

Efficient Branch-and-Bound Testing and Verification of zkVMs

Hideaki Takahashi
Columbia University
New York, USA
ht2673@columbia.edu

Suman Jana
Columbia University
New York, USA
suman@cs.columbia.edu

Junfeng Yang
Columbia University
New York, USA
junfeng@cs.columbia.edu

Abstract

Zero-knowledge virtual machines (zkVMs) enable verifiable execution of general-purpose programs by translating virtual machine semantics into algebraic constraints over execution traces. The correctness of these constraints is critical: a single missing or incorrect constraint can admit forged proofs (under-constrained) or reject valid executions (over-constrained). Existing approaches do not provide meaningful guarantees at production scale: fuzzers and unit tests often miss bugs, SMT solvers struggle with the size and non-linearity of constraints, and theorem provers require substantial manual effort.

We present ZEBRA, a fully automated verification and bug-detection framework grounded in a precise correctness criterion: for a given program and input, the constraint system must admit exactly one valid execution trace—no more (soundness) and no fewer (completeness). This reduces zkVM verification to a solution-set cardinality problem over a canonical trace space, where redundancies such as null-row padding and non-deterministic permutations are eliminated prior to counting.

To compute cardinality tractably, ZEBRA lifts analysis from finite-field witnesses to an integer interval lattice, exploiting a structural sparsity property of zkVM constraints: across 5 real-world zkVMs, constraints utilize only 14.0% of their theoretical connectivity capacity on average. This sparsity enables tight interval propagation with limited over-approximation error. ZEBRA performs a parallel branch-and-bound search that, unlike fuzzing, either produces a concrete counterexample or certifies the absence of violations within a bounded input region.

We evaluate ZEBRA on five real-world zkVMs. ZEBRA discovers 11 previously unknown vulnerabilities, including critical flaws enabling control-flow hijacking and proof forgery—none detected by a state-of-the-art fuzzer; 6 have already been independently confirmed and 3 have been fixed by developers. Compared to SMT-based verification, ZEBRA is 51.5× faster, verifies 16.5 percentage point more instances, and its range verification provides up to 63× efficiency gain over repeated single-input verification.

CCS Concepts

• **Security and privacy** → **Software security engineering**: *Cryptanalysis and other attacks*.

Keywords

Software Testing, Zero-Knowledge Proof, zkVM, Verification

ACM Reference Format:

Hideaki Takahashi, Suman Jana, and Junfeng Yang. 2026. Efficient Branch-and-Bound Testing and Verification of zkVMs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3830454.3846593>

1 Introduction

Zero-knowledge proofs (ZKPs) enable verifiable computation by reducing program execution to the satisfaction of algebraic constraints over a finite field. This process, arithmetization, has traditionally been performed manually using domain-specific languages such as Circom [4] and Noir [3], a workflow that is both labor-intensive and error-prone: even minor mismatches between intended semantics and generated constraints can produce severe soundness or completeness violations [22, 28, 33].

zkVMs & the shifting trust boundary. Zero-Knowledge Virtual Machines (zkVMs) address the above-mentioned issue by fixing a universal arithmetization at the level of a virtual machine, eliminating the need for per-program circuit construction [9]. Developers write programs in standard languages (e.g., Rust, C++, Go), which are compiled to a conventional ISA, typically RISC-V. Program execution produces a trace of machine states; the zkVM verifies that this trace satisfies a fixed constraint system encoding valid state transitions. Arithmetization is thus performed only once for the VM semantics and can be reused across arbitrary programs. zkVMs have gained rapid adoption in blockchain, machine learning, and supply-chain verification [6, 14, 25, 32].

This shift in abstraction fundamentally changes the trust boundary: correctness now depends entirely on the zkVM's arithmetization, the translation of virtual machine logic (CPU state transitions, memory consistency, ALU operations) into a polynomial constraint system over a finite field [16, 17]. Concretely, for a fixed program and input, this system must admit exactly the valid execution traces, no more (soundness) and no fewer (completeness). In practice, these constraints operate over large two-dimensional trace tables with thousands of cells and non-linear constraints, expressed via Algebraic Intermediate Representation (AIR) and lookup arguments [9]. Even a single missing or incorrect constraint can introduce critical vulnerabilities: under-constrained systems admit forged proofs, while over-constrained systems reject valid executions. Real-world zkVMs have exhibited both failure modes; for instance, an under-constrained bug in RISC Zero was severe enough to warrant a \$50,000 bounty [24].

Limitations of existing approaches. Existing approaches to securing zkVM constraints suffer from fundamental limitations. Dynamic methods such as unit testing and fuzzing uncover shallow



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
<https://doi.org/10.1145/3830454.3846593>

bugs by sampling concrete inputs [15], but offer no completeness guarantee: subtle constraint violations hidden in the vast state space of a large prime field often remain undetected, as shown in our evaluation. Formal verification via SMT solvers is precise but brittle [16]. Interactive theorem provers such as Lean 4 can validate constraints with high assurance [17], but require substantial manual effort and specialized expertise. Therefore, no existing method simultaneously achieves scalability, automation, and high semantic coverage over the full constraint system, while ZEBRA trades precision for scalable region reasoning with a fully automated pipeline.

Our approach. We reduce zkVM verification to a solution-set cardinality estimation problem and solve it using interval-based branch-and-bound over a canonicalized trace space. For a fixed program P and input x , let $C(P, x, T)$ denote the constraint system evaluated on trace T . We define the *solution set*

$$\mathcal{S}(P, x) = \{ T \mid C(P, x, T) \}.$$

We find that correctness of zkVM constraints reduces to a cardinality condition on $\mathcal{S}$: the system is *complete* if $|\mathcal{S}| \geq 1$, *sound* if $|\mathcal{S}| \leq 1$, and fully correct iff $|\mathcal{S}| = 1$. Under-constrained bugs manifest as $|\mathcal{S}| > 1$; over-constrained bugs as $|\mathcal{S}| = 0$. This formulation provides a unified, precise criterion for zkVM correctness.

We classify the cardinality of $\mathcal{S}(P, x)$ via the following three components. Rather than computing $|\mathcal{S}|$ exactly, we classify regions of the search space as empty, singleton, or ambiguous, enabling early certification without exhaustive enumeration.

Canonicalized table space. We eliminate representational redundancy in execution traces by introducing a canonicalization layer that maps each trace table to a unique canonical representation. Without canonicalization, padding, row permutations, or unconstrained witness cells can cause the same execution to be counted multiple times, rendering solution-set cardinality meaningless.

The central correctness requirement is injectivity: distinct canonical representations must correspond to distinct executions, so that $|\mathcal{S}|$ faithfully reflects the number of distinct executions. A key structural observation is that real-world zkVM designs admit canonicalizers of a simple and uniform form—in practice requiring on average fewer than 130 lines of code per zkVM. Across five independent zkVM implementations, we formally verify the injectivity of all canonicalizers in Lean 4,¹ establishing that canonicalization is well-defined over equivalence classes of traces and preserves solution-set cardinality.

Sound interval propagation. We evaluate constraints over interval ranges instead of exact finite-field values, soundly over-approximating all feasible executions. Modular arithmetic creates the main challenge: wrap-around at the field characteristic breaks standard interval behavior, so we design conservative propagation rules that preserve soundness. During branch-and-bound search, if propagation eliminates all interval assignments, then $\mathcal{S}(P, x) = \emptyset$; if all remaining assignments collapse to a singleton region, then $|\mathcal{S}(P, x)| = 1$.

Our interval propagation benefits from the sparsity of zkVM constraint systems, which designers maintain to reduce verification cost. Because each constraint touches only a small subset of

variables, updates propagate locally and introduce limited over-approximation error. Consistent with this, real-world zkVMs exhibit low connectivity: their density $\rho = T_{\text{arith}}(C)/(|C| \times m)$ —measuring the fraction of variables touched per constraint relative to a dense system—averages 0.140 across different backends (Appendix A). Branch-and-bound refinement then reduces the remaining error by partitioning the trace space, enabling aggressive pruning while preserving soundness.

Parallel branch-and-bound certification. We use interval propagation to guide a branch-and-bound search over the trace lattice, certifying solution-set cardinality. Each branch splits the trace space into independent subproblems, enabling natural parallelism in contrast to SMT solvers. The same refinement also tightens interval bounds, reduces over-approximation error, and accelerates convergence. This design supports localized verification: by restricting inputs to a bounded region, ZEBRA can certify correctness within that region without requiring a global proof over all inputs. If the search terminates with $|\mathcal{S}| = 1$ it certifies local correctness; otherwise, it returns concrete counterexamples. A fixed-point iteration over local certificates formally characterizes how they compose toward global correctness, analogous to composing layerwise bound propagation in neural network verification [26] albeit at a greater computational cost. We leave the implementation of this iteration as future work focusing on local verification in this paper.

Evaluation. We implement ZEBRA in Rust and evaluate it on 5 real-world zkVM backends: Pico, SP1, Sphinx, Valida, and Ziren. ZEBRA discovers 11 previously unknown zero-day vulnerabilities, of which 6 were confirmed by developers and 3 already fixed, demonstrating that the approach scales to production constraint systems and surfaces bugs that fuzzing and formal methods have missed. Our code, together with an extended version containing the appendices referenced throughout this paper, is publicly available at <https://github.com/Koukyosyumei/ZEBRA>.

In summary, this paper makes the following contributions:

- A **canonicalization-based correctness model** that reduces zkVM verification to solution-set cardinality over a canonical table space. We show that real-world zkVMs admit compact canonicalizers with formally verified correctness properties in Lean, ensuring that solution-set cardinality reflects the number of distinct executions rather than representational artifacts.
- **Sound interval propagation** over polynomial constraints in a modular field, enabling over-approximate reasoning about feasible executions with formal soundness guarantees.
- A **parallel branch-and-bound algorithm** for cardinality classification that either uncovers vulnerabilities or certifies local correctness by proving that the solution set is empty or singleton within a bounded input region, rather than merely testing for the existence of satisfying traces.
- **Empirical evaluation** identifying 11 zero-day vulnerabilities in widely deployed zkVM projects.

2 Background

This section reviews the foundations of Zero-Knowledge Proof (§ 2.1) and Zero-Knowledge Virtual Machine (§ 2.2).

¹We verify these properties at the algorithmic specification level; verifying full implementation correctness in Lean is orthogonal and left to future work.

