVM, followed by the total number of injections applied. The remaining columns categorize their effects:

- **`SUCCESS,EC == 0`:** The product program produced the expected output (i.e., `SUCCESS`), and the zkVM terminated successfully, that is, with exit code (EC) zero. So these injections had no observable effect—the left subcolumn shows their absolute number and the right subcolumn their percentage.
- **`SUCCESS,EC != 0`:** The output of the product program was unchanged, but the zkVM terminated unsuccessfully (e.g., crashed).
- **`OOPS,EC == 0`:** This case would correspond to a soundness bug (unexpected output while the verifier succeeds). As expected, no such cases occurred here because the experiment was run on the fixed versions of the zkVMs.
- **`OOPS,EC != 0`:** The output of the product program was altered, while the zkVM execution itself failed.

The last column is the most relevant: it shows that in the vast majority of cases, the instruction-modification injection effectively perturbs executions so that the program output changes. Such output divergence is a crucial prerequisite for detecting soundness bugs caused by underconstrained behavior (see Sect. 3.3).

5 Related Work

We present the first systematic fuzzing approach for testing zkVMs. It combines metamorphic testing with fault injection to uncover both soundness and completeness bugs. While prior work has examined testing techniques for other classes of zero-knowledge systems, we are not aware of any existing work that targets zkVMs end-to-end. Despite the growing importance of zkVMs in blockchain rollups and privacy-preserving computation, their correctness has so far remained an open challenge in the literature.

In the remainder of this section, we focus on four main areas of prior research relevant to our work: fuzzing of ZK pipelines, metamorphic-testing approaches, fault-injection techniques, and fuzzing of RISC-V CPUs.

Fuzzing of ZK pipelines. The most closely related work is CIRCUZZ [23], which applies metamorphic testing to detect bugs in ZK pipelines. While CIRCUZZ targets systems that take a domain-specific circuit language as input, ARGUZZ focuses on zkVMs that execute general-purpose code, frequently in Rust, and enforce constraints implicitly from the program semantics. This shift in input language and execution model requires new techniques for program generation and performance optimization.

MTZK [45] takes a narrower scope, focusing on testing zero-knowledge compilers via metamorphic transformations. In contrast, our work targets zkVMs (that typically rely on mature compilers, such as the Rust compiler) and tests the full processing pipeline, including execution, proof generation, and verification.

Finally, unlike either CIRCUZZ or MTZK, ARGUZZ integrates a fault-injection mechanism that can simulate a malicious prover. This enables the detection of soundness bugs caused by overly weak constraints, a class of vulnerabilities that metamorphic testing alone cannot uncover.

Metamorphic testing. Metamorphic testing [17] is widely used in domains where a reliable test oracle is unavailable [9]. Segura et al. [40] provide a comprehensive survey of its applications across different domains.

Table 3: Results of the instruction-modification injection across all tested zkVMs. The table shows the total number of injections and their outcomes. `SUCCESS` means the product program output was correct, `OOPS` indicates an altered output, and `EC` denotes the exit code returned by the zkVM.

zkVM	Total injections	`SUCCESS, EC == 0`		`SUCCESS, EC != 0`		`OOPS, EC == 0`		`OOPS, EC != 0`	
RISC ZERO	4404	275	6.2%	0	0.0%	0	0.0%	4129	93.8%
NEXUS	2366	476	20.1%	83	3.5%	0	0.0%	1807	76.4%
JOLT	1040	2	0.2%	445	42.8%	0	0.0%	593	57.0%
SP1	1483	32	2.2%	833	56.2%	0	0.0%	618	41.7%
OPENVM	3801	192	5.1%	386	10.2%	0	0.0%	3223	84.8%
PICO	1649	0	0.0%	0	0.0%	0	0.0%	1649	100.0%

The most closely related applications to our work are in the testing of compilers [16] and program analyzers (e.g., [22, 29, 33–35, 38, 44, 48–50]), such as software model checkers [15, 37] and abstract interpreters [21]. In these contexts, metamorphic testing typically generates two programs that are syntactically different yet semantically equivalent, and checks whether their outputs match.

