Automated Soundness and Completeness Vetting of Polygon zkEVM

Xinghao Peng[†,§], Zhiyuan Sun[†,‡,§], Kunsong Zhao[†], Zuchao Ma[†],
Zihao Li[†], Jinan Jiang[†], Xiapu Luo[†,∗], Yinqian Zhang[‡]

[†]*The Hong Kong Polytechnic University* [‡]*Southern University of Science and Technology*

Abstract

Zero-knowledge rollups have emerged as popular layer 2 scaling solutions for blockchains. Polygon zkEVM, a leading deployment of zk rollups, leverages non-deterministic execution to derive *free inputs* from an unconstrained command evaluator when implementing the zkEVM. This mechanism significantly simplifies the design of zkEVM and enhances the performance of proof generation. However, it introduces the challenge of requiring developers to define constraints for free inputs, a task that demands strong mathematical expertise and is prone to errors. As a component of the layer-2 infrastructure, the zkEVM's vulnerabilities could lead to powerful attacks. However, despite their importance, the security of free inputs in zkEVM remains unexplored.

In this paper, we present the first systematic exploration of free inputs in Polygon zkEVM. Our study reveals critical soundness and completeness issues with them. In particular, we uncover a new attack surface, termed the dual execution path attack, which targets unsound implementations of free inputs and can lead to chain splits. Moreover, we design the first tool, FreeVer, which facilitates the verification of the soundness and completeness of free inputs with formal semantics. Additionally, it automatically generates formal specifications for correct constraints by constructing prover state graphs that model both the behaviors of malicious and honest provers. It then uses the states from the honest prover as specifications to assist in the verification of states from the malicious one. FreeVer also adopts optimization strategies to reduce the complexity of constraints for effective verification. Our evaluation results show that FreeVer can correctly identify all previously disclosed free input related vulnerabilities and detect 7 new vulnerabilities in Polygon zkEVM. All detected bugs are submitted through the bug bounty program and are confirmed as *high impact* vulnerabilities.

§ Equal Contribution. ∗ The corresponding author.

1 Introduction

Blockchains rely on distributed networks and intricate consensus mechanisms to ensure security and integrity [21,22,47,48]. However, these mechanisms introduce significant performance overhead, resulting in low transaction throughput and high transaction costs [2]. To address this issue, zero-knowledge rollups have emerged as layer 2 scaling solutions [11,18,35,37]. The core concept is to bundle transactions into batches, execute them using an off-chain virtual machine, and generate a validity proof to guarantee computational integrity. This validity proof is then submitted to and verified by a layer 1 on-chain verifier in a single transaction [10]. By shifting expensive on-chain computations off-chain and verifying the validity proof cheaply on-chain, zk rollups maintain the same security guarantees as layer 1 blockchains while improving throughput and reducing costs [6].

Polygon zkEVM [37] is a prominent deployment of zero-knowledge rollups [19]. Polygon zkEVM is implemented as a combination of several state machines [42], where state transitions can be proved with zkSTARK [4,34]. It also implements two domain-specific languages: the *polynomial identity language* (PIL) and the *zero-knowledge assembly* (zkASM) [36]. PIL defines polynomial identities (a.k.a. polynomial constraints) for the state transitions of the state machines [40], while zkASM is the assembly language for these state machines and is used to implement EVM instructions and transaction processing logic [41]. The validity proof is accepted by the verifier only if the execution of the EVM adheres to all the constraints defined in PIL [42].

For performance considerations, zkASM supports non-deterministic execution with *free inputs* [36]. Specifically, it provides support for computations not directly handled by the state machines through command evaluations [39]. Command evaluation is performed by an external command evaluator, and the resulting outputs are provided to the zkASM executor as *free inputs* [39]. These free inputs are subsequently verified using zkASM instructions. This mechanism reduces the complexity of state machine design, thereby enhancing

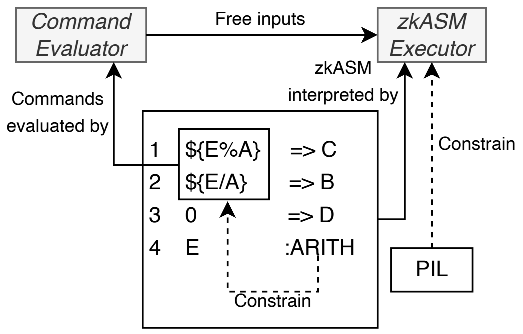

Figure 1: Free inputs

performance. Furthermore, verifying results often requires fewer computational steps than calculating them, which also contributes to performance improvement [11, 13].