2.1 Zero Knowledge Proof (ZKP)

A *Zero-Knowledge Proof (ZKP)* is a protocol where a prover convinces a verifier that a statement $R(x, w)$ is true without revealing anything beyond its validity [18]. It satisfies completeness, soundness, and zero knowledge: completeness means true statements are accepted, soundness means false statements are rejected except with negligible probability, and zero knowledge means the proof reveals nothing about w .

ZK Programming. A ZKP system requires users to represent the target statement as polynomial constraints C , which is complex and error-prone, as high-level logic does not map directly to polynomial constraints. ZK DSLs such as Circom [4] and Noir [3] address this by letting developers specify both witness computation P and constraints C , which are then compiled into executable code and constraint systems. Despite this, maintaining consistency between computation and constraints is difficult due to complex DSLs and finite-field arithmetic, leading to bugs that cause under-constrained or over-constrained circuits and resulting vulnerabilities.

2.2 Zero Knowledge Virtual Machine (zkVM)

While traditional ZK programming systems require developers to learn new DSLs and define arithmetic circuits for specific tasks, a *Zero-Knowledge Virtual Machine (zkVM)* provides a more general-purpose abstraction by enabling the proof of execution for any program written in a standard high-level language (e.g., Rust, C++, or Go). A zkVM typically adopts a modular architecture consisting of a guest program, a prover, and a verifier (see Fig. 1). After the execution of a guest program, a prover of a zkVM produces a cryptographic proof attesting that the guest program was correctly executed, and the verifier can check its correctness succinctly.

Trace Tables. A zkVM targets a specific ISA (e.g., RISC-V or custom ones like Cairo [12] and Valida [29]) and executes compiled machine code to produce execution traces of state transitions. These traces are encoded as finite-field matrices called *trace tables*, where each row represents an execution event, and each column represents a machine state component, such as the clock. Static data like binaries and public inputs are also treated as read-only *instance tables*.

Arithmetization of the VM. A zkVM converts execution logic into polynomial constraints over trace tables to prove correctness without revealing traces. Modern zkVMs use two types of constraints: *Algebraic Intermediate Representation (AIR)* constraints [5, 14] enforce local transitions between rows, and *lookup constraints* ensure global consistency across tables by matching entries between them [8, 13, 29]. The prover generates a proof that all constraints hold, which the verifier can succinctly check.

Example of Arithmetization. Consider a simplified execution trace for a program consisting of three instructions: STORE -4 3, which writes the immediate value 3 to memory address -4; STORE -8 1, which writes the value 1 to address -8; and ADD -12 -4 -8, which reads from addresses -4 and -8, then writes the sum 4 to address -12. The table column design is based on Valida [29], whereas many columns are omitted.

1. *Program Table (T_{prog})* is a read-only instance table that defines the compiled machine code. Each row specifies the instruction (opcode) and its associated destination (dst) and source (src1,

src2) operands. A lookup constraint between the CPU Table T_{CPU} ensures that every instruction executed in (T_{CPU}), defined by the tuple (pc, opcode, dst, src1, src2), actually exists within T_{prog} .

2. *CPU Table (T_{CPU})* serves as the main dynamic execution trace, where each row captures the current program counter (pc), the instruction being executed, and the values (val1, val2, res) processed during that cycle. Local AIR constraints ensure basic CPU logic, such as the validity of control flow and clock transitions, requiring that $\text{next.pc} = \text{curr.pc} + 4$ and $\text{next.clk} = \text{curr.clk} + 1$.

3. *ADD Table (T_{Add})* validates arithmetic correctness. When the CPU executes an ADD instruction, a lookup constraint ensures that the tuple (val1, val2, res) exists in T_{Add} as (a, b, c), where the AIR constraint $a + b = c$ holds.

3 Threat Model & Motivating Example

Threat Model. This work focuses on soundness and completeness violations in zkVMs arising from incorrect or incomplete constraint specifications. Our adversary exploits flaws in the constraint system to either (i) produce invalid execution traces that nevertheless satisfy the constraints (under-constrained behavior), or (ii) cause valid executions generated by the VM executors to be rejected by the constraints (over-constrained behavior).

We explicitly exclude infrastructure-level issues, including bugs in the compiler, executor, prover, or verifier implementations, as well as attacks on the cryptographic protocol itself. We also exclude zero-knowledge leakage, as confidentiality guarantees are application-specific and orthogonal to the correctness of the constraint satisfaction [28].

Motivating Example. Before presenting the technical details, we illustrate a simplified yet representative bug to build intuition for how our method identifies constraint bugs.

Consider a simple program with two instructions: STORE and ADD, where the inputs to ADD are concretely given: 3 and 1 (see Fig. 2). In this case, the constraint linking the ADD result to the CPU state contains a zkVM constraint bug: while the system enforces that (val1, val2, res) appears in the ADD table, it does not enforce that res is uniquely determined by val1 and val2. As a result, in addition to the intended execution trace {val1 : 3, val2 : 1, res : 4}, a second trace {val1 : 3, val2 : 1, res : 7} also satisfies the constraints, indicating an under-constrained violation.

ZEBRA reasons about sets of executions by lifting table cells to intervals. For the ADD step, we initialize $\text{res} \in [0, p - 1]$. In a correct system, constraint propagation enforces $\text{res} = \text{val}_1 + \text{val}_2 = 4$, collapsing the interval to the singleton [4, 4]. In the buggy system, there are multiple values (e.g., 4 and 7) satisfy the constraints, enabling ZEBRA to detect the under-constrained bug by producing multiple concrete traces satisfying the constraints.

While this motivating example assumes that the program and input are concrete values, § 5.5 shows ZEBRA can handle the case when they are given as range intervals.

4 Definitions of zkVM Vulnerabilities

This section introduces a formal model of constraint systems and bugs in zkVMs. Our model abstracts away prover and cryptographic details, focusing instead on the semantic relationship between instance tables, trace tables, and constraints. This abstraction enables

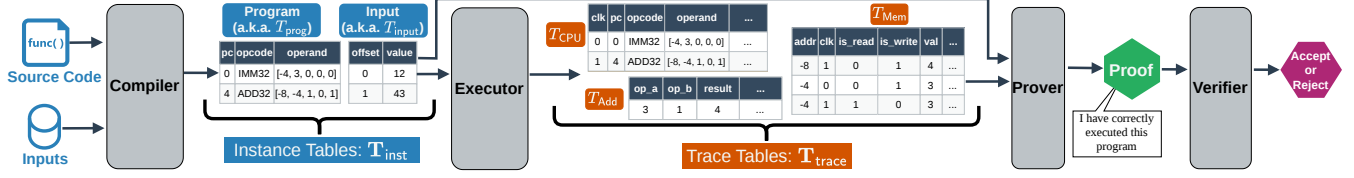


Figure 1: Typical workflow of a zkVM. The compiler converts source code into instance tables (T_{inst}), which the executor expands into trace tables (T_{trace}) representing the full execution history. If T_{trace} satisfies all algebraic constraints with respect to T_{inst} , the prover can generate a final proof to convince the verifier that the target program was executed correctly.

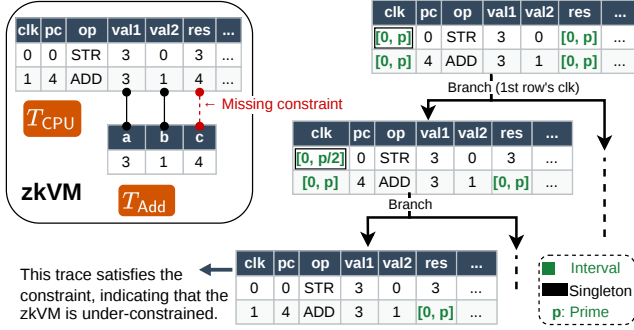


Figure 2: Motivating Example. Missing constraint allows multiple satisfying tables, leading to under-constrained bugs that ZEBRA can identify via branch-and-bound. Green values indicate intervals.

a precise classification of *under-constrained* (soundness) and *over-constrained* (completeness) bugs in the constraint systems of zkVM.

4.1 Instance and Trace Tables

Let $\mathbb{F}$ be a finite field used by the zkVM to represent scalar values. For a positive integer w , we write $(\mathbb{F}^w)^*$ to denote the set of matrices over $\mathbb{F}$ with w columns and an arbitrary number of rows.

We distinguish between *instance*, which encodes execution-invariant data such as the program binary and public inputs, and *trace tables*, which record execution-dependent data like the CPU table. For the purposes of this formalization, it is assumed that the given program binary represents a deterministic program.

Let $\mathbf{w}_{\text{inst}} = (w_{s_1}, \dots, w_{s_k})$ and $\mathbf{w}_{\text{trace}} = (w_{d_1}, \dots, w_{d_m})$ be the column widths of k instance tables and m trace tables, respectively. The spaces of instance and trace tables are then defined as:

$$\mathbb{S} = \prod_{i=1}^k (\mathbb{F}^{w_{s_i}})^*, \quad \mathbb{D} = \prod_{j=1}^m (\mathbb{F}^{w_{d_j}})^*. \quad (1)$$

4.2 Constraint System

A zkVM proves correctness of execution by enforcing algebraic constraints over trace tables, formally defined as follows.

DEFINITION 4.1 (AIR-LOOKUP-BASED ZKVM CONSTRAINTS). Formally, a constraint system in a zkVM is a function $C : \mathbb{S} \times \mathbb{D} \rightarrow$

$\{\text{True}, \text{False}\}$, which is defined as

$$C(T_{\text{inst}}, T_{\text{trace}}) = \bigwedge_{A \in \mathcal{A}} A(T_{\text{trace}}) \wedge \bigwedge_{L \in \mathcal{L}} L(T_{\text{inst}}, T_{\text{trace}}) \wedge PV(T_{\text{inst}}, T_{\text{trace}}) \quad (2)$$

Here, $\mathcal{A}$ denotes a set of AIR constraints, which enforce local transition rules between consecutive rows of trace tables, $\mathcal{L}$ denotes a set of lookup constraints enforcing global consistency across tables, and PV is a public value constraint that directly inspects the public tables.

Note that while AIR and lookup constraints are used to generate cryptographic proofs, and therefore must satisfy certain algebraic formats, the public value constraint is an auxiliary function that directly checks properties of public tables and can be arbitrary.

