Consistency Verification for Zero-Knowledge Virtual Machines on Circuit-Irrelevant Representation

Jingyu Ke[1], Boxuan Liang[2], and Guoqiang Li[1]

[1] Shanghai Jiao Tong University, Shanghai, China
{windocotber, li.g}@sjtu.edu.cn
[2] Taiyuan University of Technology, Taiyuan, China
liangboxuan7762@link.tyut.edu.cn

Abstract. Zero-knowledge virtual machines (zkVMs) rely on tabular constraint systems whose verification semantics include gate, lookup, and permutation relations, making correctness auditing substantially more challenging than in arithmetic-circuit DSLs such as Circom. In practice, ensuring that witness-generation code is consistent with these constraints has become a major source of subtle and hard-to-detect bugs. To address this problem, we introduce a high-level semantic model for tabular constraint systems that provides a uniform, circuit-irrelevant interpretation of row-wise constraints and their logical interactions. This abstraction enables an inductive, row-indexed reasoning principle that checks consistency without expanding the full circuit, significantly improving scalability. We implement this methodology in *ZIVER* and show that it faithfully captures real zkVM designs and automatically validates the consistency of diverse SP1 chip components.

Keywords: Zero-Knowledge Virtual Machines · Consistency Verification · Symbolic Execution.

1 Introduction

Zero-knowledge proofs (ZKPs) [10] allow a prover to convince a verifier that a computation was executed correctly without revealing any information about the underlying inputs. In typical ZKP systems, the target computation is first modeled as an arithmetic circuit, which provides a structured intermediate representation for expressing program logic. This circuit is then encoded into a system of polynomial constraints together with a witness, an assignment to the circuit's private variables that satisfies them. Building on this circuit-based abstraction, ZKPs support applications ranging from trusted identity authentication [8] and privacy-preserving regulation [6] to scalable blockchain infrastructures and privacy-preserving data sharing [22].

As the scope of ZKP applications expands toward general-purpose computation, there is a growing need for more programmable proof systems that reduce

reliance on manually crafted circuit-level encodings. *Zero-knowledge virtual machines (ZKVMs)* enhance the programmability of ZKPs by automatically translating a program's execution trace into polynomial constraints, thereby relieving developers from manually constructing circuit-level encodings.

This improvement in programmability, however, also introduces new risks. Audit reports [16,27,28] over the years have documented a wide range of exploitable issues in zkVMs such as RISC Zero [5], SP1 [25], and OpenVM [17], including problems in constraint logic, witness generation, and zero-knowledge proving backends. Although there are ongoing efforts [2,15] to verify zkVM implementations using theorem provers such as Lean and Rocq, these projects remain difficult to scale and maintain, since proofs often require substantial manual work and become brittle as zkVM designs evolve. This motivates the need for automated approaches that can detect vulnerabilities and assess correctness with minimal developer intervention.

Related automated verification efforts in the literature [29,24,20] have largely focused on Circom and its underlying Rank-1 Constraint System (R1CS), where computations are encoded as a sparse collection of algebraic constraints with limited structural organization. In contrast, zkVMs rely on tabular constraint systems that represent computations as structured matrices, where each constraint schema applies uniformly to every row of the trace. This encoding style is substantially more compact and imposes stronger consistency requirements across the computation. Systems such as SP1 and OpenVM use AIR-style tables, ZKWASM [9] adopts a Plonkish tableau with permutation and lookup relations, and Jolt [1] further embraces lookup-heavy designs. Despite these advances, the academic literature still lacks a suitable and efficient formal model that captures the essential structure of tabular constraint systems while enabling automated reasoning.

In this work, we propose a unified abstract semantics for tabular constraint systems and an inductive equivalence-checking algorithm that mitigates the scalability limitations of existing zkVM consistency analyses. Tabular constraint systems typically consist of two classes of operations: (1) row-wise constraints that describe the algebraic relations among the columns of each row (including cross-row offsets), and (2) global permutation constraints that equate multisets of column tuples across different logical tables and enforce global structural invariants. Our semantic model abstracts these constraints into a uniform representation in which both computation and constraint reasoning can be expressed within the same symbolic-execution framework.

A key insight is that zkVMs exhibit a structured, row-by-row correspondence between witness generation and constraint evaluation. Our inductive verification algorithm captures this structure directly: it symbolically checks the agreement between generator-side computation and constraint-side semantics one row at a time, and then lifts these row-local checks to obtain a global consistency guarantee over the entire constraint system. Both row-wise constraints and global permutation relations are interpreted through high-level semantic abstractions,

enabling the algorithm to reason modularly while retaining the expressiveness needed to model real zkVM designs.

To demonstrate practicality, we implement ZIVER, a lightweight consistency checker that instantiates our abstract model and verification algorithm. Applying ZIVER to the SP1 zkVM shows that the model is expressive, modular, and automation-friendly, and that it scales to realistic chip-level instruction sets.

In summary, this work makes three contributions: (1) formalize a unified abstract semantics for zkVM tabular constraint systems, covering both row-wise and global permutation constraints; (2) design an inductive consistency-verification algorithm that exploits row-aligned computation–constraint structure to enable scalable, modular reasoning; and (3) implement and open-source ZIVER[3], that the approach efficiently verifies a broad range of constraint and chip-level components.

2 Background

Zero-Knowledge Proofs. Zero-knowledge proof systems allow a prover to convince a verifier that a given computation has been carried out correctly, without revealing the private data involved in the computation. Modern proof systems achieve this by first expressing the computation as a system of polynomial constraints and then using polynomial-commitment schemes [13,21] and related cryptographic primitives to construct succinct proofs that can be checked efficiently.

More concretely, for a public input $x \in \mathbb{F}^{|\mathcal{I}|}$, the prover executes the computation on input x and obtains a private witness that represents the data arising during this execution. The prover then uses this private witness to generate a proof. The verifier, given only the public input and the proof, checks without learning the witness that the private witness satisfies all polynomial constraints induced by the computation.

To formalize this setting, we model the computation as an arithmetic circuit C over a finite field $\mathbb{F}$, with public input variables $\mathcal{I} = \{x_1, \ldots, x_n\}$ and private witness variables $\mathcal{W} = \{w_1, \ldots, w_m\}$. Each gate in the computation induces a polynomial relation among these variables, giving rise to a finite family

$$C = \{\, p_1(\mathcal{I}, \mathcal{W}), \ldots, p_k(\mathcal{I}, \mathcal{W}) \,\} \subseteq \mathbb{F}[\mathcal{I}, \mathcal{W}].$$

An assignment $\sigma : \mathcal{I} \cup \mathcal{W} \to \mathbb{F}$ satisfies the circuit if

$$p_j(\sigma(\mathcal{I}),\, \sigma(\mathcal{W})) = 0 \text{ for all } p_j \in C.$$

In practice, the witness values $\sigma(\mathcal{W})$ are not arbitrary but arise from the concrete execution of the computation. A designated witness-generation procedure $\mathsf{Gen} : \mathbb{F}^{|\mathcal{I}|} \to \mathbb{F}^{|\mathcal{W}|}$ computes these private values from the public input, providing the operational semantics of the computation, while the polynomial system C specifies the algebraic conditions that any valid execution must satisfy.

[3] `https://github.com/WindOctober/ZIVER`

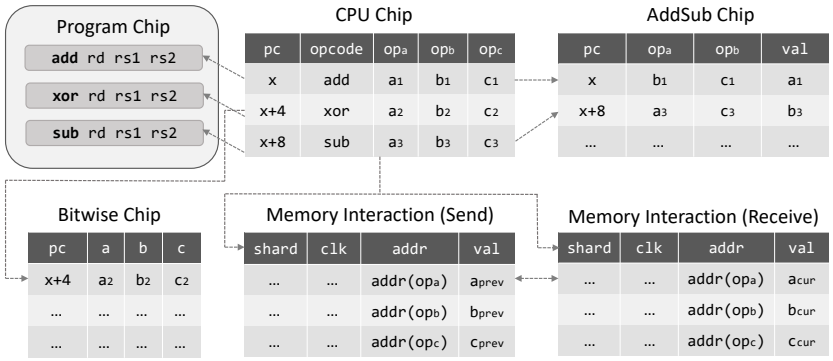

Fig. 1. Illustrative zkVM architecture: components emit witness tables; rounded nodes represent public traces or inputs; dashed arrows indicate lookup relations; double arrows indicate permutation relations.

Different arithmetization frameworks organize these polynomial constraints in different ways. Classical systems such as Rank-1 Constraint Systems (R1CS) represent each constraint as a bilinear polynomial equation; for completeness, we summarize the standard R1CS formulation in Appendix A. Modern zkVMs, in contrast, adopt trace-based, row-wise constraint systems that couple local gate relations with global consistency conditions across the entire execution trace.

Zero-Knowledge Virtual Machines. In contemporary proof systems, a zero-knowledge virtual machine (zkVM) executes a low-level program, typically obtained by compiling high-level code to an instruction set such as RISC–V, and records the resulting computation in a collection of structured tables. During execution, each instruction is expanded into a row that records the program counter, the opcode, the three register operands, any immediate value, and other instruction-specific fields. Logical components of the machine, including instruction decoding, CPU execution, arithmetic and bitwise units, and memory access logic, consume the relevant subset of these fields and record their local operational semantics in their respective tables.