Figure 1 presents an example of using free inputs. The zkASM code calculates the quotient and remainder of one number divided by another. Since division and modulus operations are not directly supported in zkASM, it relies on a command evaluator to evaluate commands and derive the results as free inputs (lines 1-2). Because PIL only constrains the zkASM code interpreted by the executor (all zkASM code except for the commands), developers must ensure that the provided free inputs are correct. This is achieved indirectly using the `ARITH` instruction in zkASM, which asserts that `A*B+C==D*2^256+E`. In other words, it constrains the free inputs such that the quotient `B` multiplied by the divisor `A` plus the modulus `C` must equal the dividend `E`.

As with the challenge of defining constraints in Circom [5, 23], checking free inputs is also prone to errors. In fact, the previous example contains an underconstrained vulnerability: the free inputs could be malformed—for instance, the provided remainder might be larger than the divisor. To properly constrain the free inputs, an additional `LT` instruction must be applied to ensure that the remainder is smaller than the divisor. Free input related vulnerabilities arise due to the mismatch between the command and the zkASM instructions used to verify them. Specifically, two types of vulnerabilities can occur:

Soundness vulnerabilities: These arise when the constraints on the free inputs are too loose. Malicious provers can exploit such underconstrained inputs to produce valid proofs with invalid data.

Completeness vulnerabilities: These occur when the constraints on the free inputs are overly strict. As a result, valid data cannot produce valid proofs.

Currently, identifying the soundness and completeness of free inputs relies on manual audits, a process that is both time-intensive and prone to oversight.

To bridge this gap, we conduct an in-depth investigation of security issues related to free inputs in Polygon zkEVM [37], one of the leading deployments of zk rollups [19]. Our study uncovers a new class of underconstrained vulnerabilities, which we term *dual execution path vulnerabilities*, and introduces a novel and powerful attack that leverages these vulnerabilities to cause chain splits. We also propose a novel formal verification framework designed to automatically verify the constraints of free inputs in Polygon zkEVM. This task is non-trivial, and the challenges are as follows:

Challenge 1: Difficulty in extracting free input constraints. The constraints of free inputs are deeply intertwined with the broader constraints of the virtual machine, resulting in a constraint system that is prohibitively large and complex for effective reasoning. Consequently, extracting the constraints of free inputs is essential for effective verification but poses a significant challenge.

Challenge 2: Unsolvable constraints for SMT solvers. Verification of free inputs relies on symbolic exploration and reasoning, both of which utilize SMT solvers. However, Polygon zkEVM frequently performs word composition and decomposition, which significantly increases the complexity of symbolic expressions. Additionally, the interpretation of free inputs often involves algorithms with loops that cannot be explored using symbolic values. These factors render most of the constraints for free inputs unsolvable, posing substantial challenges to both path exploration and verification.

Challenge 3: Difficulty in automatic generation of specifications for correct constraints. A fundamental question in verification is determining the correct constraints [5]. Tools such as CIVER [16] and CODA [20] rely on experts to manually define type refinements and pre/post conditions, respectively. However, these approaches are labor-intensive and prone to oversight.

To address these challenges, we first develop executable symbolic formal semantics for zkASM. The semantics enable the symbolic execution of zkASM code, allowing us to extract constraints for free inputs (addressing C1). Next, we optimize the symbolic execution by introducing a novel lazy evaluation technique and simplifying semantic rules for zkASM code that involves complex algorithms with loops (addressing C2). Finally, we model the behavior of honest and malicious provers using prover state graphs (PSGs). These PSGs facilitate analyzing issues in free inputs by extracting concrete specifications from the graphs (addressing C3).

We collect a dataset containing all disclosed free inputs related vulnerabilities in Polygon zkEVM and the corresponding fix from 7 public reports [14, 28–32, 43] by well-known Web3 security companies. The verification results show that our tool could correctly identify the soundness and completeness of all samples in this dataset. We also run FreeVer against the Polygon zkEVM implementation [41] and it uncovers 6 soundness issues and 1 completeness issue, which are reported to and confirmed as *high impact vulnerabilities* by the Polygon team.

We summarize our contributions as follows:

- We reveal a new class of underconstrained vulnerabilities in

Polygon zkEVM, and introduce a novel and powerful dual execution path attack against it. The attack can cause chain splits and allow attackers to profit from double spending.

- We design and implement a formal verification framework, FreeVer, to verify the constraints of free inputs in Polygon zkEVM. To the best of our knowledge, FreeVer is the first automated tool for detecting completeness and soundness vulnerabilities related to free inputs.