4.3 Bugs in zkVMs

To define bugs precisely, we reason about canonical trace tables, which uniquely represent execution traces. This avoids multiplicities caused by padding, permutation, or other representational redundancies in trace tables. For example, a zkVM might pad a trace table with empty rows to match a power-of-two height. Both the unpadded and padded versions might satisfy constraints, but they represent the same execution trace. Our canonicalization ensures these are treated as the same trace table. The detailed modeling and construction of canonical trace tables is deferred to § 4.4.

Let $\hat{\mathbb{D}} \subseteq \mathbb{D}$ denote the set of canonical trace tables, and we define a canonical solution set, which is the set of all canonical trace tables that satisfy the constraints for the fixed instance tables.

DEFINITION 4.2 (CANONICAL SOLUTION SET). For a given instance table $T_{\text{inst}} \in \mathbb{S}$, we define its canonical solution set as

$$Q(T_{\text{inst}}) = \{T_{\text{trace}} \in \hat{\mathbb{D}} \mid C(T_{\text{inst}}, T_{\text{trace}}) = \text{True}\}. \quad (3)$$

We first define the local completeness and soundness properties of the constraints in zkVMs, which are desired properties of the constraints of the zkVM for the fixed instance tables.

DEFINITION 4.3 (LOCAL COMPLETENESS). We say the constraint C in a zkVM is locally complete for the instance tables T_{inst} if

$$\exists T_{\text{trace}} \in \mathbb{D}. C(T_{\text{inst}}, T_{\text{trace}}) = \text{True} \quad (4)$$

DEFINITION 4.4 (LOCAL SOUNDNESS). We say the constraint C in a zkVM is locally sound for the instance tables T_{inst} if

$$|Q(T_{\text{inst}})| \leq 1 \quad (5)$$

Based on the above desired properties, we classify zkVM bugs according to the cardinality of the canonical solution set $Q(T_{\text{inst}})$.

DEFINITION 4.5 (OVER-CONSTRAINED zkVM). *We say the constraint C is locally over-constrained for T_{inst} if $|Q(T_{\text{inst}})| = 0$.*

Under our threat model, the execution trace generated by the VM executor for the program and input encoded in T_{inst} is a valid execution. Thus, $|Q(T_{\text{inst}})| = 0$ means that the constraint system rejects this valid execution trace, and is thus over-constrained for T_{inst} .

DEFINITION 4.6 (UNDER-CONSTRAINED zkVM). *We say the constraint C is locally under-constrained for T_{inst} if $|Q(T_{\text{inst}})| > 1$.*

This means that multiple distinct execution traces satisfy the constraint for the same program and inputs, enabling unsound proofs.

4.4 Canonical Representation

To formally reason about the uniqueness of solutions in $Q(T_{\text{inst}})$, we must ensure that the trace tables in $\mathbb{D}$ map 1-to-1 with execution traces. In practice, a zkVM might introduce representational redundancies, such as padding rows or permuting non-deterministic lookup tables, which could artificially inflate the cardinality of $Q(T_{\text{inst}})$ without representing a true logic bug. We resolve this by defining a canonical space $\hat{\mathbb{D}} \subseteq \mathbb{D}$ that filters out such artifacts.

We first introduce the set of semantic execution records $\mathbb{E}$, and the canonicalizer $\mathcal{R} : \mathbb{D} \rightarrow \mathbb{E}$ that maps trace tables to a set of execution records. Intuitively, $\mathcal{R}$ replays the trace tables to obtain the set of execution records $\mathbb{E}$, abstracting away the redundancy. Specifically, we view that $\mathcal{R}$ is a union of canonicalizers for each table, i.e., $\mathcal{R}(T_{\text{trace}}) = \mathcal{R}_{\text{cpu}}(T_{\text{cpu}}) \cup \mathcal{R}_{\text{add}}(T_{\text{add}}) \cup \mathcal{R}_{\text{mul}}(T_{\text{mul}}) \cup \dots$.

For example, consider an ALU Add Table with columns for inputs (a, b), output (res), carry bits (c), and a selector (s). While the raw table T_{Add} includes all these columns with dummy padding rows (where $s = 0$), the canonicalizer $\mathcal{R}$ acts as a semantic projection:

$$\mathcal{R}_{\text{Add}}(T_{\text{Add}}) = \{(a_i, b_i, \text{res}_i) \mid \text{row}_i \in T_{\text{Add}} \wedge s_i = 1\} \quad (6)$$

By filtering out internal witnesses (e.g., c) and padding, and normalizing the permutation, $\mathcal{R}$ maps different raw representations of the same computation to a unique record set.

DEFINITION 4.7 (SEMANTIC EQUIVALENCE OF TRACE TABLES). *We define a binary relation $\sim_{\mathcal{R}}$ on $\mathbb{D}$ derived from the canonicalizer $\mathcal{R}$. Two sets of dynamic traces $T_1, T_2 \in \mathbb{D}$ are semantically equivalent if and only if they reconstruct to the same execution record:*

$$T_1 \sim_{\mathcal{R}} T_2 \iff \mathcal{R}(T_1) = \mathcal{R}(T_2) \quad (7)$$

We then define the canonical trace tables space as the quotient set:

DEFINITION 4.8 (CANONICAL TRACE TABLE SPACE). *The space of canonical trace tables, denoted $\hat{\mathbb{D}}$, is defined as the quotient set of $\mathbb{D}$ by the relation $\sim_{\mathcal{R}}$:*

$$\hat{\mathbb{D}} := \mathbb{D} / \sim_{\mathcal{R}} = \{[T_{\text{trace}}] \sim \mid T_{\text{trace}} \in \mathbb{D}\} \quad (8)$$

, where $[T_{\text{trace}}] \sim$ represents the equivalence class of traces indistinguishable from T_{trace} by $\sim_{\mathcal{R}}$.

By construction, there exists a natural bijection between the canonical trace space $\hat{\mathbb{D}}$ and the image of the canonicalizer $\text{Im}(\mathcal{R}) \subseteq \mathbb{E}$. This ensures that every set of canonical dynamic traces corresponds to a unique semantic execution trace.

5 ZEBRA: Design

The primary challenge in testing and verifying zkVMs lies in the massive, non-linear constraint system C defined over multiple trace tables. ZEBRA addresses this by shifting from individual-witness sampling to an integer interval lattice: modeling trace cells as intervals allows the branch-and-bound method to formally and efficiently certify whole regions of the search space at once.

5.1 Localized Verification

Instead of attempting to verify the global correctness of the entire VM at once, we isolate a specific target table $T_{\text{target}} \in \mathbb{D}_i$. In this localized mode, we assume the peripheral tables $T_{\text{rest}} = T_{\text{trace}} \setminus \{T_{\text{target}}\}$ are semantically correct and consistent with the static instance tables T_{inst} . This allows us to derive a localized constraint function C by partially evaluating C on the fixed witness: $C(T_{\text{target}}) = C(T_{\text{inst}}, T_{\text{rest}} \cup \{T_{\text{target}}\})$. This modularity allows ZEBRA to bypass the complexity of global consistency proofs while exhaustively verifying the arithmetic correctness of individual units.

5.2 The Trace Interval Lattice

We model the search space for T_{target} as a lattice of integer intervals. This choice is motivated by the observation that complex zkVM constraints can be soundly approximated on intervals, enabling efficient and parallel traversal of the entire search space.

We first introduce the following notations:

1) Integer Interval: Each cell is abstracted as an integer interval, where a set of intervals is denoted as $\mathbb{I} = \{[a, b] \in \mathbb{Z} \times \mathbb{Z} \mid a \leq b\}$.

2) Interval Matrix Let $\tilde{T} \in \mathbb{I}^{H \times W}$ denote an interval matrix, where each cell $\tilde{T}_{i,j}$ is filled with an integer interval $I_{i,j}$. This $\tilde{T}$ serves as a search state in our ZEBRA.

Since the integer domain $\mathbb{Z}$ forms a lattice naturally with the partial order given by inequality, the meet by min, and the join by max, the set of all such states $\tilde{T}$ itself forms a lattice [10], under the partial order of interval inclusion ($\tilde{T} \sqsubseteq \tilde{T}'$ if $\forall i, j, \tilde{T}_{i,j} \subseteq \tilde{T}'_{i,j}$).

3) Interval-to-Field Embedding: To bridge the gap between the finite field and integer interval, we introduce the interval-to-field embedding of the integer interval $\llbracket \cdot \rrbracket_p : \mathbb{I} \rightarrow 2^{\mathbb{F}_p}$ by: $\llbracket T \rrbracket_p = \{x \bmod p \mid x \in \mathbb{Z} \text{ s.t. } a \leq x \leq b\}$. This is also naturally extended to the embedding for a table: $\llbracket \tilde{T} \rrbracket_p = \{T \mid \forall i, j, T_{i,j} \in \llbracket \tilde{T}_{i,j} \rrbracket_p\}$.

5.3 Sound Over-approximation of Constraints

This section derives the sound over-approximation for the three classes of zkVM constraints: AIR, Lookups, and public value check.

Soundness Condition. For a localized zkVM constraint $C : \mathbb{F}_p^{H \times W} \rightarrow \{\text{True}, \text{False}\}$, its sound over-approximation for an integer interval is formally defined as follows:

DEFINITION 5.1 (SOUND OVER-APPROXIMATION). $\tilde{C} : \mathbb{I}^{H \times W} \rightarrow \{\text{True}, \text{False}, \text{Uncertain}\}$ is a sound over-approximation of the constraint $C : \mathbb{F}_p^{H \times W} \rightarrow \{\text{True}, \text{False}\}$ if for any $\tilde{T} \in \mathbb{I}^{H \times W}$, it holds:

- (1) $\tilde{C}(\tilde{T}) = \text{True} \implies \forall T \in \llbracket \tilde{T} \rrbracket_p, C(T) = \text{True}.$
- (2) $\tilde{C}(\tilde{T}) = \text{False} \implies \forall T \in \llbracket \tilde{T} \rrbracket_p, C(T) = \text{False}.$
- (3) $\tilde{C}(\tilde{T}) = \text{Uncertain}$ whenever neither (1) nor (2) can hold.

This ensures that any definitive result (True or False) returned by $\tilde{C}$ is globally valid for all points in the sub-lattice.