Figure 1 illustrates this architecture. Relations between tables are expressed through lookup and permutation constraints, which provide a uniform mechanism for coordinating the information associated with each instruction. These constraints can enforce agreement on columns that uniquely identify a particular instruction instance, such as the program counter, the opcode, or register addresses, thereby ensuring that all components operate on a consistent view of the instruction stream. This form of cross-table coordination induces a symbolic data flow among logical components, functioning as a soft bus that propagates instruction information throughout the system and enables each table to contribute its local semantics while maintaining global coherence.

3 Overview

We illustrate our setting using the consistency between the computation of a register access register[addr] and the corresponding tabular constraints in SP1. Figure 2 shows a simplified fragment of the `eval_memory_access` routine of the CpuChip. At the computational level, each read or write to a logical memory location is recorded together with an explicit timestamp. These timestamps are not mutable machine state; instead, they are written directly into the execution trace as dedicated columns. For each access to address addr, the trace stores both the timestamp of the most recent prior access $\mathsf{clk}_{\mathsf{prev}}$ and the timestamp of the current access $\mathsf{clk}_{\mathsf{cur}}$, and the constraint in Figure 2 enforces the local ordering relation $\mathsf{clk}_{\mathsf{cur}} > \mathsf{clk}_{\mathsf{prev}}$.

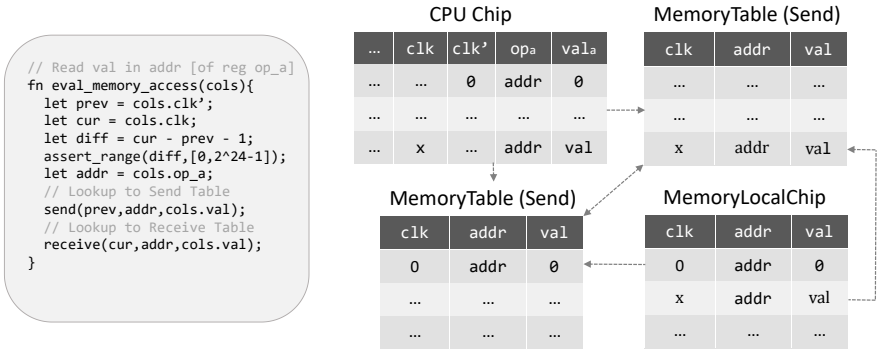

Fig. 2. A simplified illustration of the eval_memory_access procedure in SP1. The diagram on the right depicts the associated logical-table constraint structure; the meaning of arrows follows that of Figure 1. The labels *send* and *receive* indicate, respectively, that the tuple formed by the designated columns of the current row is looked up in, or supplied by, the corresponding logical table.

The right side of Figure 2 shows the tabular organization implementing this mechanism. The CpuChip table contains operational columns such as addr, $\mathsf{clk}_{\mathsf{prev}}$, $\mathsf{clk}_{\mathsf{cur}}$, $\mathsf{val}_{\mathsf{prev}}$, and $\mathsf{val}_{\mathsf{cur}}$. Two auxiliary tables, Send and Receive, encode the "previous" and "current" memory-access events; and a global permutation constraint requires the two multisets of tuples to coincide. This enforces that each read observes the value of the most recent prior write to the same address and that no spurious events can arise. A separate MemoryLocalChip exposes the initial and final memory states as public tables and participates in the same permutation argument.

This construction is an instance of the classical offline memory-checking paradigm: logical memory accesses are represented as timestamped events, and correctness is ensured by requiring that the multisets of "send" and "receive"

events match, together with per-address monotonicity of timestamps and explicit initialization/finalization of memory. A full correctness proof showing that these permutation-based conditions are equivalent to the standard `Map`-based operational semantics appears in [4], and SP1 adopts precisely this proven construction.

Beyond memory, zkVM constraint systems rely on lookup and permutation constraints in similarly disciplined ways. Lookup constraints connect a chip's execution to fixed public tables such as instruction encodings or nonlinear operation tables (e.g., bit decomposition or range checks). Permutation constraints act as inter-chip buses that propagate operational invariants across heterogeneous components, and they mediate access to dynamically populated tables whose entries are produced during witness generation—such as the memory example above.

Guided by these observations, we adopt a high-level semantic abstraction of tabular constraint systems. Rather than encoding lookup or permutation constraints directly as first-order formulas, we interpret them according to their intended operational meaning: propagating invariants across tables, interpreting lookups as their generating computations, and relating dynamically produced table entries to the behavior of the underlying operations.

Built on top of this abstraction, our verification procedure checks the consistency between each component's computational semantics and its associated tabular constraints. When an inconsistency is detected, we construct an input program that triggers the mismatch; if the resulting counterexample is infeasible, the abstraction is refined. This iterative structure enables systematic reasoning about lookup and permutation constraints at the semantic level, without table expansion or quantifiers.

4 Technique

This section develops the formal framework underlying our approach to zkVM consistency checking. We begin by introducing the tabular constraint model used to represent the algebraic semantics of a zkVM, together with a generic DSL for describing witness generation and constraint schemas. We then give a precise definition of a zkVM program and formalize what it means for the execution of the witness generator to be consistent with the tabular constraints. Finally, we introduce a symbolic execution framework that embeds both computation and constraints into a uniform symbolic state, enabling automated consistency checking.

4.1 Consistency Model

DSL for Generation and Constraints. We use a DSL to specify the witness-generation procedures and the constraint schemas associated with each logical table. Figure 3 summarizes its syntax. All expressions range over the field $\mathbb{F}$, and all operators are interpreted as field operations. The generator side describes how witness values are computed from public inputs, while the constraint side

$$
\begin{aligned}
\mathsf{Gen} ::= & \ \textbf{for}\ i\ \textbf{in}\ 0..N\{S\} \\
S ::= & \ V \leftarrow E \mid S; S \\
& \mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \\
E ::= & \ E \oplus E \mid V \mid \text{const} \\
V ::= & \ \mathrm{m}[E_1, \ldots, E_k] \mid r^{(\delta)}[c] \mid v
\end{aligned}
$$

(a) Syntax of Gen

$$
\begin{aligned}
\mathsf{C} ::= & \ \textbf{for}\ i\ \textbf{in}\ 0..N\{S\} \\
S ::= & \ S; S \mid E == E \mid (E_1, \ldots, E_k) \in T_i \\
& \mid (T_i[C_i] = T_j[C_j]) \\
E ::= & \ E + E \mid E \times E \mid r^{(\delta)}[c] \mid \text{const}
\end{aligned}
$$

(b) Syntax of constraints C

Fig. 3. DSL syntax for the witness generator (a) and the constraint system (b). In $(T_i[C_i] = T_j[C_j])$, each C denotes a list of column indices, and $T[C]$ denotes the multiset of tuples formed by collecting all rows restricted to these columns.

specifies both the row-wise and global conditions that each logical table must satisfy.

Tabular Constraint Systems. A tabular constraint system consists of a finite collection of logical tables. Each table T is represented as a matrix $\mathcal{G}_{\mathsf{T}} \in \mathbb{F}^{N_{\mathsf{T}} \times M_{\mathsf{T}}}$.[4] For any row l and column c, we write $r^{(\delta)}_{\mathsf{T}}[c] := \mathcal{G}_{\mathsf{T}}[\, l + \delta \bmod N_{\mathsf{T}},\, c\,]$,[5] reflecting the cyclic evaluation domain commonly used in polynomial IOPs. Whenever multiple logical tables share a physical column in the underlying arithmetization, their corresponding entries are interpreted as aliases of the same field-valued column.

The constraint language consists of three classes of constraints:

- *Gate constraints* $E == E$, which compare two field expressions evaluated at the current row (or at offset rows). These express the row-local algebraic relations of the zkVM.
- *Lookup constraints* $(E_1, \ldots, E_k) \in T_i$, which require the tuple of field expressions at the current row to match *some* row of the target table T_i. These constraints specify finite relations such as opcode decoding or range checks.
- *Permutation constraints* $(T_i[C_i] = T_j[C_j])$, where each C is a list of column indices. For a table T and a column list $C = (c_1, \ldots, c_m)$, the notation $T[C]$ denotes the multiset of tuples

$$(T[l, c_1], \ldots, T[l, c_m]) \quad \text{for all rows } l.$$

  A permutation constraint requires $|C_i| = |C_j|$ and that the multisets $T_i[C_i]$ and $T_j[C_j]$ coincide, thereby enforcing global consistency between the two column sets.

[4] In concrete arithmetizations, these logical tables are embedded into one physical trace with selector columns; this realization detail does not affect the logical view adopted here.

[5] This convention reflects the cyclic nature of the underlying evaluation domain (e.g., a multiplicative subgroup).

A table assignment satisfies the tabular constraint system if all gate, lookup, and permutation constraints evaluate to true under the induced interpretation of row expressions and multisets.