- Our evaluation of FreeVer reveals 7 new vulnerabilities in Polygon zkEVM, all of which are submitted to the Polygon team through bug bounty program and confirmed as high impact vulnerabilities.

2 Background and Threat Model

We provide a brief introduction to the Polygon zkEVM [37] and the K Framework [7].

2.1 Polygon zkEVM

In this study, we focus primarily on the design and implementation of the prover, named *zkProver*, of Polygon zkEVM. For a comprehensive and detailed overview of the entire architecture, we refer readers to the official documentation [36].

We first explain important terminologies:

• *State machines*: zkProver follows a modular design, consisting of one main state machine, six secondary state machines, and six auxiliary state machines. Each secondary state machine specializes in a specific type of computation (e.g., arithmetic, hashing, etc.), while the auxiliary state machines handle tasks such as padding and memory alignment. The main state machine dispatches tasks to the other state machines.

• *Polynomial identity language*: PIL is a domain specific language used by zkProver. PIL defines polynomial identities (i.e., constraints) that the computations of the state machines must satisfy.

• *Zero-knowledge assembly*: zkASM is another domain specific language designed for zkProver. It is the assembly of the state machines and is used to implement the ROM.

• *ROM*: The ROM is a program written in zkASM. It interprets all EVM opcodes and handles batch processing and transaction execution logic. Within the Polygon zkEVM, the ROM serves a role similar to that of the EVM Interpreter in Ethereum. The official implementation for ROM is in the zkevm-rom repository [41].

• *zkASM Executor*: The executor is a program that interprets the zkASM code and generate the execution trace. Its execution must satisfies the constraints defined in PIL in order to generate a validity proof. The executor is a component of the zkProver, implemented as part of the zkevm-prover repository [39].

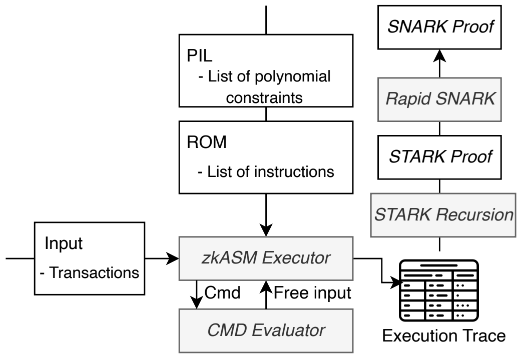

Figure 2: Overview of Polygon zkEVM proof generation

• *Execution trace*: The execution trace is a record of state transitions represented as a table with predetermined dimensions. Each row corresponds to an intermediate state, and the rows are arranged sequentially in execution order.

• *Counters*: There are six counters that track how many rows of the execution trace have been used during computation. Each state machine is limited to a maximum of 2^{23} rows. If all rows are exhausted, the prover triggers an out-of-counters (OOC) error and generates a proof indicating that no blockchain state changes occurred due to the error.

• *Free input*: A free input is derived from computations that are not handled by state machines. It is the evaluation result of a command. Since PIL only constrains state machine computations, commands are not constrained.

• *Command evaluator*: It is a component of the zkProver that evaluates free input commands, implemented in the zkevm-prover repository [39].

zkProver workflow. The workflow of zkProver is illustrated in Figure 4. The zkASM Executor interprets the ROM, and takes the transactions to be proved as input. The executor also verifies the interpretation against the constraints specified in PIL [40]. During execution, the executor can query an external command evaluator to generate free inputs. The executor records all internal states as an execution trace. This trace is then passed into the proving system, which first generates STARK proofs using STARK recursion. Subsequently, SNARK is used to produce a succinct proof that verifies the STARK proof.

2.2 K Framework

K is a matching logic [8, 25] based framework specifically designed for defining formal language semantics. It automatically generates parsers, interpreters, symbolic executors, model checkers, and deductive verifiers directly from syntax and semantics definitions [9, 17]. K has been successfully used to formalize instruction sets (e.g., x86-64 ISA [9], EVM [15]) and programming languages (e.g., C, Java [45], Solidity [17]), with these formal semantics enabling the discovery of critical real-world bugs, such as the x86-64 ISA inconsistency between specification and implementation [9].

To implement a language in K, users specify the syntax, rewrite rules, and configuration of it. The *syntax* is described using the conventional Backus-Naur form [46]. The semantics are expressed as a parametric transition system, which consists of a set of reduction rules, known as *rewrite rules*, applied over the *configuration*. The configuration organizes the program's code and state into units called cells, which are labeled and can be nested. A rewrite rule represents a single step in the transition between configurations.