AIR Constraints. AIR $\mathcal{A}$ consists of a set of polynomials $\{\phi_1, \phi_2, \dots\}$ where each polynomial $\phi : \mathbb{F}^{2W} \rightarrow \mathbb{F}$ is defined for two consecutive rows. The satisfaction of $\mathcal{A}$ for a concrete table T is defined as

$$\mathcal{A}(T) := \forall \phi \in \mathcal{A}. \forall i \in \{1, \dots, H-1\}. \phi(T_i, T_{i+1}) = 0 \quad (9)$$

, where T_i denotes the i -th row of the table T .

We then define the sound over-approximation $\tilde{\phi}$ using the basic interval arithmetic (see Appendix B.2 for the details). The satisfiability of AIR over $\tilde{T}$ is over-approximated using the zero-satisfiability predicate $\text{SAT}_Z([L, H]) \iff \exists k \in \mathbb{Z}. kp \in [L, H]$:

$$\tilde{\mathcal{A}}(\tilde{T}) := \begin{cases} \text{True} & \text{if } \forall \phi \in \mathcal{A}. \exists k \in \mathbb{Z}. \tilde{\phi}(\tilde{T}) = [kp, kp] \\ \text{False} & \text{if } \exists \phi \in \mathcal{A}. \neg \text{SAT}_Z(\tilde{\phi}(\tilde{T})) \\ \text{Uncertain} & \text{otherwise} \end{cases} \quad (10)$$

Lookup Constraints. Lookup constraints ensure consistency between the target table T and auxiliary tables. We categorize these into five types: Program, Byte, ALU, Control-Flow, and Memory. Detailed constructions are in Appendix B.1. For all types, the sound over-approximation $\tilde{\mathcal{L}}$ returns True if the interval collapses to a valid singleton, False if the intervals are entirely disjoint from valid states, and Uncertain otherwise.

(L-1) *Program*: $\mathcal{L}_{\text{prog}}$ ensures every executed instruction (PC, opcode, operands) exists in the static program table T_{prog} .

(L-2) *Byte*: $\mathcal{L}_{\text{byte}}$ enforces 8-bit range $[0, 255]$ and bitwise consistency (e.g., AND, XOR). $\tilde{\mathcal{L}}_{\text{byte}}$ uses bit-masking to track fixed/varying bits for tight interval bounds.

(L-3) *ALU*: $\mathcal{L}_{\text{alu}}$ validates word-level arithmetic $\diamond \in \{+, -, \times, \div, \dots\}$. $\tilde{\mathcal{L}}_{\text{alu}}$ leverages opcode-specific interval approximations $\tilde{\diamond}$ to detect arithmetic soundness violations.

(L-4) *Control-Flow*: $\mathcal{L}_{\text{cf}}$ verifies the transitions of program counters $\text{pc}' = \text{cf}_o(\text{pc}, a, b, c)$. $\tilde{\mathcal{L}}_{\text{cf}}$ lifts the transition function to its interval-based counterpart $\tilde{\text{cf}}_o$.

(L-5) *Memory*: $\mathcal{L}_{\text{mem}}$ enforces global Read-After-Write consistency and zero-value initialization. $\tilde{\mathcal{L}}_{\text{mem}}$ models a memory by dynamically splitting address segments to track value intervals across the execution timeline.

Public Value Check. Public value constraint PV inspects the public parts of tables, such as the final execution state, including the final program counter and exit code. Most of these constraints are simple equalities ($=$) or non-equalities ($\neq$), and they can be evaluated over the integer interval domain in the same way as AIR.

5.4 Lattice Branch-and-Bound Search

Algo. 1 systematically explores the lattice from the initial state $\tilde{T}_{\text{root}}$ toward concrete traces. One of the natural choices of initial interval is $[0, p-1]$, where p is the prime order of the finite field $\mathbb{F}_p$.

Basic Workflow. The algorithm maintains a queue of search states. In each iteration, it 1) SelectCellToSplit($\tilde{T}$) selects a state $\tilde{T}$ and a "divisible" cell $\tilde{T}_{i,j}$, where an interval $[a, b]$ is considered divisible if $a \neq b$; 2) SplitInterval($\tilde{T}_{i,j}$) branches by splitting the interval $\tilde{T}_{i,j}$ into k smaller intervals $\{I_1, \dots, I_k\}$, and 3) $\tilde{\mathcal{C}}$ evaluates the new states. If $\tilde{\mathcal{C}}(\tilde{T}) = \text{False}$, the entire sub-lattice is discarded, efficiently pruning the search space. When $\tilde{\mathcal{C}}(\tilde{T}) = \text{True}$, $\tilde{T}$ is concretized into

a set of concrete tables. Each table T_i is added to the result set Q only if its canonical representation $\mathcal{R}(T_i)$ has not been seen before.

Bug Condition. Any discrepancy in the cardinality of $Q(\tilde{T})$ signals a critical vulnerability: an under-constrained bug is reported if $|Q(\tilde{T})| > 1$, while an over-constrained bug is flagged if $|Q(\tilde{T})| = 0$.

Framework Guarantee. The reliability of ZEBRA rests on the convergence and completeness properties of the branch-and-bound framework applied to the interval lattice of the target table. By employing a sound approximation $\tilde{\mathcal{C}}$, ZEBRA ensures that any region of the search space pruned as False is mathematically guaranteed to contain no valid execution trace for the target component.

THEOREM 5.2 (SOUNDNESS AND COMPLETENESS OF LOCAL VERIFICATION). *Let C be a localized zkVM constraint system restricted to a target table T_{target} , given fixed instance tables T_{inst} and assumed valid peripheral trace tables T_{rest} . Then, Algo. 1 terminates, and the resulting canonical solution set Q satisfies:*

- **Local Soundness:** Every element $T \in Q$ is a concrete trace that satisfies the localized constraints: $C(T) = \text{True}$.
- **Local Completeness:** For any concrete T with $C(T) = \text{True}$, there exists $T' \in Q$ such that $\mathcal{R}(T') = \mathcal{R}(T)$.

See Appendix B.4 for the proof sketch.

Algorithm 1 Branch-and-Bound Lattice Search for zkVM

Require: Localized constraints C with respect to the given instance tables T_{inst} and the target table T_{target}

- 1: $\tilde{T}_{\text{root}} \leftarrow$ Initialize all cells in T_{target} with interval $[0, p-1]$
 - 2: $\text{Queue.push}(\tilde{T}_{\text{root}}), \quad Q \leftarrow \emptyset$ ▷ Initialization
 - 3: **while** Queue is not empty **do**
 - 4: $\tilde{T} \leftarrow \text{Queue.pop}()$
 - 5: $\text{flag} \leftarrow \tilde{\mathcal{C}}(\tilde{T})$
 - 6: **if** flag = False **then** continue ▷ Pruned
 - 7: **else if** flag = True **then** ▷ Certain
 - 8: $\{T_1, T_2, \dots\} \leftarrow \text{Concretize}(\tilde{T})$
 - 9: $Q \leftarrow Q \cup \{T_i \mid T_i \in \{T_1, T_2, \dots\}, \forall q \in Q, \mathcal{R}(q) \neq \mathcal{R}(T_i)\}$
 - 10: **else** ▷ Uncertain
 - 11: $(i, j) \leftarrow \text{SelectCellToSplit}(\tilde{T})$
 - 12: $(\tilde{T}_{\text{left}}, \tilde{T}_{\text{right}}) \leftarrow \text{SplitInterval}(\tilde{T}_{i,j})$
 - 13: $\text{Queue.push}(\tilde{T}_{\text{left}}), \quad \text{Queue.push}(\tilde{T}_{\text{right}})$
 - 14: **if** $|Q| > 1$ **then** Found Under-Constrained Bug
 - 15: **if** $|Q| = 0$ **then** Found Over-Constrained Bug
-

5.5 Input Range Verification

We apply input range verification to instruction tables, such as ALU and Control-Flow tables, whose columns cleanly partition into inputs, output, and other auxiliary cells. Under a sound and complete constraint system, fixing the inputs determines a unique canonical row: local soundness forces at most one satisfying assignment to the output, and local completeness forces at least one.

To verify a range, we initialize the inputs with a broad interval $[L, R]$ rather than a singleton value. Because our branch-and-bound algorithm maintains a total cover of the search space, it systematically partitions this range until every valid execution trace is found

and canonicalized. Let I_{in} be the set of inputs initialized with intervals, and let $V = \prod_{I \in I_{in}} |I|$ be the total number of discrete input combinations within that range. If the constraint system is perfectly sound and complete, each unique input combination should correspond to exactly one semantically distinct execution trace. ZEBRA evaluates the correctness of the range by comparing the cardinality of the resulting canonical solution set $|Q|$ with the expected volume V : sound and complete if $|Q| = V$; under-constrained if $|Q| > V$; over-constrained if $|Q| < V$.

6 ZEBRA: Implementation

We implement ZEBRA as a high-performance localized verification engine in approximately 4,500 lines of Rust.

6.1 Automated Constraint Extraction

ZEBRA automates the extraction of arithmetic constraints from source code of Plonky3-based zkVMs using a dual-path architecture: a macro-based engine for AIR and static semantic lifting for lookups. The framework is highly extensible; adapting ZEBRA to a new zkVM requires around 500 lines of tailored code.

AIR Extraction via Rust Macros. To handle diverse zkVM forks, ZEBRA uses a unified Rust macro interface that intercepts symbolic execution. This automatically captures symbolic nodes (e.g., Add, Mul, IsTransition) directly from AIR definitions, ensuring compatibility across heterogeneous implementations.

Lookup Extraction and Semantic Lifting. ZEBRA statically analyzes cross-table value transfers, which are implemented as channel-based virtual buses, and reconstructs their functional intent into semantically equivalent symbolic constraints (Fig. 3).

Public Value and Extensibility. Public value constraints, typically simple checks in standard Rust, are manually rewritten in our symbolic representation.

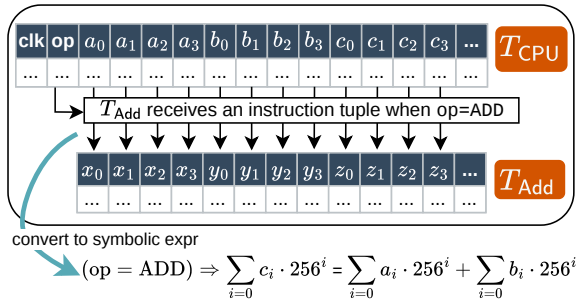


Figure 3: The CPU table offloads complex operation constraints to auxiliary tables such as the ADD table, by connecting their columns via a channel, and ZEBRA converts them into semantically equivalent constraints.