Definition 1 (zkVM Program Model). *A zkVM program is represented as a tuple* $P = (\mathcal{T}, \mathcal{X}, \mathcal{M}, \mathsf{Gen}, \mathsf{C}, \mathit{\Pi})$*, where:*

- $\mathcal{T}$ *is the set of logical tables, each equipped with its own row structure* $\mathcal{G}_T$ *and constraint schemas.*
- $\mathcal{X}$ *is the set of scalar variables, consisting of all intermediate variables* v *together with all table entries* $\mathcal{G}_T[l, c]$ *for* $T \in \mathcal{T}$*. For each table* T*, we write* $\mathcal{X}_T$ *for its associated variables and partition them as* $\mathrm{Pub}(\mathcal{X}_T)$ *and* $\mathrm{Wit}(\mathcal{X}_T)$*, denoting respectively public inputs and witness values. The global domains are*
$$\mathrm{Pub}(\mathcal{X}) = \bigcup_{T \in \mathcal{T}} \mathrm{Pub}(\mathcal{X}_T), \qquad \mathrm{Wit}(\mathcal{X}) = \bigcup_{T \in \mathcal{T}} \mathrm{Wit}(\mathcal{X}_T).$$
- $\mathcal{M}$ *is the domain of map variables representing the logical memory structures manipulated by the witness generator.*
- Gen *is the family of witness-generation procedures. For each* T*, the generator is a function*
$$\mathsf{Gen}_T : \mathbb{F}^{|\mathrm{Pub}(\mathcal{X}_T)|} \to \mathbb{F}^{|\mathrm{Wit}(\mathcal{X}_T)|},$$
and each witness variable is assigned exactly once.
- C *is the collection of all gate and lookup constraints, and* C^S *denotes the corresponding set of row-wise constraint schemas. For each table* T*, we write* $\mathsf{C}^S_T \subseteq \mathsf{C}^S$ *for the schemas governing the constraints applied to row* i *of* T*.*
- $\mathit{\Pi}$ *is the set of permutation constraints relating multisets of column tuples across tables.*

Given an assignment $\sigma : \mathcal{X} \to \mathbb{F}$, let $[\![\mathsf{C}]\!](\sigma)$ and $[\![\mathit{\Pi}]\!](\sigma)$ denote the truth values obtained by instantiating all row-wise and permutation constraints under σ. The assignment σ satisfies the zkVM program if both

$$[\![\mathsf{C}]\!](\sigma) = \mathrm{true}, \qquad [\![\mathit{\Pi}]\!](\sigma) = \mathrm{true}.$$

Definition 2 (Consistency of a zkVM Program). *A zkVM program* P *is* consistent *if for every public assignment* σ_{pub} *and every witness assignment* σ_{wit}*, letting* $\sigma = \sigma_{\mathsf{pub}} \cup \sigma_{\mathsf{wit}}$*, we have*

$$\Big([\![\mathsf{C}]\!](\sigma) \ \wedge \ [\![\mathit{\Pi}]\!](\sigma)\Big) \iff \Big(\bigwedge_{T \in \mathcal{T}} \forall x \in \mathrm{Wit}(\mathcal{X}_T).\ \sigma_{\mathsf{wit}}(x) = \mathsf{Gen}_T\big(\sigma_{\mathsf{pub}}(\mathcal{X}_T)\big)(x)\Big).$$

Here we abuse notation and write $\mathsf{Gen}_T(\sigma_{\mathsf{pub}}(\mathcal{X}_T))$ *for the witness assignment produced by the generator* Gen_T *under the given public inputs* σ_{pub}*.*

Intuitively, for each public input, the tabular constraint system admits *exactly one* witness assignment, and this assignment must coincide with the witness produced by the generator side of the DSL.

4.2 High-Level Semantic Modeling

To compare the execution semantics of Gen with the tabular constraints (C, Π) within a unified reasoning framework, we introduce a symbolic execution model that provides a single logical interpretation for computational steps, row-wise constraints, and lookup and permutation relations.

We first define symbolic expressions by the grammar

$$e ::= v \mid c \mid (e\ \mathsf{op}\ e),$$

where v ranges over symbolic variables introduced during initialization, c ranges over field constants, and op ranges over the usual binary operators. Let $\mathcal{E}$ denote the set of all expressions generated by this grammar. Boolean expressions form the subset

$$\mathcal{B} = \{\, e_1\ \mathsf{cmp}\ e_2 \mid e_1, e_2 \in \mathcal{E},\ \mathsf{cmp} \in \{=, \neq, <, \leq\} \,\}.$$

With this expression language in place, we now define the symbolic state, which captures symbolic variable assignments, accumulated path conditions, and the structure of logical memories.

Definition 3 (Symbolic State). *A symbolic state is a triple*

$$\Sigma = (\sigma, \pi, \mu),$$

where:

1. $\sigma : \mathcal{X} \to \mathcal{E}$ *is the symbolic store mapping each variable to a symbolic expression. Initially, for every* $x \in \mathrm{Pub}(\mathcal{X})$ *the store assigns a fresh symbolic variable* $\sigma(x) = \mathsf{sym}(x)$*, while all witness variables remain unassigned.*
2. $\pi \subseteq \mathcal{B}$ *is the current path condition, initially* $\pi = \emptyset$*.*
3. $\mu \subseteq \mathcal{P}(\mathcal{E}^* \times \mathcal{E}^* \times \mathcal{E})$ *is the global symbolic memory table, where* $\mathcal{E}^*$ *denotes the set of finite tuples of symbolic expressions. Each entry of* μ *is a read/write event of the form* (τ, k, v)*, with* $\tau, k \in \mathcal{E}^*$ *representing (possibly composite) timestamp and key tuples, and* $v \in \mathcal{E}$ *the associated symbolic value. All mapping variables share this global event table, and initially* μ *is empty.*

Expression evaluation under a symbolic state is formalized using the judgement

$$\Sigma \vdash E \Downarrow e,$$

which states that the DSL expression E evaluates to the symbolic expression $e \in \mathcal{E}$ at the current row i of table T when interpreted under the symbolic state $\Sigma = (\sigma, \pi, \mu)$. The complete set of rules for $\Sigma \vdash E \Downarrow e$ is shown in Figure 4. We assume throughout that all map variables read in Gen have been initialized in some logical table and that intermediate variables are assigned before being used.

$$\frac{c \in \mathbb{F}}{\Sigma \vdash c \Downarrow c}\ (\textsc{Const}) \qquad \frac{v \in \mathcal{X}_T \qquad \sigma(v) = e}{\Sigma \vdash v \Downarrow e}\ (\textsc{Var})$$

$$\frac{r_T^{(\delta)}[c] = \mathcal{G}_T[i + \delta \bmod N_T,\, c] \qquad \sigma(\mathcal{G}_T[i + \delta \bmod N_T,\, c]) = e}{\Sigma \vdash r_T^{(\delta)}[c] \Downarrow e}\ (\textsc{Trace})$$

$$\frac{\Sigma \vdash E_1 \Downarrow e_1 \qquad \Sigma \vdash E_2 \Downarrow e_2}{\Sigma \vdash (E_1\ \mathsf{op}\ E_2) \Downarrow (e_1\ \mathsf{op}\ e_2)}\ (\textsc{BinOp})$$

$$\frac{\Sigma \vdash E_j \Downarrow e_j\ (\forall j.\ 1 \le j \le k) \qquad \begin{array}{c}\mathsf{IsTimestamp}(E_1, \ldots, E_i) \\ \mathsf{Read}\big(\mu, (e_{i+1}, \ldots, e_k)\big) = v\end{array}}{\Sigma \vdash m[E_1, \ldots, E_k] \Downarrow v}\ (\textsc{Map-Read})$$

Fig. 4. Evaluation rules for DSL expressions under a symbolic state Σ. In rule Map-Read, the predicate $\mathsf{IsTimestamp}(E_1, \ldots, E_i)$ identifies the prefix of the access tuple that corresponds to the timestamp component τ of an event (τ, k, v) in the global memory table μ, while $\mathsf{Read}\big(\mu, (e_{i+1}, \ldots, e_k)\big) = v$ retrieves the value v of the most recent event whose key tuple k matches $(e_{i+1}, \ldots, e_k)$.

Symbolic environments and statement semantics. Let **State** denote the set of symbolic states $\Sigma = (\sigma, \pi, \mu)$ as defined in Definition 3. A *symbolic environment* is a finite set $\mathcal{S} \subseteq \mathbf{State}$, initialized as $\mathcal{S}_0 = \{\Sigma_0\}$, where Σ_0 is the initial symbolic state.

To interpret the control-flow constructs in Figure 3, we formalize statement execution at the current row i of table T using the big-step judgment

$$\Sigma \vdash S \Rightarrow \mathcal{S},$$

which states that, under symbolic state Σ, executing the statement S for this row may produce any of the successor states in $\mathcal{S}$. This semantics is lifted point-wise to environments by taking the union over all states in the input environment. The complete set of rules for $\Sigma \vdash S \Rightarrow \mathcal{S}$ is shown in Figure 5.

We present here only the semantic rules specific to tabular constraint systems. The complete set of statement-execution rules is given in Appendix B. Below we brief the auxiliary predicates that appear in the rules.

$\mathsf{RowMatch}_{T'}(c_1, \ldots, c_k)$ Extracts the tuple $(r_{T'}^{(0)}[c_1], \ldots, r_{T'}^{(0)}[c_k])$ from the row pattern of T'. A one-step symbolic execution of this row pattern under the initial state $\Sigma_{T'}^0$ yields a formula $\Phi_{T'}(z_1, \ldots, z_k)$, whose free variables correspond exactly to these row elements. This formula $\Phi_{T'}$ is the conjunction of all row-wise constraints in $\mathsf{C}_{T'}^S$, expressed over the symbolic placeholders $z_1, \ldots, z_k$.