Our work differs in that we embed both executions into a single product program, rather than running them separately and comparing results externally. This design enables knowing the correct output of the product program in advance and often reduces execution overhead, especially in the prover. While metamorphic testing provides a natural way to design oracles, our work complements it with fault injection, which explores a different dimension of zkVM robustness.

Fault injection. Fault injection [7, 19, 25] has long been used to evaluate the robustness and dependability of systems. Building on these foundations, practical frameworks such as LFI [36] and dynamic stub injection [18] provide general-purpose mechanisms. These tools allow developers to inject faults at the library or API boundary to test error-handling code, independent of the underlying application logic.

More recent work combines fuzzing and fault injection. For example, FUZZTRUCTION [10] and FUZZTRUCTION-NET [11] explore cross-application testing by generating and consuming data across paired applications, such as encryption or compression tools. FUZZERR [41] focuses on inserting faults into API calls to evaluate robustness of error handling, while context-sensitive software fault injection has been used to fuzz error-handling code by tailoring injected faults to the surrounding program context [28]. IFIZZ [32] focuses on IoT firmware, efficiently generating deep-state fault scenarios to test resilience in resource-constrained environments.

In contrast, our work is the first to apply fault injection to zkVMs by injecting faults directly into the VM's execution logic. This simulates malicious prover behavior and tests whether the verifier can be deceived—revealing soundness bugs specific to zkVM constraint systems that are not addressed by prior fault-injection frameworks.

Although our approach does not infer what faults to inject, related work has explored automatically inferring likely faults and error specifications [6, 27, 36] in other domains.

Fuzzing of RISC-V CPUs. There are fuzzers that have targeted RISC-V CPUs, such as [26, 30, 42, 46], aiming to check if the hardware correctly implements the RISC-V instruction set architecture. Our work is fundamentally different and complementary to these hardware-focused approaches as ARGUZZ operates entirely at the software layer. Of course, a zkVM needs both a correctly functioning hardware CPU to run on (which CPU fuzzers like the above test) and correct software logic (which ARGUZZ validates).

6 Conclusion

We presented ARGUZZ, the first automated fuzzer for detecting soundness and completeness bugs in zkVMs. Our approach introduces a novel testing methodology that combines an efficient, product-program-based variant of metamorphic testing with a fault-injection mechanism designed to simulate malicious provers. Our evaluation demonstrates that ARGUZZ is effective across multiple zkVM implementations, uncovering both soundness and completeness bugs. The modular design allows it to be easily adapted to other zkVMs with modest engineering effort, positioning it as a practical tool for improving the reliability of this rapidly evolving ecosystem.

For future work, we plan to extend ARGUZZ in several directions. First, we aim to enhance the circuit generator to produce more complex and stateful programs, which could uncover deeper bugs in the VM execution logic. Second, exploring more sophisticated fault-injection strategies may uncover deeper vulnerabilities. Finally, we plan to adapt ARGUZZ to support zkVMs based on instruction sets beyond RISC-V, further broadening its impact and helping to secure a wider range of such systems.

Acknowledgments

We are grateful to the zkVM developers for their valuable help and to the anonymous reviewers for their constructive feedback. This work has been funded by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22007] as well as the Ethereum Foundation [Grant ID: FY25-2240].

Ethical Considerations

Our research, involving the discovery and reporting of bugs in zkVMs, was guided by a stakeholder-based ethics analysis rooted in the principles of The Menlo Report.

Stakeholder Analysis

We identified several key stakeholders who could be impacted by our research:

- **zkVM developers and companies:** The teams behind RISC ZERO, NEXUS, JOLT, and other zkVMs.
- **Users of zkVMs:** Individuals and entities who rely on applications built on these zkVMs, such as blockchain rollups, for financial transactions and other services.
- **The security-research community:** Our peers who study and build tools to improve software security.
- **The public:** The broader community that benefits from secure and trustworthy digital infrastructure.
- **Potential malicious actors:** Individuals who might seek to exploit our findings or tool for harmful purposes.