6.2 Canonicalization

A significant challenge in zkVM verification is representational redundancy, where multiple distinct matrices may represent the same execution trace due to implementation-level artifacts. For instance, a zkVM might pad a trace table with empty rows to satisfy height

requirements or utilize row permutations. Without normalizing these variations, a solver would report an inflated cardinality for Q , leading to false-positive under-constrained bug reports.

To address this, ZEBRA utilizes a canonicalizer ($\mathcal{R}$) to project raw trace tables into unique sets of semantic execution records. This function acts as a semantic filter, ignoring internal helper columns and padding while normalizing permutations.

For tables representing each ALU and control-flow instruction, the canonicalized representation of a single row is implemented as a tuple of the current program counter, the next program counter, and all operands, which sufficiently captures the input and the output effect of each instruction (see Fig. 8 for the example pseudocode).

For tables that manage memory instructions such as LW and SW, the canonicalized representation of each row is a tuple of columns corresponding to accessed memory values and immediates.

The CPU table canonicalizer focuses on reconstructing the unified timeline of execution. Each row is projected into a tuple comprising the clock cycle, the program counter, and a record of memory writes performed during that cycle.

Formalized correctness in Lean 4. We formalize the canonicalizer design in Lean 4 and verify its key correctness properties at the level of the algorithmic specification. The development assumes that, for each zkVM, the trace-table generator records every execution step as a non-padding row, with no spurious or missing records. Under this assumption, we prove three properties.

First, *generator-independence*: any two generators satisfying this assumption produce the same canonical form for a given execution, ensuring that representational artifacts such as padding and row permutations are absorbed by $\mathcal{R}$. Second, a constructive *bijection* between the canonical trace space and the image of the canonicalizer, establishing that canonical representations correspond one-to-one with trace equivalence classes. Third, *injectivity* with respect to distinct executions, which underpins the cardinality criterion of § 4.3: distinct executions map to distinct canonical forms, so $|Q|$ counts distinct executions rather than representational variants.

The framework is instantiated for the concrete column layouts of each evaluated zkVM; see Appendix C.1 for the formal statements.

6.3 Search Optimization

The search space for zkVM trace tables is inherently high-dimensional, often involving thousands of cells in a table. To ensure that ZEBRA can traverse this lattice within practical time limits, we implement several systems-level optimizations, including parallel execution, heuristic prioritization, and strategic initialization.

Parallelization. The search process is implemented as a multi-thread parallelized algorithm. Multiple worker threads concurrently pop candidate tables from a global queue, evaluate ($\hat{C}$), and execute recursive partitioning. If a thread determines that a sub-lattice is False, it immediately discards the entire branch and steals the next-highest-priority candidate from the queue.

Backward Refinement. To accelerate the convergence of the branch-and-bound search, ZEBRA implements Backward Interval Refinement, which propagates constraints in reverse to shrink the search space, based on common patterns in real-world zkVMs: Conditional Constant-Gate, Conditional Variable-Gate, and Affine

$$\begin{array}{l}
\frac{I_s = [1, 1] \quad s \cdot (v - [\tau, \tau]) = 0}{I_v \leftarrow v \cap [\tau, \tau]} \quad (\text{CONDITIONAL-CONST-GATE}) \\
\frac{I_s = [1, 1] \quad s \cdot (a - b) = 0}{I_a, I_b \leftarrow a \cap b} \quad (\text{CONDITIONAL-VAR-GATE}) \\
\begin{array}{l} I_a = [a_{lo}, a_{hi}] \quad I_L = [\ell, h] \quad I_e = [e_{lo}, e_{hi}] \\ a = L - d \cdot e \quad d > 0 \quad a, e \notin \text{FV}(L) \\ |\ell| + |h| + d(|e_{lo}| + |e_{hi}|) + |a_{lo}| + |a_{hi}| < p \end{array} \\
\frac{}{I_e \leftarrow I_e \cap \left[\left\lfloor \frac{\ell - a_{hi}}{d} \right\rfloor, \left\lfloor \frac{h - a_{lo}}{d} \right\rfloor \right]} \quad (\text{AFFINE-BACKWARD})
\end{array}$$

Figure 4: Rules for Backward Refinement. $\text{FV}(L)$ is a set of free variables in an expression L .

Backward Refinement. We denote the interval assigned for a variable x by I_x , and the interval evaluated for an expression L by I_L . See Fig. 4 for the formal inference rule and Appendix C for proofs.

(BR-1) Conditional Constant-Gate: A common pattern in zkVM arithmetization is the use of a selector variable (s) to gate a specific constraint. For instance, a constraint might enforce that a register v must equal a target constant τ only when $s = 1$. ZEBRA automatically detects this pattern and shrinks the interval of v to its intersection $v \cap [\tau, \tau]$. If the intersection $v \cap [\tau, \tau]$ is empty, the branch is immediately pruned as unsatisfiable.

(BR-2) Conditional Variable-Gate: Another popular pattern is conditional equality used to represent register-to-register moves or alias resolution. ZEBRA replaces the intervals for both a and b with their intersection $a \cap b$, and prunes the state if they are disjoint.

(BR-3) Affine Backward Refinement: The third common pattern is the affine relation. In many zkVMs, a value is decomposed into smaller limbs or quotients, and they are modeled as affine relations of the form $a = L - d \cdot e$, where a is a remainder or low-limb variable, L is an affine symbolic expression (independent of a and e), e is a quotient variable, and $d > 0$ is a constant stride. Given the current interval bounds for the affine expression $I_L = [\ell, h]$, the limb $I_a = [a_{min}, a_{max}]$, and the variables a and e do not occur free in L , we can derive tighter bounds for the quotient e .

Certified Circuit Elision. Many zkVMs use range-check circuits to verify that four-limb field elements fit within a specific range, like the BabyBear or KoalaBear range. Following best practices in formal verification, we simplify these by lifting them into dedicated IR nodes, such as BabyBearRange or KoalaBearRange. This allows the interval evaluator to apply immediate bounds to the variable, significantly reducing the complexity of the symbolic tree and accelerating the pruning process.

Optimization for Detecting Under-Constrained Bugs. In addition to the general optimizations that help the search terminate as early as possible, we introduce two techniques specifically designed to detect under-constrained bugs, which require finding satisfying assignments as quickly as possible.

Prioritization: Rather than exploring the lattice using a naive depth-first or breadth-first strategy, ZEBRA uses a priority queue to direct the search toward regions that are more likely to contain valid but unintended execution traces. For each search state $\tilde{T}$, we define its priority score as $\eta(\tilde{T}) \triangleq \sum_{\tilde{c} \in \tilde{C}} \mathbb{1}(\tilde{c}(\tilde{T}) = \text{True})$, i.e., the

number of constraints that are already definitively satisfied over the corresponding interval region. We call a state almost valid when $\eta(\tilde{T})$ is close to $|\tilde{C}|$. Because an under-constrained bug manifests as a second, distinct satisfying trace, regions in which most constraints are already satisfied are heuristically more likely to yield such a trace after further refinement than regions in which fewer constraints are definitively satisfied. This score affects only the order in which states are explored and is never used to prune a region; therefore, it does not affect soundness or the set of states explored when the search runs to completion.

Warm-Starting: ZEBRA can also leverage domain knowledge of the virtual machine. In particular, the search can be warm-started by initializing the lattice near the "honest" execution trace produced by the VM's executor. This initialization allows the tool to quickly determine whether a second, conflicting witness exists by exploring the neighborhood of a known valid execution.

7 Evaluation

We conduct a comprehensive evaluation to answer the following research questions:

RQ1: How effectively can ZEBRA locally verify real-world zkVMs at a given input point?

RQ2: How effectively can ZEBRA locally verify real-world zkVMs over a given input range?

RQ3: How does increasing the number of parallel workers affect ZEBRA's verification throughput?

RQ4: What is the contribution of each optimization to the verification performance of ZEBRA?

RQ5: Can ZEBRA uncover previously unknown bugs in real-world zkVM implementations?

RQ6: How does the bug-detection performance of ZEBRA compare to existing fuzzers?

Benchmarks. We evaluate ZEBRA on five real-world zkVMs implemented using Plonky3: Pico [7], SP1 [27], Sphinx [1], Valida [19, 30], and Ziren [23]. These systems were selected based on their popularity, measured by the number of GitHub stars and forks. Pico, SP1, and Sphinx target the RISC-V architecture, Ziren targets MIPS, and Valida targets a custom instruction set architecture (ISA). Our study focuses on verifying and testing the core tables of these zkVMs, including the CPU table, ALU tables, control-flow tables, and memory instruction tables.

Configurations. Experiments were conducted on an Intel Xeon 2.20GHz server with 31GB RAM running Ubuntu 22.04.3 LTS, utilizing four worker threads by default. We distinguish between instruction tables, which constrain the input-output effects of specific opcodes (e.g., ALU, control-flow, and memory instruction tables), and non-instruction tables, which manage global state and execution flow (e.g., the CPU table and global memory table).

Exhaustive verification analysis (RQ1–3) focused exclusively on instruction tables. We excluded non-instruction tables from this exhaustive scope because their high dimensionality (60–100 columns) and trace-dependent row counts make interval-based search computationally impractical for a full proof.

Instead, for bug detection (RQ5–6), we expanded our scope to include these non-instruction tables by leveraging warm-starting. For each table, we generate up to 30 short programs whose opcodes

and operands are instantiated uniformly at random. Thus, for a zkVM with n target tables, we use up to $30n$ honest seed traces in total. We execute each program using the target zkVM's native executor and table generator to obtain one honest seed trace. During the search, ZEBRA randomly selects subsets of target columns and incrementally increases the subset size. The selected columns are lifted to interval domains, while the remaining cells retain their concrete witness values from the seed trace. This warm-starting strategy enables ZEBRA to explore a localized neighborhood of valid executions and identify bugs in complex, large-scale tables.