In the Lookup rule, the sets π' are obtained by instantiating $\Phi_{T'}$ with $z_j := e_j$, where e_j is the symbolic evaluation of E_j. To avoid infinite unfolding, lookup constraints form directed edges between tables, and lookup-connected components (via their transitive closure) reuse the union of all Φ_T formulas from tables in the same component, instead of recursively expanding them.

$$\frac{\Sigma_1 = (\sigma, \pi \cup \{e\}, \mu) \quad \Sigma_2 = (\sigma, \pi \cup \{\neg e\}, \mu) \quad \Sigma \vdash E \Downarrow e \qquad \Sigma_1 \vdash S_1 \Rightarrow \mathcal{S}_1 \qquad \Sigma_2 \vdash S_2 \Rightarrow \mathcal{S}_2}{\Sigma \vdash \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \Rightarrow \mathcal{S}_1 \cup \mathcal{S}_2} \text{ (If)}$$

$$[\![S]\!](\mathcal{S}) \;=\; \bigcup_{\Sigma \in \mathcal{S}} \{\, \Sigma' \mid \Sigma \vdash S \Rightarrow \mathcal{S}',\ \Sigma' \in \mathcal{S}' \,\} \text{ (Env-Lift)}$$

$$\frac{\Sigma \vdash E_j \Downarrow e_j\ (1 \le j \le k) \qquad \Sigma^0_{T'} \vdash \mathsf{C}^S_{T'} \Rightarrow \mathcal{S}_{T'} \quad \Pi^{\mathsf{lookup}} = \{\, \pi'[\mathsf{RowMatch}_{T'}(c_1, \ldots, c_k) \mapsto (e_1, \ldots, e_k)] \mid (\sigma', \pi', \mu') \in \mathcal{S}_{T'} \,\}}{\Sigma \vdash (E_1, \ldots, E_k) \in T' \Rightarrow \{\, (\sigma,\ \pi \cup \pi',\ \mu) \mid \pi' \in \Pi^{\mathsf{lookup}} \,\}} \text{ (Lookup)}$$

$$\frac{(c_1, \ldots, c_m) = C_i \qquad (d_1, \ldots, d_m) = C'_j \quad \Sigma \vdash r^{(0)}_T[c_\ell] \Downarrow e_\ell\ (1 \le \ell \le m) \qquad \Sigma^0_{T'} \vdash \mathsf{C}^S_{T'} \Rightarrow \mathcal{S}_{T'} \quad \Pi^{\mathsf{perm}} = \{\, \pi'[\mathsf{RowMatch}_{T'}(d_1, \ldots, d_m) \mapsto (e_1, \ldots, e_m)] \mid (\sigma', \pi', \mu') \in \mathcal{S}_{T'} \,\}}{\Sigma \vdash T[C_i] = T'[C'_j] \Rightarrow \{\, (\sigma,\ \pi \cup \pi',\ \mu) \mid \pi' \in \Pi^{\mathsf{perm}} \,\}} \text{ (Perm)}$$

$$\frac{\Sigma \vdash E_j \Downarrow e_j\ (\forall j.\ 1 \le j \le k) \qquad \mathsf{Public}(T')}{\Sigma \vdash (E_1, \ldots, E_k) \in T' \Rightarrow \{\, (\sigma,\ \pi \cup \{\mathsf{LookupPred}_{T'}(e_1, \ldots, e_k)\},\ \mu) \,\}} \text{ (Lookup-Pub)}$$

$$\frac{\Sigma \vdash E_j \Downarrow e_j\ (1 \le j \le k) \qquad \Sigma \vdash E \Downarrow e_v \quad \mathsf{IsTimestamp}(E_1, \ldots, E_i) \quad \tau = (e_1, \ldots, e_i) \quad k = (e_{i+1}, \ldots, e_k)}{\Sigma \vdash \mathsf{m}[E_1, \ldots, E_k] \leftarrow E \Rightarrow \{\, (\sigma, \pi,\ \mu \cup \{(\tau, k, e_v)\}) \,\}} \text{ (Mem-Write)}$$

$$\frac{(E'_1, \ldots, E'_i, E_{i+1}, \ldots, E_{k-1}, E'_k) \in T' \quad \Sigma \vdash E_j \Downarrow e_j\ (1 \le j \le k) \quad \Sigma \vdash E'_j \Downarrow e'_j\ (1 \le j \le i) \quad \tau = (e_1, \ldots, e_i) \quad k = (e_{i+1}, \ldots, e_{k-1}) \quad (e'_1, \ldots, e'_i) \le (e_1, \ldots, e_i) \quad \mathsf{Init}(T) \qquad \mathsf{Final}(T') \qquad \mathsf{IsTimestamp}(E_1, \ldots, E_i) \qquad \mathsf{Perm}(T, T')}{\Sigma \vdash (E_1, \ldots, E_k) \in T \Rightarrow \{\, (\sigma, \pi,\ \mu \cup \{(\tau, k, e_k)\}) \,\}} \text{ (Read-Write)}$$

Fig. 5. Big-step semantics for statements.

Public(T') Indicates that $\mathcal{X}_{T'} \subseteq \mathrm{Pub}(\mathcal{X})$. In this case, the internal structure of T' is summarized by a built-in predicate $\mathsf{LookupPred}_{T'}(e_1, \ldots, e_k)$ extracted once from its known public rows.

Perm(T, T') Indicates that the designated column sets C_T and $C_{T'}$ have the same cardinality and that the permutation relation $T[C_T] = T'[C_{T'}]$ holds.

Init(T)**, Final**(T') Express that, for any mapping variable governed by table T, its initial value is correctly constrained and made public by the constraint system. Similarly, $\mathsf{Final}(T')$ ensures that the final access of each mapping variable tracked by T' is also correctly constrained and public.

Remark. In practice, a pair of tables T and T' typically corresponds to a single logical mapping variable. When multiple mapping variables coexist, an extended definition of the event trace μ can be used to distinguish them, but this association often relies on external oracles that specify which table corresponds to each mapping variable.

Algorithm 1 Inductive Consistency Checking for zkVM Programs

Require: zkVM program P
Ensure: ✓ if consistent, ✗ if inconsistent
1: $\Gamma \leftarrow \{\Sigma_0\}$
2: **for** $T \in \mathcal{T}$ **do**
3: $\quad \Gamma_1 \leftarrow \mathsf{Exec}(\Gamma, \mathsf{Gen}_T^S)$
4: $\quad \Gamma_2 \leftarrow \mathsf{Exec}(\Gamma, \mathsf{C}_T^S \cup \Pi)$
5: $\quad$ **for** each pair $(\Sigma_1, \Sigma_2) \in \mathrm{Zip}(\Gamma_1, \Gamma_2)$ **do**
6: $\quad\quad$ Let $\Sigma_i = (\sigma_i, \pi_i, \mu_i)$ for $i \in \{1, 2\}$
7: $\quad\quad \phi_{\mathsf{path}} \leftarrow \pi_1 \wedge \pi_2$
8: $\quad\quad \phi_{\mathsf{io}} \leftarrow \left(\bigwedge_{x \in \mathrm{Pub}(\mathcal{X}_T)} \sigma_1(x) = \sigma_2(x)\right) \wedge \left(\bigvee_{y \in \mathrm{Wit}(\mathcal{X}_T)} \sigma_1(y) \neq \sigma_2(y)\right)$
9: $\quad\quad \phi_\mu \leftarrow (\mu_1 = \mu_2)$
10: $\quad\quad (\mathsf{res}, m) \leftarrow \textsc{Query}(\phi_{\mathsf{path}} \wedge \phi_{\mathsf{io}} \wedge \phi_\mu)$
11: $\quad\quad$ **if** $\mathsf{res} = \mathsf{sat}$ **then**
12: $\quad\quad\quad$ **if** $\textsc{ValidateConcreteAssignment}(m, P)$ **then return** ✗
13: $\quad\quad\quad \phi_{\mathrm{full}} \leftarrow \textsc{ReconstructFullConstraints}(P)$
14: $\quad\quad\quad \mathsf{res}' \leftarrow \textsc{Query}(\phi_{\mathrm{full}})$
15: $\quad\quad\quad$ **if** $\mathsf{res}' = \mathsf{sat}$ **then return** ✗
16: **if** $\textsc{ValidateConcreteAssignment}(\Sigma_0, P)$ **then return** ✓
17: **return** ✗

4.3 Inductive Consistency-Checking Algorithm

Algorithm 1 presents our inductive procedure for checking whether a zkVM program P is consistent in the sense of Definition 2. For each table $T \in \mathcal{T}$, the algorithm symbolically executes one row of the generator semantics Gen_T^S and one row of the constraint semantics $\mathsf{C}_T^S \cup \Pi$ under the current symbolic environment, obtaining two successor environments. The resulting symbolic states are then paired, and for each pair the algorithm builds a verification condition that captures any possible mismatch between generator-side and constraint-side behavior (Line 10).

The key intuition is that the abstract generator semantics is exact with respect to the concrete execution of P, whereas the abstract constraint semantics over-approximates the concrete constraint system (C, Π). Consequently, SMT queries may produce *spurious* counterexamples. Whenever a model is found, the algorithm first validates it concretely by running P on the induced witness assignment (Line 12); if this validation fails, the model is discarded as an artifact of the abstraction.