Application of Ethical Principles

Our ethical framework weighed our responsibilities to these stakeholders across the core principles, namely, beneficence, respect for persons, justice, and respect for law and public interest.

Beneficence (maximizing benefit, minimizing harm). The primary ethical driver for this work was beneficence. The research provides a significant benefit by identifying and enabling the fixing of soundness and completeness bugs in zkVMs. This directly protects users from potential exploits that could lead to fraudulent transactions and financial loss. For developers, our work serves as a valuable security layer, helping them harden their systems against critical bugs. For the research community, we contribute a novel testing technique and a powerful open-source tool, ARGUZZ.

The most significant potential harm is the misuse of ARGUZZ by malicious actors to find new, undisclosed vulnerabilities. We took direct steps to mitigate this:

- **Coordinated disclosure:** Our most critical mitigation was a strict, responsible disclosure process. For RISC ZERO, we privately reported all findings to the development team, providing them with sufficient time and information to patch the vulnerabilities before any public disclosure. This ensured that the specific threats we uncovered were neutralized before they could be exploited. For other zkVMs, the developers themselves encouraged us to report the bugs as public GitHub issues.
- **Empowering defenders:** We open-source ARGUZZ. We believe the benefit of providing a powerful defensive tool to the open-source community, enabling systematic validation and hardening of zkVMs, substantially outweighs the risk of its misuse.

Respect for persons. Our methodology centered on respecting the developers and users involved. We engaged with the zkVM development teams as partners in a collaborative effort to improve security. We did not publicly disclose vulnerabilities until we were given permission, thereby respecting the developers' process and preventing premature exposure that could harm their users and reputation.

Justice. Our work promotes justice by distributing security benefits broadly. The bugs we found protect all users of these systems, particularly those who may not have the technical expertise to assess the security of the infrastructure they rely on. By open-sourcing ARGUZZ, we make a state-of-the-art security tool available to all developers and projects, regardless of their size or financial resources, leveling the playing field for securing these complex systems.

Respect for law and public interest. Our research was conducted in a manner that serves the public interest in secure and reliable financial infrastructure. Our adherence to the industry-standard practice of coordinated vulnerability disclosure aligns with legal and ethical norms.

Decision to Proceed and Publish

Weighing these principles, the decision to proceed with the research and its publication was clear. The potential for harm was carefully managed through a robust and respectful disclosure protocol. The benefits—patched vulnerabilities, a more secure ecosystem for users, and the contribution of a new defensive tool and techniques for the research community—were substantial. The above core ethical principles all strongly supported the execution and dissemination of this work.

Open Science

We provide the artifact associated with this paper at the following link:

https://doi.org/10.5281/zenodo.16939845

The provided artifact contains the following:

- **Source code:** The complete source code for ARGUZZ.
- **Experimental scripts:** All scripts used to conduct our experimental evaluation.
- **Documentation:** A detailed `README.md` file that explains the directory structure of the artifact and provides instructions for building and running the fuzzer.

References

[1] gnark: A fast zk-SNARK library that offers a high-level API to design circuits. https://docs.gnark.consensys.io.

[2] Jolt. https://github.com/a16z/jolt.

[3] Nexus. https://github.com/nexus-xyz/nexus-zkvm.

[4] Noir. https://noir-lang.org.

[5] RISC Zero. https://github.com/risc0/risc0.

[6] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *FASE*, volume 5503 of *LNCS*, pages 370–384. Springer, 2009.

[7] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *TSE*, 16:166–182, 1990.

[8] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023. https://eprint.iacr.org/2023/1217.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *TSE*, 41:507–525, 2015.

[10] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge. In *Security*, pages 1847–1864. USENIX, 2023.

[11] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. No peer, no cry: Network application fuzzing via fault injection. In *CCS*, pages 750–764. ACM, 2024.

[12] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

[13] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.

[14] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina Melé. Circom: A circuit description language for building zero-knowledge applications. *Trans. Dependable Secur. Comput.*, 20:4733–4751, 2023.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[16] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *Comput. Surv.*, 53:4:1–4:36, 2020.