For single-point trials, we performed 200 independent runs per (table, opcode) pair using 32-bit random operands and a 100-second timeout. Range verification (RQ2) targeted the standard two-operand (op_b, op_c) instruction format by injecting a random interval of width $r \in \{1, 2, 8, 32, 128\}$ into one randomly selected limb per input. This configuration yields a search volume proportional to r^2 to evaluate solver convergence under uncertainty. Scalability (RQ3) reuses exactly the single-point opcode-wise benchmark instances from RQ1, with the same operands, random seeds, and timeout; only the number of worker threads $W \in \{1, 2, 3, 4\}$ is varied. We compared ZEBRA against two baselines: the Z3 SMT solver (using QF_BV logic via SMT-LIB2) and Arguzz, a mutation-based fuzzer we extended to support our five target zkVMs (see Appendix D for the detailed configurations for those baselines).

7.1 RQ1: Local Verification Performance

Table 1 reports local verification performance under two aggregations. Opcode-wise treats each instruction opcode as a distinct target (e.g., ADD, BEQ); table-wise groups opcodes that share a single trace table (e.g., all bitwise operations folded into one BiTwis table) into one entry. We use opcode-wise figures as the headline figures and report table-wise figures as a robustness check.

Overall Performance. At the opcode level, ZEBRA verified 48.0% (61/127) of instruction targets, against Z3's 31.5% (40/127), a +16.5 pp gain. Aggregating by table widens the gap to +28.6 pp (55.8% vs. 27.2%), confirming that ZEBRA's advantage is not an artifact of how opcodes are bucketed. On the 31 operations that both tools certify, ZEBRA is on average 51.5× faster.

Per-zkVM Analysis. ZEBRA outperforms Z3 on every evaluated zkVM. The opcode-wise gap is largest on Ziren (+22.0 pp) and SP1 (+18.8 pp), and remains positive on Sphinx (+16.7 pp) and Pico (+12.0 pp); on Valida the two tools tie at 18.2% opcode-wise but ZEBRA leads 33.3% vs. 16.7% table-wise. Valida's lower absolute coverage is consistent with its bit-decomposition-heavy design, in which several tables use up to 4× as many columns as their counterparts in other zkVMs; the resulting blow-up inflates the search space for any solver operating cell-by-cell. The largest table-wise speedup, 82.4× on SP1, is driven by its jump table: ZEBRA certifies JAL and JALR in ≈ 5 ms each, whereas Z3 spends ≈ 1.3 s.

Failure Analysis. Successful verifications included simple ALU tables, multiplication, shifts, and jumps. However, deep non-linearities, such as division and multi-byte stores, remained beyond the 100s timeout. Notably, Z3 solved the division table in Valida that ZEBRA could not, as bit-vector theory provides formal decision procedures

that interval approximations lack, suggesting that hybrid interval-SMT approaches could further improve the zkVM verification.

Answer to RQ1: ZEBRA outperforms SMT verification in both efficiency and coverage, achieving a 48.0% success rate (a 16.5 percentage point (pp) improvement over Z3) and a 51.5× speedup, establishing a new performance benchmark for automated ZK constraint analysis.

7.2 RQ2: Range Verification Performance

Tab. 2 reports the speedup of ZEBRA's input range verification over a theoretical iterative point-wise verification baseline, where the baseline cost for a volume $|V|$ is computed as $|V|$ times the mean single-point verification time from RQ1. Even at the smallest volume $|V| = 2^2$, ZEBRA already outperforms point-wise enumeration, because a single interval-lattice traversal covers multiple discrete inputs with shared pruning. As $|V|$ grows, the gap widens rapidly: at $|V| = 2^{14}$, ZEBRA achieves 34.1× on SP1, 39.7× on Ziren, 42.1× on Sphinx, 44.2× on Pico, and 63.1× on Valida. This sublinear growth in verification time with respect to volume is the core benefit of input range verification: branch-and-bound collapses large contiguous regions into a single sound judgment, whereas iterative point-wise verification pays the full verification cost at every point.

Fig. 5 shows this effect on the ADD table across all zkVMs, plotting ZEBRA's measured verification time against the corresponding theoretical iterative point-wise verification cost as the volume increases. The solid curves lie strictly below the dashed iterative point-wise verification references, and the vertical gap widens as volume grows, confirming that ZEBRA verifies orders of magnitude more inputs than point-wise testing would allow; equivalently, for a fixed target volume, ZEBRA reaches it substantially faster. See Appendix D for the SMT solver baseline for range verification, where the Z3 SMT solver exhibits a scalability bottleneck.

Answer to RQ2. ZEBRA achieves substantial amortized efficiency by processing input domains as unified interval lattices rather than discrete points, yielding up to a 63× speedup over the theoretical iterative point-wise verification baseline.

7.3 RQ3: Parallel Search Scalability

Tab. 3 shows speedups from $W = 1$ baseline. Ziren drops from 0.521s to 0.246s (2.12×) at $W = 4$, while SP1, Sphinx, and Pico achieve 1.24–1.63×, showing that ZEBRA effectively distributes independent sub-lattices across workers.

Answer to RQ3. The multi-threaded parallelized algorithm significantly enhances ZEBRA's throughput, providing up to a 2.12× speedup with 4 workers.

7.4 RQ4: Ablation Study of Optimizations

Tab. 4 shows the ablation result, where #T and #O show the number of locally verified tables and instruction opcodes, respectively. We

²The two operations in Valida verified by ZEBRA and Z3 belong to different sets without any overlap.

Table 1: Local verification performance at a single input point for ZEBRA and the Z3 SMT solver. N is the number of tables or opcodes. Table-wise, N_v counts fully verified tables and Succ% is the fraction of verified opcodes averaged over tables; opcode-wise, N_v and Succ% denote verified opcodes and N_v/N . Time is the average verification time. N_c is the number of instances verified by both, and Spd is the average speedup of ZEBRA over Z3 on this common set. ZEBRA achieves higher coverage (+16.5 pp) and up to 51.5× speedup.

zkVM	Table-Wise										Opcode-Wise									
	N	ZEBRA			SMT			N _c	Spd	N	ZEBRA			SMT			N _c	Spd		
		N _v	Succ%	Time	N _v	Succ%	Time				N _v	Succ%	Time	N _v	Succ%	Time				
Valida	7	2	33.3%	0.010s	1	16.7%	0.593s	0	NA ²	11	2	18.2%	0.010s	2	18.2%	0.593s	0	NA		
Sphinx	7	4	57.1%	0.109s	2	28.6%	1.429s	2	4.7x	18	10	55.6%	0.158s	7	38.9%	1.630s	7	22.3x		
Pico	8	4	50.0%	0.095s	2	25.0%	1.117s	2	4.0x	25	10	40.0%	0.144s	7	28.0%	1.274s	7	27.5x		
SP1	10	6	60.0%	0.070s	3	33.3%	4.621s	3	82.4x	32	16	50.0%	0.100s	10	31.2%	2.497s	9	56.8x		
Ziren	12	8	66.7%	0.239s	2	28.2%	4.516s	2	1.6x	41	23	56.1%	0.246s	14	34.1%	2.976s	8	92.2x		
All	44	24	55.8%	0.132s	10	27.2%	3.339s	9	29.8x	127	61	48.0%	0.169s	40	31.5%	2.204s	31	51.5x		

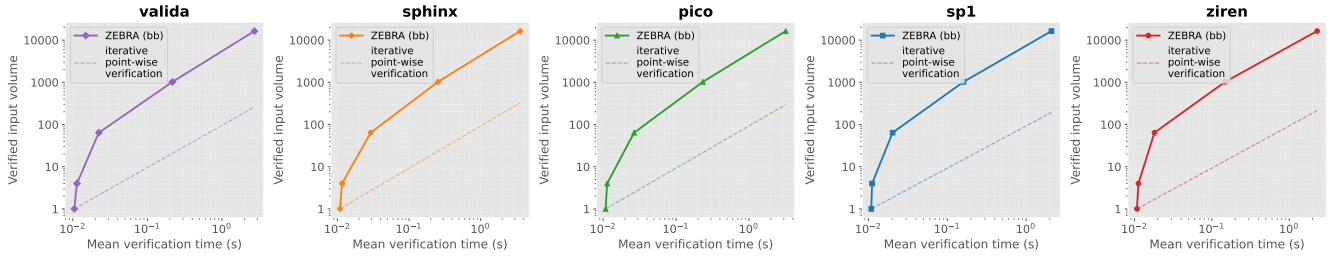


Figure 5: Verified input volume (y) versus mean verification time (x) for ZEBRA on the ADD table. The widening vertical gap at larger volumes indicates that interval-lattice reasoning amortizes increasingly well as the range grows. Z3 SMT solver cannot certify any of these tables even for a singleton input.

Table 2: Speedup of ZEBRA’s input range verification over the iterative point-wise baseline, across input volumes $|V|$.

zkVM	$ V = 2^2$	$ V = 2^6$	$ V = 2^{10}$	$ V = 2^{14}$
Valida	3.9×	30.1×	49.7×	63.1×
Sphinx	2.8×	18.3×	35.1×	42.1×
Pico	3.0×	27.3×	39.5×	44.2×
SP1	1.7×	10.0×	28.4×	34.1×
Ziren	2.5×	16.2×	31.2×	39.7×

Table 3: Impact of parallelization on verification throughput with respect to the number of parallel worker threads W

zkVM	$W = 1$	$W = 2$	$W = 3$	$W = 4$
Valida	0.011	0.011	0.011	0.010
Sphinx	0.257	0.168	0.166	0.158
Pico	0.179	0.156	0.150	0.144
SP1	0.148	0.123	0.105	0.100
Ziren	0.521	0.271	0.260	0.246

conduct an ablation study to quantify the contributions of Backward Refinement (BR) and Certified Circuit Elision (CCE). BR is the

Table 4: Impact of Backward Refinement (BR) and Certified Circuit Elision (CCE). Succ% is opcode-wise average. BR and CCE contribute to coverage by 24.4 and 11.8 pp, respectively.

zkVM	ZEBRA			ZEBRA w.o. BR			ZEBRA w.o. CCE		
	#T	#O	Succ%	#T	#O	Succ%	#T	#O	Succ%
Valida	2	2	18.2%	2	2	18.2%	2	2	18.2%
Sphinx	4	10	55.6%	2	5	27.8%	4	10	55.6%
Pico	4	10	40.0%	2	5	20.0%	4	10	40.0%
SP1	6	16	50.0%	3	7	21.9%	5	12	37.5%
Ziren	8	23	56.1%	4	11	26.8%	5	12	29.3%
Total	24	61	48.0%	13	30	23.6%	20	46	36.2%

primary driver of verification coverage: disabling it drops coverage from 48.0% to 23.6% (-31 operations), with the largest losses on Ziren (-12), SP1 (-9), Sphinx (-5) and Pico (-5). CCE is more architecture-specific, reducing coverage to 36.2% (-11.8 pp, -15 operations) when disabled. The effect is strongest on Ziren (-11) and SP1 (-4), whose AIR constraints contain selector-multiplied sub-expressions that CCE simplifies; Pico is unaffected, as its constraint structure is already relatively canonical. CCE also improves efficiency, cutting mean verification time by up to 42% on Ziren and SP1.