To further remove spurious behaviors, the algorithm refines lookup and permutation semantics by expanding them into their first-order logical form and fully materializing the relevant table entries (Line 13). If the refined constraints still admit the counterexample, the inconsistency is genuine and the algorithm terminates with ✗.

Finally, because C_T^S over-approximates the concrete constraints, symbolic consistency alone does not guarantee concrete consistency. After all table-wise checks succeed, the algorithm performs a final concrete validation (Line 16) to

Table 1. Verification results for representative zkVM components. The *Type* column uses ∘ for gate constraints, ⊙ for gate constraints with lookups, and • for designs that additionally involve permutation relations.

Component	Type	Time (s)	Result	Solver
Add	⊙	11.320	✓	Z3
Add4	⊙	5.958	✓	Z3
And	⊙	0.035	✓	Z3
IsEqualWordOperation	∘	0.299	✓	cvc5
IsZeroOperation	∘	0.011	✗	cvc5
IsZeroWordOperation	∘	0.070	✓	cvc5
MapRead	•	0.012	✓	Z3

ensure that the concrete assignment induced by the symbolic generator satisfies the fully expanded constraint semantics. This last step, implemented by ValidateConcreteAssignment, is polynomial in the size of the logical tables, since all computation variables are symbolically instantiated and no assumptions relate the public inputs.

Theorem 1 (Soundness and Completeness). *For any zkVM program P, Algorithm 1 returns ✓ if and only if P is consistent in the sense of Definition 2. Moreover, if P is inconsistent, the algorithm returns ✗ together with a concrete counterexample model witnessing the violation.*

A full proof is provided in Appendix C.

5 Evaluation

Experimental Environment. All experiments were conducted on a workstation running Ubuntu 22.04.5 LTS (kernel 6.6.87.2, WSL2). The machine is equipped with an AMD Ryzen 9 9950X3D processor (16 cores, 32 threads) and 30 GiB of RAM. Our implementation is written in Rust 1.93.0 (nightly), and relies on the cvc5 1.3.2 solver [3] (commit 84c7e48) and the Z3 4.15.3 solver [7] (64-bit).

Benchmark. We evaluate our approach on 7 representative components from the SP1 zkVM, covering all major forms of abstract semantics. These components, originally implemented in Rust, were manually translated into our DSL to isolate the core computational behavior and avoid incidental Rust-specific semantics that are irrelevant to constraint consistency.

Analysis. Our experimental results, summarized in Table 1, show that the proposed inductive verification framework effectively mitigates the state-space explosion typically encountered in symbolic reasoning. By analyzing each row independently, the algorithm avoids the exponential blow-up that would arise from symbolically unrolling entire tables, and scales cleanly across a range of SP1 components.

During evaluation, we identified an inconsistency in the IsZeroOperation component. The issue arises in the inverse operation a^{-1}: when $a = 0$, the computation sets $a^{-1} = 0$, but the corresponding constraint logic does not explicitly enforce this behavior. This discrepancy is benign and intentionally tolerated by the circuit designers, as it does not affect the semantic correctness of zkVM execution. Beyond this case, our analysis demonstrates that the abstract semantics faithfully captures both lookup and permutation constraints, allowing the algorithm to reason about these relations without expanding them into large first-order encodings.

We also compare solver behavior across two theories: QF-NIA in Z3 and the FF theory [18] in cvc5. For certain components, QF-NIA performs better, as finite-field comparisons such as $x > a$ require auxiliary variables and byte-decomposition encodings under the FF theory, leading cvc5 to struggle with complex range constraints. Furthermore, some SP1 components use native types such as `u32`, which do not map cleanly to the finite field $p \approx 2^{31}$ used by SP1, complicating their direct translation into FF encodings. Conversely, QF-NIA performs poorly on formulas involving both nonlinearity and modular arithmetic, motivating our mixed use of the two theories to overcome SMT-level bottlenecks.

We do not compare against existing automated tools such as Picus, since such systems focus exclusively on Circom and do not directly extend to the lookup and permutation constraints that are fundamental to zkVMs. A discussion of validity considerations and limitations is provided in Appendix D.

6 Related Work

Verification for Circom-Based Systems. Prior work [29] on verifying circuits generated by Circom has focused on detecting *underconstrained* behavior in R1CS encodings. Tools such as CIVER [12] and Picus [20] develop automated techniques for identifying underconstrained variables by combining structural circuit analysis with SMT-style reasoning. These approaches, however, inherit the scalability limitations of R1CS, whose sparse and largely unstructured constraint representation provides little support for inductive reasoning. In contrast, the circuit forms used in zkVMs differ fundamentally from those in Circom: their tabular constraint systems possess row-wise structural regularity that naturally supports inductive verification. By exploiting this structure, our model achieves scalability for real-world zkVM subsystems. Moreover, the properties used to detect underconstrained variables in Circom can be directly expressed within our framework: by propagating well-constrained variables row by row, our model offers a more uniform and scalable foundation for automated reasoning.

Recent literature [24,14,26] on Circom additionally introduces a *consistency* property, which aims to verify that the computation encoded by a circuit matches the behavior prescribed by its constraint system. This line of work captures the correspondence between program-level computation and the algebraic structure of R1CS, leveraging its fine-grained product relationships between intermediate variables and constraint expressions. However, these approaches operate within

a model that does not encompass lookup operations or mapping-style data structures, both of which are pervasive in zkVM designs and central to their constraint semantics. In contrast, zkVMs based on tabular constraint systems do not provide such fine-grained correspondences; achieving consistency instead requires additional dataflow analysis to reconcile execution semantics with tabular constraints. Our semantic model addresses these structural differences by aligning row-wise execution semantics with gate constraints and lookup relations in a unified manner, enabling modular verification suited to the architectural scale of modern zkVM designs.

Verification and Testing for zkVM Systems. To the best of our knowledge, there is currently no automated verification framework that handles zkVMs in a comprehensive, end-to-end manner. Our goal is not to solve this challenge directly. Instead, we present a uniform formal model for zkVM infrastructure components that is intended to serve as a foundational layer for future compositional verification, where the behavior of individual components can be linked and analyzed within a single coherent framework.

Existing efforts focus primarily on fuzzing-based testing and VM-specific theorem proving [2,15]. Arguzz [11] performs fuzz testing of zkVM implementations by mutating inputs to uncover potential security vulnerabilities, and has reported a zero-day in RISC Zero. More recently, a theorem-proving development for Jolt [15] formalizes lasso-style lookup-table semantics for a zkVM whose design is centered around lookup arguments. Because its logical model differs from the tabular constraint systems studied in this paper, that line of work is complementary to ours. In addition, Halo2Analyzer [23] studies circuits written in Halo2, the proving backend also used by zkWASM. Its method expands lookup constraints into explicit set-membership enumerations, which leads to poor scalability when tables grow large. Our work mitigates this limitation by leveraging the inductive row-wise structure of tabular constraint systems and by providing a higher-level abstraction for lookup semantics.

7 Conclusion

This work introduced a unified abstract semantics for zkVM tabular constraint systems and an inductive procedure for validating the consistency between the computational semantics implemented by the witness generator and the algebraic semantics encoded by the tabular constraints. By interpreting gate, lookup, and permutation relations through their operational meaning and embedding them within a symbolic framework, the approach enables modular, row-oriented reasoning without expanding full tables or relying on large quantified encodings. Our implementation, ZIVER, demonstrates that this methodology accurately captures the structure of real zkVM designs and can automatically verify the consistency of diverse SP1 components, providing a practical foundation for scalable correctness analysis of trace-based zero-knowledge virtual machines.

References

1. Arun, A., Setty, S., Thaler, J.: Jolt: Snarks for virtual machines via lookups. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2024)
2. Avigad, J., Goldberg, L., Levit, D., Seginer, Y., Titelman, A.: A verified algebraic representation of cairo program execution. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 153–165 (2022)
3. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
4. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica **12**(2), 225–244 (1994)
5. Bruestle, J., Gafni, P., et al.: Risc zero zkvm: scalable, transparent arguments of risc-v integrity. Draft. July **29** (2023)
6. Burleson, J., Korver, M., Boneh, D.: Privacy-protecting regulatory solutions using zero-knowledge proofs (2022)
7. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
8. Dwivedi, A.D., Singh, R., Ghosh, U., Mukkamala, R.R., Tolba, A., Said, O.: Privacy preserving authentication system based on non-interactive zero knowledge proof suitable for internet of things. Journal of Ambient Intelligence and Humanized Computing **13**(10), 4639–4649 (2022)
9. Gao, S., Li, G., Fu, H.: Zkwasm: A zksnark wasm emulator. IEEE Transactions on Services Computing **17**(6), 4508–4521 (2024)
10. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC). pp. 291–304 (1985)
11. Hochrainer, C., Wüstholz, V., Christakis, M.: Arguzz: Testing zkvms for soundness and completeness bugs. arXiv preprint arXiv:2509.10819 (2025)
12. Isabel, M., Rodriguez-Nunez, C., Rubio, A.: Scalable verification of zero-knowledge protocols. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 1794–1812. IEEE (2024)
13. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: International conference on the theory and application of cryptology and information security. pp. 177–194. Springer (2010)
14. Kolozyan, A., Vandenbogaerde, B., Swalens, J., Hoste, L., Chaliasos, S., De Roover, C.: Language-agnostic detection of computation-constraint inconsistencies in zkp programs via value inference. Cryptology ePrint Archive (2025)
15. Kwan, C., Dao, Q., Thaler, J.: Verifying jolt zkvm lookup semantics. Cryptology ePrint Archive (2024)
16. Labs, S., Auditors, E.: Audit reports: Sp1 zero-knowledge virtual machine. `https://github.com/succinctlabs/sp1/security` (2025), accessed: 2025-11-23
17. OpenVM Contributors: Openvm whitepaper. `https://openvm.dev/whitepaper.pdf` (2025), accessed: 2025-11-26
18. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields. In: International Conference on Computer Aided Verification. pp. 163–186. Springer (2023)