Consider a simple language whose syntax, configuration, and semantic rules are defined in Listing 1. This language supports only two statements: 8-bit integer variable definition and increment. The syntax rules from lines 1 to 3 describe that the language allows variable definitions and increments, and statements are evaluated sequentially from left to right. The configuration specifies the initial program state, which contains a `k` cell holding the statements and a `vars` cell that starts as an empty map. The three rewrite rules then define the semantics for single-step execution, variable definition, and self-increment. From the semantics, we can deduce that the integers in SIMPLE are 8-bit signed integers, since all operations are modulo 256.

```
1   syntax Stmt ::= "var" Id "=" Int
2                 | "++" Id
3                 > left: Stmt Stmt
4
5   configuration <k> $PGM:Stmt </k>
6                 <vars> .Map </vars>
7
8   rule [step]:
9       <k> S1:Stmt S2:Stmt => S1 ~> S2 ... </k>
10  rule [def]:
11      <k> var X = I => .K ... </k>
12      <vars> V => V[X <- (I %Int 256)] </vars>
13  rule [inc]:
14      <k> ++ X => .K ... </k>
15      <vars> ... (X |-> V) => (X |-> ((V +Int 1)) %Int
        ↪  256 ) ... </vars>
```

Listing 1: SIMPLE semantics

Given the semantics, we can construct specifications to check certain properties. For example, to ensure that no integer overflow occurs in a SIMPLE program (lines 3-4), we could write the following claim and attempt to prove it using K Framework. This claim asserts that after evaluating the program, the final value of the variable `$x` should be larger than its initial value (i.e., no integer overflow should occur). If K fails to prove the claim, it indicates that an integer overflow exists in this code snippet.

```
1   claim <k>
2           var $x = 255
3           ++ $x
4           => .K ...
5         </k>
6         <vars> ... $x |-> V ... </vars> ensures V >Int
          ↪  255
```

2.3 Threat Model

We assume a trustless setup where an attacker has full access to the implementations of the prover and verifier at the source code level. Additionally, we assume the attacker can act as a prover, submitting proofs for verification. Furthermore, we assume the attacker can arbitrarily modify the execution trace, provided the modified trace remains valid with respect to the constraints, to exploit vulnerabilities related to free inputs. Specifically, we define two kinds of prover:

Honest Prover: An honest prover faithfully executes transactions and generates proofs. It follows the official implementation [39].

Malicious Prover: A malicious prover aims to exploit free input related vulnerabilities in the zkEVM system. To achieve this, it modifies both the command evaluator and the zkASM executor to derive malformed free inputs and execution traces. Finally, it generates a malformed proof and submits it to the layer 1 verifier contract to trigger chain splits.

3 The Dual Execution Path Attack

In this section, we uncover a new attack vector against Polygon zkEVM, termed the dual execution path attack.

Vulnerable code pattern. zkASM code that: ① contains an underconstrained free input, which can be exploited to manipulate the execution path; ② still produces a valid output even after maliciously altering the execution path; and ③ consumes different counters on the original and manipulated paths, is vulnerable to the dual execution path attack. For instance, Figure 3 illustrates a vulnerability, simplified from a real-world issue detected by FreeVer in Polygon zkEVM. The code takes two input values and performs either multiplication or addition based on whether the inputs are the same. It branches based on a free input, derived from a command that checks if the two inputs are equal. If they are the same, path 1 is taken; otherwise, path 2 is chosen. Since the jump is based on a free input, the validity of the free input must be checked on both paths. However, while path 1 includes an `ASSERT` instruction to ensure that `A == C`, path 2 lacks a similar check. This underconstrained free input on path 2 enables malicious provers to provide a non-zero free input, even when `A` and `C` are both x, thereby manipulating the execution to follow path 2. Since both paths yield the same output $2x$ when the inputs are equal, altering the path does not affect the validity of the subsequent computations. However, the critical issue is that the computations on the two paths consume different numbers of counters, which can be exploited for powerful attacks.