Answer to RQ4: BR and CCE form a synergistic pipeline: BR contributes a 24.4 pp coverage increase via search-space reduction, and CCE adds up to 11.8 pp through constraint simplification. Neither alone matches the full ZEBRA pipeline.

7.5 RQ5: Previously Unknown Bugs

ZEBRA identified 13 under- and over-constrained behaviors across the evaluated zkVMs (see Tab. 5). Of these, 11 are previously unknown bugs discovered via warm-started bug-finding mode; the remaining 2 (IDs 11 and 13) are intentional design choices surfaced during exhaustive verification. We classify severity into five categories: Critical affects arbitrary programs; High affects specific instruction families; Medium provides limited exploitability; Design Flaw denotes ISA-vs-constraint mismatches that occur only for corrupted programs; and Design Choice denotes behaviors structurally required to preserve constraint well-formedness.

ID 1 is a critical under-constrained bug in Valida where the `is_beq` and `is_bne` CPU flags are not linked to the opcode. Due to the lack of those links, a prover can toggle them to manipulate program counter transitions. IDs 2, 3, and 7 allow premature termination. In Valida, CPU constraints accept a matrix of padding rows, enabling empty executions. In Sphinx, termination is checked via `next_pc = 0`, but this is bypassed for sufficiently short traces. IDs 4 and 5 further show soundness issues: branch instructions do not verify the second operand address, enabling substitution of comparison values, and the Memory table does not enforce that initial reads return zero, allowing arbitrary initial memory.

IDs 8, 9, 10, and 12 are classified as Design Flaw, reflecting gaps between VM logic and constraints due to corrupted programs. All of them are due to program counter overflow over a finite field, where the counter wraps around to zero and falsely signals termination, despite the assumption that only `HALT` sets `next_pc = 0`. This behavior can happen only when the counter can be extremely large, and developers noted that these systems assume well-formed compiler-generated binaries, making such edge cases rare; thus, these behaviors remain unfixed in favor of performance.

IDs 11 and 13 are classified as Design Choice because the over-constrained behavior is structurally necessary: enforcing `next_pc < p` in the jump table prevents silent field wrap-around, which would otherwise enable the PC-overflow bugs. We nonetheless label these as over-constrained rather than intended behavior as the zkVM executors themselves execute such jumps without error, and the rejection discards traces that are semantically valid executions. However, this reflects the zkVM designers' intention to trade completeness on these edge cases for prover performance, as supporting targets $\geq p$ would require an additional range-decomposition machinery.

Note that our findings are not tied to specific hand-crafted programs or inputs. Eight findings (IDs 1–3, 7–10, and 12) are reachable from any suitable warm-start program and are independent of its concrete contents. IDs 4–6 require only the presence of a relevant instruction family, such as branch or memory operations, rather than a particular program or input. IDs 11 and 13 were found during exhaustive verification rather than warm-started bug finding and therefore do not depend on the RQ5 trace corpus.

Answer to RQ5. ZEBRA identified 11 previously unknown bugs across five real-world zkVMs, plus 2 known design choices, for 13 total flagged behaviors.

7.6 RQ6: Comparison with Fuzzer

Overall Detection Efficiency. We evaluate ZEBRA's bug-detection capability against Arguzz, a state-of-the-art mutation-based zkVM fuzzer. Arguzz was run for six hours per target but failed to identify any vulnerabilities in these benchmarks, consistent with prior limitations. In contrast, ZEBRA flagged 13 under- and over-constrained behaviors, 11 of them previously unknown bugs, with a 100-second budget applied per table (on average 900 seconds per zkVM) rather than per discovered bug.

Root Cause Analysis. ZEBRA and Arguzz both target soundness and completeness bugs in zkVM constraints. For under-constrained detection, however, ZEBRA explores a strictly larger search space than Arguzz due to their structural gap. The zkVM prover pipeline composes two stages: an executor E mapping a program to an execution trace, and a table generator G lifting the trace into full tables $T \in \mathbb{D}$ by populating auxiliary columns such as selector flags, carry bits, and padding witnesses. Arguzz mutates E itself into a faulty executor E' , so the tables it tests are of the form $G(E'(P))$. ZEBRA searches directly over $\mathbb{D}$. Because G is a deterministic function, Arguzz's reachable tables are contained in $\text{Im}(G \circ E') \subsetneq \mathbb{D}$. Any under-constrained bug whose exploit requires an auxiliary column to take a value G would not produce lies outside this image and is undetectable by executor-level mutation under any time budget; G does not bind a real malicious prover who writes T directly.

Bug ID 1 (Valida) illustrates this. The `is_beq` and `is_bne` flags are set by G as pure functions of the opcode, so Arguzz cannot observe them as independent variables. The constraint system fails to tie them back to the opcode, so a prover writing T directly can set `is_beq = 1` on a non-BEQ instruction and hijack control flow. The bug lives exactly in the gap between $\text{Im}(G \circ E')$ and $\mathbb{D}$, and the executor-mutation fuzzers cannot catch it.

Answer to RQ6. ZEBRA outperforms mutation-based fuzzers by operating directly on the table domain $\mathbb{D}$ rather than the executor-reachable state spaces. While Arguzz found no bugs even after 6 hours/zkVM, ZEBRA detected 13 under- and over-constrained behaviors, including 11 zero-day bugs in 900 seconds/zkVM.

8 Related Work

The correctness of zkVM constraint systems has become an active area of research as these systems gain adoption in production. Prior work falls into three broad categories: dynamic testing, SMT-based automated verification, and interactive theorem proving.

Dynamic Testing. Traditional testing methodologies for zkVMs rely on point-wise sampling to surface logic discrepancies. Production zkVMs such as SP1 employ extensive negative unit tests that inject random mutations into the proof generation process to uncover under-constrained bugs. zkFuzz [28] introduced the first general fuzzing framework for ZK circuits based on code mutation, and Arguzz [15] extended these ideas to zkVMs by combining

Table 5: Under- and over-constrained behaviors discovered by ZEBRA. UC = under-constrained; OC = over-constrained. Severity: **Critical: affects arbitrary programs; **High**: affects specific instruction families with broad consequences; **Medium**: affects specific instruction families with limited exploitability; **Design Flaw**: affects only malformed programs. **Design Choice**: structurally required to preserve constraint well-formedness.**

zkVM	ID	Type	Severity	Description	Status
Valida	1	UC	Critical	Opcode flags such as <code>is_beq</code> / <code>is_bne</code> are not properly linked to the opcode; suppressing these flags allows malicious provers to alter control flow.	Fixed
	2	UC	Critical	The CPU table does not constrain <code>is_real = 1</code> on the first row; a prover can produce an accepted proof of empty execution for any program.	Fixed
	3	UC	Critical	The Final instruction is not constrained to be <code>STOP</code> in any verification stage; execution may halt arbitrarily for any program with an accepted proof.	Fixed
	4	UC	High	Branch instructions do not verify the second operand when it is an address; a prover can substitute an arbitrary address to alter the comparison value.	In contact
	5	UC	High	The Memory table does not constrain zero-initialized cells to hold zero; a prover can supply non-zero initial values to any cell.	Confirmed
	6	UC	Medium	Written values lack a range check; distinct byte decompositions evaluate identically over the finite field, allowing the modification of written values.	In contact
Sphinx	7	UC	Critical	When the execution trace is sufficiently short, program termination is not checked, allowing a malicious prover to terminate prematurely.	In contact
	8	UC	Design Flaw	only <code>next_pc = 0</code> is required to show the normal termination; counter overflow over finite field can satisfy this condition without executing a halt instruction.	In contact
Pico	9	UC	Design Flaw	The verifier uses <code>next_pc = 0</code> as proof of normal termination by a halt syscall; counter overflow produces the same field value, falsely signaling a clean halt.	Confirmed
SP1	10	UC	Design Flaw	Both <code>next_pc = 0</code> and <code>exit_code = 0</code> checks the successful termination, yet neither condition distinguishes a halt syscall from counter overflow.	In contact
	11	OC	Design Choice	Jump table enforces <code>next_pc < p</code> where $p = 2^{31} - 2^{27} + 1$; jump targets $\geq p$ are unrepresentable in a valid trace, causing the table to reject it.	Known
Ziren	12	UC	Design Flaw	The same <code>next_pc = 0</code> check is applied over KoalaBear ($p = 2^{31} - 2^{24} + 1$); counter overflow satisfies the halt condition without a halt syscall.	Confirmed
	13	OC	Design Choice	Jump table enforces <code>next_pc < p</code> where $p = 2^{31} - 2^{24} + 1$; jump targets $\geq p$ are unrepresentable in a valid trace, causing the table to reject it.	Known

metamorphic testing with fault injection into the virtual machine’s executor. While effective at uncovering shallow bugs, these dynamic approaches provide no theoretical guarantee of completeness and are structurally unable to reach tables whose helper columns are populated only during witness generation.

SMT-based Verification. Another direction is SMT solvers to verify ZK constraint systems automatically. Picus [22] combines structural analysis with SMT-based reasoning to detect under-constrained bugs in ZK circuits, and has been extended to SP1 [20] and Ziren [21]. These deployments, while promising, have three structural limits, and they are not directly comparable to our evaluated Z3 baseline and cannot serve as drop-in experimental baselines for ZEBRA: Picus targets determinism, so over-constrained behaviors are out of scope; published deployments verify only small sub-chips, as scaling is blocked by Plonky3’s lack of modular constraint blocks and I/O annotations [20]; and Picus’s circuit model does not natively handle lookup or permutation constraints [16].

ZIVER [16] addresses the last point with an inductive consistency-checking algorithm for SP1 that supports lookups, but requires manual translation to a custom DSL and covers only 7 small sub-components. Finally, while finite-field-aware SMT theories such as `cvc5` have recently emerged, they still lack support for constraint patterns common in zkVMs, such as disequality ($\neq$) constraints used in public value checks.