[17] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST–CS98–01, HKUST, 1998.

[18] Maria Christakis, Patrick Emmisberger, Patrice Godefroid, and Peter Müller. A general framework for dynamic stub injection. In *ICSE*, pages 586–596. IEEE Computer Society/ACM, 2017.

[19] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection: A method for validating computer-system dependability. *Computer*, 28:47–56, 1995.

[20] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, pages 51–65. IEEE Computer Society, 2008.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[22] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. Finding and understanding defects in static analyzers by constructing automated oracles. *PACMSE*, 1:1656–1678, 2024.

[23] Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. Fuzzing processing pipelines for zero-knowledge circuits. In *CCS*, pages 783–797. ACM, 2025.

[24] Christoph Hochrainer, Valentin Wüstholz, and Maria Christakis. Arguzz: Testing zkVMs for soundness and completeness bugs. *CoRR*, abs/2509.10819, 2025.

[25] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, 1997.

[26] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential fuzz testing to find CPU bugs. In *S&P*, pages 1286–1303. IEEE Computer Society, 2021.

[27] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *Security*, pages 345–362. USENIX, 2016.

[28] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Security*, pages 2595–2612. USENIX, 2020.

[29] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. Interrogation testing of program analyzers for soundness and precision issues. In *ASE*, pages 319–330. ACM, 2024.

[30] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities. In *Security*, pages 3219–3236. USENIX, 2022.

[31] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. *Cryptol. ePrint Arch.*, page 1274, 2020.

[32] Peiyu Liu, Shouling Ji, Xuhong Zhang, Qinming Dai, Kangjie Lu, Lirong Fu, Wenzhi Chen, Peng Cheng, Wenhai Wang, and Raheem Beyah. IFIZZ: Deep-state and efficient fault-scenario generation to test IoT firmware. In *ASE*, pages 805–816. IEEE Computer Society, 2021.

[33] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of Datalog engines. In *ESEC/FSE*, pages 639–650. ACM, 2021.

[34] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE*, pages 701–712. ACM, 2020.

[35] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. Dependency-aware metamorphic testing of Datalog engines. In *ISSTA*, pages 236–247. ACM, 2023.

[36] Paul Dan Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *DSN*, pages 379–388. IEEE Computer Society, 2009.

[37] Kenneth L. McMillan. Interpolation and model checking. In *Handbook of Model Checking*, pages 421–446. Springer, 2018.

[38] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. ECSTATIC: An extensible framework for testing and debugging configurable static analysis. In *ICSE*, pages 550–562. IEEE Computer Society, 2023.

[39] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *OSDI*, pages 667–682. USENIX, 2020.

[40] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *TSE*, 42:805–824, 2016.

[41] Shashank Sharma, Sai Ritvik Tanksalkar, Sourag Cherupattamoolayil, and Aravind Machiry. Fuzzing API error handling behaviors using coverage guided fault injection. In *AsiaCCS*. ACM, 2024.

[42] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: CPU fuzzing via intricate program generation. In *Security*. USENIX, 2024.

[43] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

[44] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In *PLDI*, pages 718–730. ACM, 2020.

[45] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. MTZK: Testing and exploring bugs in zero-knowledge (ZK) compilers. In *NDSS*. The Internet Society, 2025.

[46] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *Security*, pages 1307–1324. USENIX, 2023.

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294. ACM, 2011.

[48] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and understanding bugs in software model checkers. In *ESEC/FSE*, pages 763–773. ACM, 2019.

[49] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. Statfier: Automated testing of static analyzers via semantic-preserving program transformations. In *ESEC/FSE*, pages 237–249. ACM, 2023.

[50] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. Understanding and detecting annotation-induced faults of static analyzers. *PACMSE*, 1:722–744, 2024.