19. Ozdemir, A., Pailoor, S., Bassa, A., Ferles, K., Barrett, C., Dillig, I.: Split gröbner bases for satisfiability modulo finite fields. In: International Conference on Computer Aided Verification. pp. 3–25. Springer (2024)
20. Pailoor, S., Chen, Y., Wang, F., Rodríguez, C., Van Geffen, J., Morton, J., Chu, M., Gu, B., Feng, Y., Dillig, I.: Automated detection of under-constrained circuits in zero-knowledge proofs. Proceedings of the ACM on Programming Languages **7**(PLDI), 1510–1532 (2023)
21. Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: Theory of Cryptography Conference. pp. 222–242. Springer (2013)
22. Shalannanda, W.: Using zero-knowledge proof in privacy-preserving networks. In: 2023 17th International Conference on Telecommunication Systems, Services, and Applications (TSSA). pp. 1–6. IEEE (2023)
23. Soureshjani, F.H., Hall-Andersen, M., Jahanara, M., Kam, J., Gorzny, J., Ahmadvand, M.: Automated analysis of halo2 circuits. Cryptology ePrint Archive (2023)
24. Stephens, J., Pailoor, S., Dillig, I.: Automated verification of consistency in zero-knowledge proof circuits. In: International Conference on Computer Aided Verification. pp. 315–338. Springer (2025)
25. Succinct Labs: Sp1 zkvm. `https://docs.succinct.xyz/docs/sp1/introduction` (2024), accessed: 2025-11-26
26. Takahashi, H., Kim, J., Jana, S., Yang, J.: zkfuzz: Foundation and framework for effective fuzzing of zero-knowledge circuits. arXiv preprint arXiv:2504.11961 (2025)
27. Team, O., Auditors, E.: Audit report: Openvm zero-knowledge virtual machine. `https://github.com/openvm-org/openvm/security/advisories` (2025), accessed: 2025-11-23
28. Team, R.Z.: Security audit reports: Risc zero zero-knowledge virtual machine. `https://github.com/risc0/rz-security/tree/main/audits` (2025), accessed: 2025-11-23
29. Wen, H., Stephens, J., Chen, Y., Ferles, K., Pailoor, S., Charbonnet, K., Dillig, I., Feng, Y.: Practical security analysis of Zero-Knowledge proof circuits. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 1471–1487 (2024)

A Background on Rank-1 Constraint Systems (R1CS)

For completeness, we provide here a brief summary of the standard Rank-1 Constraint System (R1CS) formulation used in many zkSNARK frameworks. The main paper does not rely on R1CS semantics directly, but the definition is included for readers who may wish to compare our tabular-constraint view with the classical circuit-based representation.

Rank-1 Constraint Systems. A Rank-1 Constraint System is defined over a variable set $\mathcal{V} = \mathcal{I} \cup \mathcal{W} = \{v_1, \ldots, v_n\}$, where $\mathcal{I}$ and $\mathcal{W}$ denote public and witness variables respectively. An R1CS instance consists of matrices $A, B, C \in \mathbb{F}^{m \times n}$. Given an assignment $\sigma : \mathcal{V} \to \mathbb{F}$, its vector representation is $z = (\sigma(v_1), \ldots, \sigma(v_n))^\top$. The constraints require

$$(Az) \circ (Bz) = Cz,$$

where $\circ$ denotes component-wise multiplication in $\mathbb{F}^m$. Each row of the system expands to a polynomial equation

$$\big(\sum_i A_{j,i} v_i\big)\big(\sum_i B_{j,i} v_i\big) - \big(\sum_i C_{j,i} v_i\big) = 0.$$

An assignment satisfies the R1CS exactly when all such equations hold. The R1CS therefore provides a standard way to encode the polynomial constraints induced by an arithmetic circuit without changing its computational semantics.

B Big-Step Rules for Statements

For completeness, we collect here the basic big-step rules for statements that are used throughout the main development. These rules cover assignments to non-mapping variables, sequential composition, and primitive equality constraints. They are routine components of the operational semantics and are therefore omitted from the main text to keep the presentation focused on the table-oriented constructs.

$$\frac{\Sigma \vdash E \Downarrow e}{\Sigma \vdash V \leftarrow E \Rightarrow \{\, \Sigma[V \mapsto e] \,\}} \quad (\textsc{Assign})$$

$$\frac{\Sigma \vdash S_1 \Rightarrow \mathcal{S}_1 \qquad \forall \Sigma' \in \mathcal{S}_1.\ \Sigma' \vdash S_2 \Rightarrow \mathcal{S}_2(\Sigma')}{\Sigma \vdash S_1; S_2 \Rightarrow \bigcup_{\Sigma' \in \mathcal{S}_1} \mathcal{S}_2(\Sigma')} \quad (\textsc{Seq})$$

$$\frac{\Sigma \vdash E_1 \Downarrow e_1 \qquad \Sigma \vdash E_2 \Downarrow e_2}{\Sigma \vdash (E_1 == E_2) \Rightarrow \{\, (\sigma,\ \pi \cup \{e_1 = e_2\},\ \mu) \,\}} \quad (\textsc{Eq-Constr})$$

Fig. 6. Basic big-step rules (deferred).

C Proof of Algorithmic Correctness

This appendix establishes the correctness of Algorithm 1. We first relate the concrete execution semantics of a zkVM program to the symbolic semantics introduced in Section 4. We then show that the constraint-side symbolic semantics is an over-approximation of the concrete constraint semantics. Finally, we combine these results with a first-order encoding of lookup and permutation constraints to prove that Algorithm 1 is sound and complete with respect to the consistency notion of Definition 2.

C.1 Concrete and Symbolic Semantics

Recall that a zkVM program is a tuple $P = (\mathcal{T}, \mathcal{X}, \mathcal{M}, \mathsf{Gen}, \mathsf{C}, \Pi)$ (Definition 1). For a logical table $T \in \mathcal{T}$ with row pattern and generator/constraint fragments $\mathsf{Gen}_T, \mathsf{C}_T$, we write $[\![\mathsf{Gen}_T]\!]$ and $[\![\mathsf{C}_T, \Pi]\!]$ for their concrete big-step semantics on table rows and memories, as described in Figure 5.

On the symbolic side, a symbolic state is a triple $\Sigma = (\sigma, \pi, \mu)$ (Definition 3). A symbolic environment is a finite set $S \subseteq \mathit{State}$ of symbolic states, and the big-step judgment $\Sigma \vdash S \Rightarrow S'$ (Figure 6 and 5) defines the symbolic transition relation for a single row of table T. We write $\mathsf{Exec}(S, S_T)$ for the environment obtained by repeatedly applying these rules over all rows of T, where S_T denotes either Gen_T^S or $\mathsf{C}_T^S \cup \Pi$.

To relate symbolic and concrete semantics, we interpret symbolic expressions under a valuation of the initial symbols.

Definition 4 (Interpretation and Concretization). *Let $\mathcal{V}$ be the set of symbolic variables introduced by initialization (one symbol $\mathsf{sym}(x)$ for each public variable $x \in \mathsf{Pub}(\mathcal{X})$ and for each fresh unknown). An* interpretation *is a map $\eta : \mathcal{V} \to \mathbb{F}$ into the base field.*

Given an interpretation η and a symbolic expression $e \in E$, we write $[\![e]\!]_\eta \in \mathbb{F}$ for the usual evaluation of e obtained by replacing each symbolic variable v with $\eta(v)$ and evaluating arithmetic operators in $\mathbb{F}$.

The concretization *of a symbolic state $\Sigma = (\sigma, \pi, \mu)$ under η, written $\gamma_\eta(\Sigma)$, is defined as follows:*

- *The concrete store $\rho : \mathcal{X} \to \mathbb{F}$ is given by $\rho(x) = [\![\sigma(x)]\!]_\eta$ whenever $\sigma(x)$ is defined.*
- *The concrete path condition holds iff every $b \in \pi$ evaluates to true, i.e., $\bigwedge_{b \in \pi} [\![b]\!]_\eta = \mathsf{true}$.*
- *The concrete memory trace is the set $\mu_\eta = \{([\![\tau]\!]_\eta, [\![k]\!]_\eta, [\![v]\!]_\eta) \mid (\tau, k, v) \in \mu\}$.*

We write $\gamma_\eta(\Sigma) \Downarrow (s, m)$ to mean that the concretization of Σ under η yields a concrete store s and concrete memory trace m, and that the path condition holds under η.