Interactive Theorem Proving. Interactive theorem provers (ITPs) offer strong correctness guarantees at the cost of substantial manual effort. Cairo and Jolt have been formalized in Lean and ACL2 [2, 17]. OSS tools such as Clean [31] and zkLean [11] pursue the complementary approach of co-designing circuits alongside their correctness proofs. Despite their strong correctness guarantees, they require expert-written proofs for every new component or architectural change, and scale poorly as zkVM designs evolve.

Relation to abstract interpretation. ZEBRA’s interval evaluator can be viewed as an abstract interpretation of a finite constraint system, and its branch-and-bound structure is related to interval constraint propagation. Classical abstract interpretation typically computes an inductive invariant or fixpoint over a program’s control-flow semantics. ZEBRA instead analyzes a bounded, already-extracted zkVM constraint instance and uses three-valued interval evaluation as a sound pruning oracle inside a finite branch-and-bound search. Its output is a cardinality classification of canonical satisfying trace tables, rather than a reachability invariant. This setting additionally requires sound treatment of finite-field wrap-around, selector-gated polynomial constraints, lookup and public-value constraints, and canonicalization of padding, permutation, and auxiliary-witness redundancy. Thus, our contribution is not interval abstraction itself, but its finite-field specialization and integration with canonicalized cardinality reasoning for zkVM constraints.

9 Limitations and Future Work

Global verification. ZEBRA’s correctness guarantees are table-localized: given a set of consistent zkVM tables, ZEBRA certifies the target table. Bugs from cross-table interactions, such as lookup arguments whose two sides are individually satisfied but whose interface is mis-specified, fall outside this guarantee. No existing automated technique verifies these cross-table interfaces. A fixed-point iteration over local certificates produced by ZEBRA can formally compose toward global correctness, analogous to layerwise bound propagation in neural network verification [26] albeit at greater computational cost. Reducing this cost via smarter branch-and-bound is a natural direction for future work.

Precision loss from interval abstraction. Interval abstraction is sound but incomplete: over-approximation may merge feasible and infeasible assignments and therefore return Uncertain even when the concrete constraint system has a definite answer. This imprecision cannot produce a false certificate or a spurious bug report, because ZEBRA prunes a region only when it is definitely infeasible and reports an under-constrained bug only after constructing two distinct concrete satisfying traces. However, it may force additional splitting and, under a finite time budget, reduce verification coverage or delay the discovery of a second witness in under-constrained cases. Tighter abstract domains or a hybrid interval-SMT procedure could address these residual cases.

Verifying Canonicalizer Implementation. While the canonicalizer algorithm is formally verified in Lean 4, verifying the equivalence between the Lean specification and the actual Rust implementation is left for future work.

Verifying non-Instruction Table. We exclude the non-instruction table from exhaustive verification in our current prototype due to its high complexity; scaling ZEBRA to cover it through advanced heuristics such as opcode-based slicing is an interesting direction for future work.

10 Conclusion

We presented ZEBRA, a fully automated framework for testing and verifying zkVM constraint systems. We introduced a formalization of zkVM vulnerabilities that separates representational redundancy

from true under- and over-constrained behaviors, and developed an interval-based branch-and-bound engine for reasoning about sets of executions.

Across five real-world zkVMs, ZEBRA uncovers 11 previously unknown bugs, including vulnerabilities enabling control-flow hijacking and forged proofs, all missed by a state-of-the-art fuzzer. Compared to SMT-based verification, ZEBRA achieves higher coverage (+16.5 pp) and a 51.5× speedup, while input range verification provides up to 63× efficiency gains over point-wise testing.

These results suggest that reasoning about solution-set cardinality via interval-based branch-and-bound provides a practical and scalable foundation for zkVM constraint verification.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful feedback. We would also like to thank Tim Roughgarden and Mahimna Kelkar at The Columbia-Ethereum Research Center for Blockchain Protocol Design for constructive and helpful discussions. Hideaki is supported by Funai Fellowship and Columbia-Ethereum Center Fellowship. Junfeng is also supported by DARPA ScAN, NSF CNS 2526620, Google, Samsung, Amazon, and Columbia CAIFI (with Capital One), Enterprise AI (with Infosys), and CDFT centers.

References

- [1] Argument Computer. [n. d.]. Sphinx. <https://github.com/argumentcomputer/sphinx>. Accessed: 2026-04-21.
- [2] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. 2022. A verified algebraic representation of cairo program execution. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 153–165.
- [3] Aztec Network. 2024. Noir: The Universal Language of Zero-Knowledge. <https://noir-lang.org>. Accessed: February 23, 2025.
- [4] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 4733–4751.
- [5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive* (2018).
- [6] Wyatt Benno, Alberto Centelles, Antoine Douchet, and Khalil Gibran. 2026. Jolt Atlas: Verifiable Inference via Lookup Arguments in Zero Knowledge. *arXiv preprint arXiv:2602.17452* (2026).
- [7] Brevis Network. [n. d.]. Pico. <https://github.com/brevis-network/pico>. Accessed: 2026-04-21.
- [8] OpenVM Contributors. 2025. *OpenVM Whitepaper*. Whitepaper. OpenVM. <https://openvm.dev/whitepaper.pdf>
- [9] Tim Dokchitser and Alexandr Bulkin. 2023. Zero knowledge virtual machine step by step. *Cryptology ePrint Archive* (2023).
- [10] Antonio J Fernández and Patricia M Hill. 2006. An interval constraint branching scheme for lattice domains. *J. Univers. Comput. Sci.* 12, 11 (2006), 1466–1499.
- [11] Galois, Inc. [n. d.]. zkLean. <https://github.com/GaloisInc/zkLean>. Accessed 2026-04-23.
- [12] Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo—a Turing-complete STARK-friendly CPU architecture. *Cryptology ePrint Archive* (2021).
- [13] Ulrich Haböck. 2022. Multivariate lookups based on logarithmic derivatives. *Cryptology ePrint Archive* (2022).
- [14] Yahya Hassanzadeh-Nazarabadi and Sanaz Taheri-Boshrooyeh. 2025. Constraint-Level Design of zkEVMs: Architectures, Trade-offs, and Evolution. *arXiv preprint arXiv:2510.05376* (2025).
- [15] Christoph Hochrainer, Valentin Wüstholtz, and Maria Christakis. 2025. Arguzz: Testing zkvm for soundness and completeness bugs. *arXiv preprint arXiv:2509.10819* (2025).
- [16] Jingyu Ke, Boxuan Liang, and Guoqiang Li. 2025. Consistency Verification for Zero-Knowledge Virtual Machine on Circuit-Irrelevant Representation. *Cryptology ePrint Archive* (2025).
- [17] Carl Kwan, Quang Dao, and Justin Thaler. 2024. Verifying jolt zkVM lookup semantics. *Cryptology ePrint Archive* (2024).

- [18] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. 2024. A Survey on the Applications of Zero-Knowledge Proofs. *arXiv preprint arXiv:2408.00243* (2024).
- [19] Lita Foundation. [n. d.]. Valida VM. <https://github.com/lita-xyz/valida-vm>. Accessed: 2026-04-21.
- [20] Shankara Pailoor. 2025. Verifying SP1 circuit determinism with Picus: A collaboration between Veridise and Succinct. <https://veridise.com/blog/audit-insights/verifying-sp1-circuit-determinism-with-picus-a-collaboration-between-veridise-and-succinct/>. Accessed 2026-04-23.
- [21] Shankara Pailoor. 2025. Verifying Ziren's circuits with Picus: Another step towards deterministic zkVMs. <https://audithub.dev/blog/picus/verifying-zirens-circuits-with-picus-another-step-towards-deterministic-zkvm/>. Accessed 2026-04-23.
- [22] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. 2023. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1510–1532.
- [23] ProjectZKM. [n. d.]. Ziren. <https://github.com/ProjectZKM/Ziren>. Accessed: 2026-04-21.
- [24] RISC Zero. 2025. Security Disclosure: A missing constraint was recently discovered in the rv32im circuit. <https://x.com/RiscZero/status/1935404812146725042>. RISC-V instructions in risc0-zkvm versions 2.0.0–2.0.2 were affected; patch released in version 2.1.0.
- [25] Javier Ron and Martin Monperrus. 2026. Verifiable Provenance of Software Artifacts with Zero-Knowledge Compilation. *arXiv preprint arXiv:2602.11887* (2026).
- [26] Zhouxing Shi, Qirui Jin, J. Zico Kolter, Suman Jana, Cho-Jui Hsieh, and Huan Zhang. 2025. Neural Network Verification with Branch-and-Bound for General Nonlinearities. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer.
- [27] Succinct Labs. 2026. SP1. <https://github.com/succinctlabs/sp1>. Accessed: 2026-04-21.
- [28] Hideaki Takahashi, Jihwan Kim, Suman Jana, and Junfeng Yang. 2025. zkFuzz: Foundation and Framework for Effective Fuzzing of Zero-Knowledge Circuits. *arXiv preprint arXiv:2504.11961* (2025).
- [29] Morgan Thomas, Mamy Ratsimbazafy, Marcin Bugaj, Lewis Revill, Carlo Modica, Sebastian Schmidt, Ventali Tan, Daniel Lubarov, Max Gillett, and Wei Dai. 2025. Valida ISA Spec, version 1.0: A zk-Optimized Instruction Set Architecture. *arXiv preprint arXiv:2505.08114* (2025).
- [30] Valida. [n. d.]. Valida (Old Version). <https://github.com/valida-xyz/valida>. Accessed: 2026-04-21.
- [31] Verified-zkEVM. [n. d.]. Clean: A verified zkEVM. <https://github.com/Verified-zkEVM/clean/>. Accessed 2026-04-23.
- [32] Lizheng Wang, Hancheng Lou, Chongrong Li, Yu Yu, and Yuncong Hu. 2026. zkAgent: Verifiable Agent Execution via One-Shot Complete LLM Inference Proof. *Cryptology ePrint Archive* (2026).
- [33] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. 2024. Practical Security Analysis of {Zero-Knowledge} Proof Circuits. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1471–1487.

11 Ethical Considerations

We responsibly disclosed all previously unknown vulnerabilities discovered in public projects. We first contacted the maintainers privately and created public GitHub issues only with their approval. We also worked closely with each team to assess the severity of the issues and support the development of appropriate fixes.