A ARGUZZ Rewrite Rules

<table>
<tr><th>Rule ID</th><th>Match Pattern</th><th>Rewrite Template</th></tr>
<tr><td>comm-or</td><td>?a | ?b</td><td>?b | ?a</td></tr>
<tr><td>assoc-and</td><td>(?a & ?b) & ?c</td><td>?a & (?b & ?c)</td></tr>
<tr><td>comm-and</td><td>?a & ?b</td><td>?b & ?a</td></tr>
<tr><td>and-zero</td><td>?a & 0</td><td>0</td></tr>
<tr><td>inv-xor</td><td>?a ^ ?a</td><td>0</td></tr>
<tr><td>comm-xor</td><td>?a ^ ?b</td><td>?b ^ ?a</td></tr>
<tr><td>zero-or-rev</td><td>?a | 0</td><td>?a</td></tr>
<tr><td>zero-xor-rev</td><td>?a ^ 0</td><td>?a</td></tr>
<tr><td>inv-xor-rev</td><td>0</td><td>($r:int ^ $r:int)</td></tr>
<tr><td>zero-or</td><td>?a:int</td><td>(?a | 0)</td></tr>
<tr><td>zero-xor</td><td>?a:int</td><td>(?a ^ 0)</td></tr>
<tr><td>idem-and</td><td>?a:int</td><td>(?a & ?a)</td></tr>
<tr><td>zero-and</td><td>0</td><td>($r:int & 0)</td></tr>
<tr><td>one-div</td><td>1</td><td>($r:int / $r:int)</td></tr>
<tr><td>comm-add</td><td>?a + ?b</td><td>?b + ?a</td></tr>
<tr><td>comm-mul</td><td>?a * ?b</td><td>?b * ?a</td></tr>
<tr><td>dist-mul-add</td><td>(?a + ?b) * ?c</td><td>(?a * ?c) + (?b * ?c)</td></tr>
<tr><td>dist-add-mul</td><td>(?a * ?c) + (?b * ?c)</td><td>(?a + ?b) * ?c</td></tr>
<tr><td>assoc-add</td><td>(?a + ?b) + ?c</td><td>?a + (?b + ?c)</td></tr>
<tr><td>assoc-add-rev</td><td>?a + (?b + ?c)</td><td>(?a + ?b) + ?c</td></tr>
<tr><td>assoc-mul</td><td>(?a * ?b) * ?c</td><td>?a * (?b * ?c)</td></tr>
<tr><td>assoc-mul-rev</td><td>?a * (?b * ?c)</td><td>(?a * ?b) * ?c</td></tr>
<tr><td>zero-add-des</td><td>?a + 0</td><td>?a</td></tr>
<tr><td>one-mul-des</td><td>?a * 1</td><td>?a</td></tr>
<tr><td>one-div-des</td><td>?a / 1</td><td>?a</td></tr>
<tr><td>inv-zero-add-des</td><td>?a - 0</td><td>?a</td></tr>
<tr><td>inv-add-des</td><td>?a - ?a</td><td>0</td></tr>
<tr><td>inv-assoc-neg2pos</td><td>(?a - ?b) - ?c</td><td>?a - (?b + ?c)</td></tr>
<tr><td>inv-assoc-pos2neg</td><td>?a - (?b + ?c)</td><td>(?a - ?b) - ?c</td></tr>
<tr><td>pow2-to-mul</td><td>?a ** 2</td><td>?a * ?a</td></tr>
<tr><td>pow3-to-mul</td><td>?a ** 3</td><td>(?a * ?a) * ?a</td></tr>
<tr><td>mul-to-pow2</td><td>?a * ?a</td><td>?a ** 2</td></tr>
<tr><td>mul-to-pow3</td><td>(?a * ?a) * ?a</td><td>?a ** 3</td></tr>
<tr><td>zero-add-con</td><td>?a:int</td><td>?a + 0</td></tr>
<tr><td>one-mul-con</td><td>?a:int</td><td>?a * 1</td></tr>
<tr><td>one-div-con</td><td>?a:int</td><td>?a / 1</td></tr>
<tr><td>rem-of-one-con</td><td>0</td><td>$r:int % 1</td></tr>
<tr><td>rem-of-one-des</td><td>?a % 1</td><td>0</td></tr>
<tr><td>and-to-rem</td><td>?a & 1</td><td>?a % 2</td></tr>
<tr><td>rem-to-and</td><td>?a % 2</td><td>?a & 1</td></tr>
<tr><td>inv-zero-add-con</td><td>?a:int</td><td>?a - 0</td></tr>
<tr><td>inv-addition-exp</td><td>?a - ?c</td><td>?a + (0 - ?c)</td></tr>
<tr><td>double-negation-add-con</td><td>?a:int</td><td>0 - (0 - ?a)</td></tr>
<tr><td>add-sub-random-value</td><td>?a:int</td><td>(?a - $r:int) + $r:int</td></tr>
<tr><td>zero-lor-des</td><td>?a || F</td><td>?a</td></tr>
<tr><td>zero-land-des</td><td>?a && T</td><td>?a</td></tr>
<tr><td>taut-lor</td><td>?a || T</td><td>T</td></tr>
<tr><td>contra-land</td><td>?a && F</td><td>F</td></tr>
<tr><td>assoc-lor</td><td>(?a || ?b) || ?c</td><td>?a || (?b || ?c)</td></tr>
<tr><td>assoc-land</td><td>(?a && ?b) && ?c</td><td>?a && (?b && ?c)</td></tr>
<tr><td>comm-lor</td><td>?a || ?b</td><td>?b || ?a</td></tr>
<tr><td>comm-lan</td><td>?a && ?b</td><td>?b && ?a</td></tr>
<tr><td>dist-lor-land</td><td>(?a && ?b) || ?c</td><td>(?a || ?c) && (?b || ?c)</td></tr>
<tr><td>dist-land-lor</td><td>(?a || ?c) && (?b || ?c)</td><td>(?a && ?b) || ?c</td></tr>
</table>