Definition 5 (Abstraction Relation). *We say that a symbolic state Σ* abstracts *a concrete configuration (s, m), written $\Sigma \sqsupseteq (s, m)$, if there exists an*

interpretation η such that $\gamma_\eta(\Sigma) \Downarrow (s, m)$. This extends point-wise to environments: a symbolic environment S abstracts a set of concrete configurations C, written $S \sqsupseteq C$, if for every $(s, m) \in C$ there exists $\Sigma \in S$ with $\Sigma \sqsupseteq (s, m)$.

C.2 Exactness of Generator Semantics

The generator fragment Gen contains only computation statements: assignments to scalar and trace variables, control flow, and map reads/writes. By construction, the symbolic expression language and the symbolic memory table μ faithfully represent these operations.

Lemma 1 (Generator Exactness). *Let $T \in \mathcal{T}$ be a logical table, and let (s_0, m_0) be an initial concrete configuration for the rows of T. Let $S_0 = \{\Sigma_0\}$ be the initial symbolic environment used by Algorithm 1. Then:*

1. *For every concrete execution of $[\![\mathsf{Gen}_T]\!]$ from (s_0, m_0) to (s_1, m_1), there exists $\Sigma_1 \in \mathsf{Exec}(S_0, \mathsf{Gen}_T^S)$ and an interpretation η such that $\Sigma_1 \sqsupseteq (s_1, m_1)$.*
2. *Conversely, for every $\Sigma_1 \in \mathsf{Exec}(S_0, \mathsf{Gen}_T^S)$ and interpretation η with $\gamma_\eta(\Sigma_1) \Downarrow (s_1, m_1)$, there exists a concrete execution of $[\![\mathsf{Gen}_T]\!]$ from (s_0, m_0) to (s_1, m_1).*

Proof. By induction on the big-step derivation of $\Sigma \vdash S \Rightarrow S'$ in Figure 5, restricted to generator statements. The base cases (assignments, trace reads, arithmetic expressions) follow directly from the definition of $[\![e]\!]_\eta$ and the fact that the symbolic store σ records exactly the expressions computed in the concrete semantics. The inductive cases for sequencing and conditionals follow by standard compositionality arguments: in the If rule, the symbolic semantics splits the path condition into e and $\neg e$, and each branch coincides with the corresponding branch in the concrete semantics under the same interpretation. Since map reads and writes in Gen are modeled by inserting and retrieving events from μ according to the same key and timestamp tuples as in the concrete memory model, the evolution of m is also matched exactly.

Lemma 1 implies that, for every public input, the set of witness assignments produced by the concrete generator coincides with the set of witnesses represented by symbolic generator executions.

Definition 6 (Generator Witness Sets). *For a fixed public assignment σ_{pub} and table T, let $\mathcal{W}_T^{\mathsf{gen}}(\sigma_{\mathsf{pub}})$ be the set of witness projections on $\mathsf{Wit}(X_T)$ arising from concrete executions of $[\![\mathsf{Gen}_T]\!]$ under σ_{pub}. Let $\mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}})$ be the set of witness projections induced by pairs (Σ, η) such that $\Sigma \in \mathsf{Exec}(\{\Sigma_0\}, \mathsf{Gen}_T^S)$, the public slice of σ equals σ_{pub}, and $\gamma_\eta(\Sigma)$ is defined.*

Corollary 1. *For every table T and public assignment σ_{pub}, $\mathcal{W}_T^{\mathsf{gen}}(\sigma_{\mathsf{pub}}) = \mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}})$.*

C.3 Over-Approximation of Constraint Semantics

On the constraint side, the symbolic semantics deliberately abstracts away some structural information. Gate constraints are modeled exactly, but lookup and permutation constraints are treated using summary predicates and lookup-connected components, which can admit spurious behaviors.

Let $[\![\mathsf{C}_T, \Pi]\!]$ denote the concrete satisfaction relation between a table assignment and the constraint system. We write $\mathcal{W}_T^{\mathsf{con}}(\sigma_{\mathsf{pub}})$ for the set of witness assignments to $\mathsf{Wit}(X_T)$ that can be extended (together with the public slice) to a full table assignment satisfying all concrete constraints (C, Π).

On the symbolic side, given an environment $S_0 = \{\Sigma_0\}$, we obtain $S_2 = \mathsf{Exec}(S_0, \mathsf{C}_T^S \cup \Pi)$. Each symbolic state $\Sigma_2 = (\sigma_2, \pi_2, \mu_2)$ induces a formula $\varphi_{\mathsf{con}}(\Sigma_2)$ over initial symbols and auxiliary placeholders that encodes the conjunction of all accumulated path conditions and row-wise constraints; Algorithm 1 extracts exactly this information into φ_{path} and φ_μ.

Lemma 2 (Constraint Over-Approximation). *Fix a table T and public assignment σ_{pub}. Then*

$$\mathcal{W}_T^{\mathsf{con}}(\sigma_{\mathsf{pub}}) \subseteq \mathcal{W}_T^{\mathsf{abs}}(\sigma_{\mathsf{pub}}),$$

where $\mathcal{W}_T^{\mathsf{abs}}(\sigma_{\mathsf{pub}})$ is the set of witness projections induced by all pairs (Σ_2, η) such that $\Sigma_2 \in \mathsf{Exec}(S_0, \mathsf{C}_T^S \cup \Pi)$, σ_2 agrees with σ_{pub} on public variables, and $\gamma_\eta(\Sigma_2)$ is defined.

Proof (Proof sketch). Gate constraints $E == E'$ are symbolically represented as equalities between symbolic expressions, and the evaluation rules for expressions (Figure 4) are point-wise sound with respect to $[\![\cdot]\!]_\eta$. Thus any concrete satisfaction of a gate constraint can be represented by some symbolic execution and interpretation.

Lookup constraints $(E_1, \ldots, E_k) \in T'$ are symbolically summarized by instantiating the row-pattern formula $\Phi_{T'}(z_1, \ldots, z_k)$ at the tuple $z_j := e_j$ obtained from symbolic evaluation (rule Lookup in Figure 5). The use of lookup-connected components avoids infinite unfolding by reusing the union of all $\Phi_{T'}$ formulas in the same component. By construction, every concrete lookup event is captured by some instance of $\Phi_{T'}$, but the converse need not hold, so the set of symbolic models is a superset of the concrete ones.

Permutation constraints $(T_i[C_i] = T_j[C_j])$ are treated as abstract equalities between multisets of column tuples; the symbolic semantics records only that the corresponding tuples are drawn from the same underlying evaluation domain and share the same field values. Again, any concrete satisfaction of the permutation relation can be embedded into a symbolic state with a suitable interpretation, but the symbolic semantics may admit additional spurious permutations that do not correspond to actual rowwise bijections.

Finally, the offline memory-checking constraints used to connect memory read/write events across tables are modeled by the global memory table μ and the Read-Write rule in Figure 5. The soundness of this abstraction with respect to the standard sequential memory semantics is established by the underlying

memory-checking scheme (e.g., [4]); we rely on these results without reproving them here. Combining these observations yields the inclusion.

C.4 First-Order Encoding of Lookup and Permutation

To eliminate spurious behaviors introduced by the abstraction in Lemma 2, Algorithm 1 refines potential counterexamples by expanding lookup and permutation constraints into first-order formulas over fully materialized tables.

Definition 7 (Lookup Encoding). *Let T and T' be tables, and let $(E_1, \dots, E_k) \in T'$ be a lookup constraint, where the first i components encode a timestamp and the remaining components form a key. For a concrete table assignment with row index sets $I_T, I_{T'}$, the lookup constraint can be expressed as the first-order formula*

$$\forall \ell \in I_T. \bigvee_{\ell' \in I_{T'}} \bigwedge_{1 \le j \le k} E_j(\ell) = r^{(0)}_{T'}[c_j](\ell'),$$

where each $E_j(\ell)$ denotes the interpretation of E_j at row ℓ of T, and $r^{(0)}_{T'}[c_j](\ell')$ denotes the value of the j-th selected column at row ℓ' of T'.

Definition 8 (Permutation Encoding). *Let $(T_i[C_i] = T_j[C_j])$ be a permutation constraint, with column lists $C_i = (c_1, \dots, c_m)$ and $C_j = (d_1, \dots, d_m)$, and index sets I_i, I_j such that $|I_i| = |I_j|$. We encode the permutation relation by introducing a bijection $f : I_i \to I_j$ and asserting*

$$\exists f. \left(\forall \ell \in I_i. \bigwedge_{1 \le q \le m} T_i[\ell, c_q] = T_j[f(\ell), d_q]\right) \wedge \mathsf{Bijective}(f),$$

where $\mathsf{Bijective}(f)$ is the usual first-order encoding of injectivity and surjectivity of f over I_i and I_j.

By instantiating these formulas with concrete index sets and table cells obtained from the DSL program, the procedure $\mathsf{ReconstructFullConstraints}(P)$ builds a first-order formula $\varphi_{\mathsf{full}}(P)$ whose models are in one-to-one correspondence with concrete table assignments that satisfy all gate, lookup, permutation, and memory-checking constraints.

Lemma 3 (Full Encoding Equivalence). *For any zkVM program P and any public assignment σ_{pub}, an assignment to the witness and table variables satisfies the concrete constraint system (C, Π) under σ_{pub} if and only if it can be extended to a model of $\varphi_{\mathsf{full}}(P)$.*

Proof. Immediate from the definitional nature of the encodings above and the soundness of the offline memory-checking scheme.

C.5 Soundness of Rejecting Counterexamples

Algorithm 1 can return 7 in two ways: either directly at line 12 after validating a model of $\varphi_{\mathsf{path}} \wedge \varphi_{\mathsf{io}} \wedge \varphi_{\mu}$, or after refining to $\varphi_{\mathsf{full}}(P)$ at lines 13–15.

Lemma 4 (Soundness of Direct Validation). *Suppose Algorithm 1 returns* 7 *at line 12 with model m. Then P is inconsistent in the sense of Definition 2.*

Proof. The model m satisfies $\varphi_{\mathsf{path}} \wedge \varphi_{\mathsf{io}} \wedge \varphi_{\mu}$ for some pair of symbolic states Σ_1, Σ_2 produced by executing Gen_T^S and $\mathsf{C}_T^S \cup \Pi$ on the same table T. Thus m induces an interpretation η and concrete configurations (s_1, m_1), (s_2, m_2) such that:

- The public slices of s_1 and s_2 coincide (by φ_{io}).
- The witness slices of s_1 and s_2 disagree on at least one witness variable in $\mathsf{Wit}(X_T)$ (by the disequality in φ_{io}).
- The memory traces m_1 and m_2 are identical (by φ_{μ}).

By Lemma 1 and Lemma 2, there exists a concrete execution of Gen_T leading to (s_1, m_1) and a concrete satisfying assignment of the constraint system leading to (s_2, m_2) under the same public inputs. The procedure $\mathsf{ValidateConcreteAssignment}(m, P)$ replays this assignment in the concrete semantics of P and returns true exactly when the concrete generator and constraints disagree on the resulting witness. Hence, if line 12 returns 7, there exists a public input and a witness assignment that is accepted by the constraints but not produced by the generator, contradicting consistency.

Lemma 5 (Soundness of Refined Counterexamples). *Suppose Algorithm 1 returns* 7 *at line 15 after constructing $\varphi_{\mathsf{full}}(P)$. Then P is inconsistent.*

Proof. If line 10 yields sat but $\mathsf{ValidateConcreteAssignment}(m, P)$ returns false, the model m witnesses a spurious behavior admitted by the abstract semantics of lookup and permutation constraints. The refinement step reconstructs $\varphi_{\mathsf{full}}(P)$, whose models exactly correspond to concrete satisfying assignments of (C, Π) by Lemma 3. If the subsequent call to $\mathsf{Query}(\varphi_{\mathsf{full}}(P))$ returns sat, the resulting model is a genuine counterexample in which the constraints accept a witness that is not produced by the generator. Hence P is inconsistent.

C.6 Soundness of Acceptance

We now argue that, if Algorithm 1 returns ✓, then P is consistent. Intuitively, unsatisfiability of $\varphi_{\mathsf{path}} \wedge \varphi_{\mathsf{io}} \wedge \varphi_{\mu}$ for all tables T and all pairs of generator/constraint states implies that, for every public assignment, no witness admitted by the abstract constraint semantics lies outside the generator image. Combined with the abstraction sandwich

$$\mathcal{W}_T^{\mathsf{gen}} = \mathcal{W}_T^{\mathsf{sym}} \supseteq \mathcal{W}_T^{\mathsf{abs}} \supseteq \mathcal{W}_T^{\mathsf{con}},$$

this yields the desired property for the concrete semantics.

Lemma 6 (No Abstract Counterexamples). *Assume that, for every table T and every pair $(\Sigma_1, \Sigma_2) \in \mathsf{Zip}(\Gamma_1, \Gamma_2)$ produced at lines 3–4, the formula $\varphi_{\mathsf{path}} \wedge \varphi_{\mathsf{io}} \wedge \varphi_{\mu}$ is unsatisfiable. Then, for every public assignment σ_{pub} and table T,*

$$\mathcal{W}_T^{\mathsf{abs}}(\sigma_{\mathsf{pub}}) \subseteq \mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}}).$$

Proof (Proof sketch). Suppose, towards a contradiction, that there exist a public assignment σ_{pub} and a witness valuation w belonging to $\mathcal{W}_T^{\mathsf{abs}}(\sigma_{\mathsf{pub}})$ but not to $\mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}})$. By definition of $\mathcal{W}_T^{\mathsf{abs}}$, there exists some $\Sigma_2 \in \mathsf{Exec}(\{\Sigma_0\}, \mathsf{C}_T^S \cup \Pi)$ and interpretation η such that $\gamma_\eta(\Sigma_2)$ realizes σ_{pub} and witness w. Since w is not in $\mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}})$, no generator state Σ_1 with the same public slice and witness w is reachable under Gen_T^S.

Consider the pair (Σ_1, Σ_2) obtained by aligning Σ_2 with any generator state Σ_1 whose public slice matches σ_{pub} (Lemma 1 ensures at least one such state exists). Under the interpretation η, the path conditions π_1, π_2 and the memory traces μ_1, μ_2 can be made to agree, while the witness slices necessarily differ. This yields a model of $\varphi_{\mathsf{path}} \wedge \varphi_{\mathsf{io}} \wedge \varphi_{\mu}$, contradicting the assumed unsatisfiability.

Combining Lemmas 2, 1, and 6, we obtain:

Lemma 7 (Abstraction Sandwich). *If Algorithm 1 does not reject at lines 12 or 15, then, for every table T and public assignment σ_{pub},*

$$\mathcal{W}_T^{\mathsf{con}}(\sigma_{\mathsf{pub}}) \subseteq \mathcal{W}_T^{\mathsf{abs}}(\sigma_{\mathsf{pub}}) \subseteq \mathcal{W}_T^{\mathsf{sym}}(\sigma_{\mathsf{pub}}) = \mathcal{W}_T^{\mathsf{gen}}(\sigma_{\mathsf{pub}}).$$

In particular, every witness admitted by the concrete constraint system arises from some execution of the generator.

The final call $\mathsf{ValidateConcreteAssignment}(\Sigma_0, P)$ at line 16 checks that the concrete generator and constraint system agree on the witness produced by the generator for the initial public assignment. Together with Lemma 7, this yields the main correctness theorem.

Theorem 2 (Correctness of Algorithm 1). *Algorithm 1 satisfies:*

1. *If it returns 7, then P is inconsistent.*
2. *If it returns ✓, then P is consistent in the sense of Definition 2.*

Proof. Item 1 follows from Lemmas 4 and 5. For Item 2, assume the algorithm returns ✓. Then all candidate abstract counterexamples are ruled out (Lemma 6), and thus $\mathcal{W}_T^{\mathsf{con}} \subseteq \mathcal{W}_T^{\mathsf{gen}}$ for every table T by Lemma 7. The final validation at line 16 ensures that, for the concrete execution starting from Σ_0, the generator and constraints agree on the resulting witness. Unfolding Definition 2 gives the desired property that, for every public assignment, any witness accepted by the constraint system is also produced by the generator.

D Discussion

Compositional Properties Verification. Our formalization of tabular constraint systems relies on several structural assumptions regarding the behavior of mapping variables. These assumptions are necessary for reasoning about a single logical table in isolation: for example, we assume that the initialization of mapping entries, the public exposure of their initial values, and the constraints governing their final states are all well-formed. Under these assumptions, the inductive correspondence between the witness generator and the row-wise constraint semantics holds for any individual table.

In practical zkVM architectures, these assumptions are not arbitrary. They are typically enforced by other dedicated logical tables—for instance, tables that govern memory initialization and finalization. Such tables ensure that mapping variables satisfy the required global invariants, thereby legitimizing the table-local assumptions used in our model. Our formalization focuses on these local properties and does not attempt to establish the full set of cross-table invariants; verifying such system-level interactions remains important future work.

Validity of Experimental Benchmarks and Verification Challenges. The benchmarks used in our prototype evaluation are intentionally simplified. They cover a representative subset of SP1 components but exclude certain cross-chip lookup relations that primarily function as propagation mechanisms in a full zkVM execution environment. This design choice reflects our focus on validating the core abstract semantics and inductive reasoning framework rather than performing an end-to-end system verification of SP1. A complete compositional verification would require incorporating all inter-chip interactions under a finalized, stable zkVM specification, which constitutes substantial engineering work and is beyond the intended scope of this prototype.

Even within this restricted setting, our evaluation highlights several challenges that may affect future scalability. One difficulty arises from computational patterns that do not map cleanly to the finite-field reasoning supported by cvc5-ff. For instance, the BabybearRange component performs bitwise decomposition on 32-bit machine integers, whereas the underlying SP1 field size $p \approx 2^{31}$ cannot directly represent all u32 values. A faithful encoding of such behavior would require either more refined semantic models or auxiliary encodings, increasing the complexity of verification.

A second challenge concerns SMT support for modular arithmetic and nonlinear operations. Formulas combining modular constraints with non-additive or non-multiplicative operators remain difficult for current solvers. Although recent finite-field solving techniques in cvc5 leverage specialized Gröbner-basis methods [19] for zero-knowledge applications, they still do not fully capture the richer operational patterns emerging in modern zkVM architectures. Further advances in SMT solving—particularly in mixed nonlinear–modular theories and finite-field reasoning—will be critical to scaling automated zkVM verification to system-level deployments.