Rule ID	Match Pattern	Rewrite Template
de-morgan-land-con	!(?a && ?b)	(!?a) || (!?b)
de-morgan-land-des	(!?a) || (!?b)	!(?a && ?b)
de-morgan-lor-con	!(?a || ?b)	(!?a) && (!?b)
de-morgan-lor-des	(!?a) && (!?b)	!(?a || ?b)
double-negation-des	!(!?a)	?a
double-land-des	?a && ?a	?a
double-lor-des	?a || ?a	?a
double-lxor-des	?a ^^ ?a	F
comm-lxor	?a ^^ ?b	?b ^^ ?a
lxor-to-or-and	?a ^^ ?b	((!?a) && ?b) || (?a && (!?b))
zero-lor-con	?a:bool	?a || F
zero-land-con	?a:bool	?a && T
double-negation-con	?a:bool	!(!?a)
double-land-con	?a:bool	?a && ?a
double-lor-con	?a:bool	?a || ?a
double-lxor-con	F	$r:bool ^^ $r:bool
or-and-to-lxor	((!?a) && ?b) || (?a && (!?b))	?a ^^ ?b
commutativity-equ	?a == ?b	?b == ?a
relation-geq-to-leq	?a >= ?b	?b <= ?a
relation-leq-to-geq	?a <= ?b	?b >= ?a
relation-leq-to-lth-and-equ	?a <= ?b	(?a < ?b) || (?a == ?b)
relation-lth-and-equ-to-leq	(?a < ?b) || (?a == ?b)	?a <= ?b
relation-geq-to-gth-and-equ	?a >= ?b	(?a > ?b) || (?a == ?b)
relation-gth-and-equ-to-geq	(?a > ?b) || (?a == ?b)	?a >= ?b
relation-leq-to-not-gth	?a <= ?b	!(?a > ?b)
relation-not-gth-to-leq	!(?a > ?b)	?a <= ?b
relation-geq-to-not-lth	?a >= ?b	!(?a < ?b)
relation-not-lth-to-geq	!(?a < ?b)	?a >= ?b
relation-neq-to-not-equ	?a != ?b	!(?a == ?b)
relation-not-equ-to-neq	!(?a == ?b)	?a != ?b