The attack. The dual execution path attack exploits the differing consumption of counters across paths to generate two

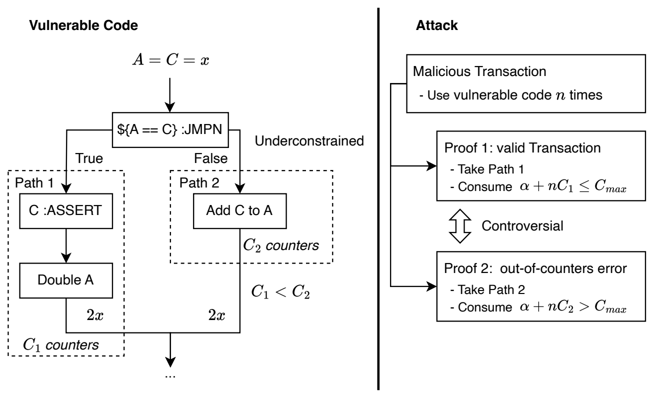

Figure 3: Dual execution path attack

conflicting but valid proofs, which can be utilized to cause a chain-split. The attack proceeds as follows: First, the attacker constructs a transaction that repetitively uses the vulnerable code n times. For proof 1, the attacker does not manipulate the execution path, so the proof consumes a total of $\alpha + nC_1$ counters, where α is the counter consumption of other computations in the transaction. For proof 2, the attacker modifies the free input to alter the execution path to path 2, causing the total counter consumption to be $\alpha + nC_2$. The malicious prover adjusts n such that $\alpha + nC_1 \leq C_{max}$ and $\alpha + nC_2 > C_{max}$. As a result, proof 1 will successfully validate the transaction, while proof 2 will demonstrate that the transaction fails due to an out-of-counters error.

In Polygon zkEVM, transactions are first soft-finalized on layer 2 and later hard-finalized on layer 1 once validity proofs are submitted and verified. Currently, Polygon zkEVM relies on centralized sequencer and zkProver to execute transactions and maintain its layer 2 soft-finalized state. However, by exploiting a dual-path attack, an adversary could submit a malicious proof (e.g., Proof 2 in Figure 3) to layer 1 before the official zkProver submits its valid proof (e.g., Proof 1 in Figure 3). This could cause the hard-finalized state maintained by layer 1 smart contract to deviate from the soft-finalized state maintained by layer 2 blockchain. As a result, Polygon zkEVM would be forced to roll back its layer 2 state to match the layer 1 state, creating a double-spending risk for applications such as cross-chain bridges that rely on soft-finalization.

Notably, the dual execution path vulnerability is common. FreeVer has identified 6 instances of this issue, all of which can be exploited to launch dual execution path attacks.

4 The FreeVer Framework

In this section, we first provide an overview of FreeVer, followed by a detailed illustration of its main components.

4.1 Overview

FreeVer is a formal verification tool to verify the soundness and completeness of free inputs in Polygon zkEVM. As shown in Figure 4, it consists of three main components: the zkASM semantics, the PSG constructor, and the property verifier.

zkASM semantics. Formal semantics of zkASM are defined in K [7] to facilitate reasoning about the constraints over free inputs. Apart from the syntax, rewrite rules, and configuration defined according to the official zkASM executor implementation [39,40], FreeVer also adds a command switcher to toggle the command evaluator. It also adds support for symbolic value introduction to enable symbolic reasoning. Moreover, it adopts optimizations to reduce unsolvable constraints.

PSG constructor. The PSG constructor executes the zkASM function to be verified as honest and malicious provers, respectively, and generates two aligned Prover State Graphs (PSGs) that encode all reachable states during execution. The aligned PSGs are then used by FreeVer to perform verification.

Property verifier. The property verifier takes as input the aligned PSGs of each zkASM function and verifies constraints over free inputs to identify potential soundness and completeness issues. For soundness verification, it compares the corresponding leaves of the aligned PSGs to determine if their states are consistent. Inconsistency indicates that the path is unsound, allowing malicious provers to reach a state that deviates from that of honest provers. For completeness, the verifier employs a heuristic: it asserts that applying constraints on free inputs does not cause unreachable nodes. If any node is not reachable, it suggests that the constraints on the free inputs are potentially incomplete.

4.2 zkASM Semantics

The design and implementation of the zkASM semantics aim to achieve the following goals:

- **G1:** It must encode the same semantics as the original zkASM, ensuring that executing our semantics produces the same constraints as those specified in PIL.
- **G2:** It must be capable of modeling the behavior of both honest and malicious provers.
- **G3:** It must support symbolic execution, enabling the exploration of all possible states during function execution.
- **G4:** It must produce simplified symbolic expressions to minimize the occurrence of unsolvable constraints for SMT solvers.

Given the lack of comprehensive documentation for zkASM, we derive its semantics by referencing the official implementations of the zkASM executor and command evaluator in C++ [39] and JavaScript [40], as well as the PIL constraints of the state machines [40]. Based on these references, we implement the corresponding semantics (G1). Moreover, to support executing both as an honest prover and a malicious

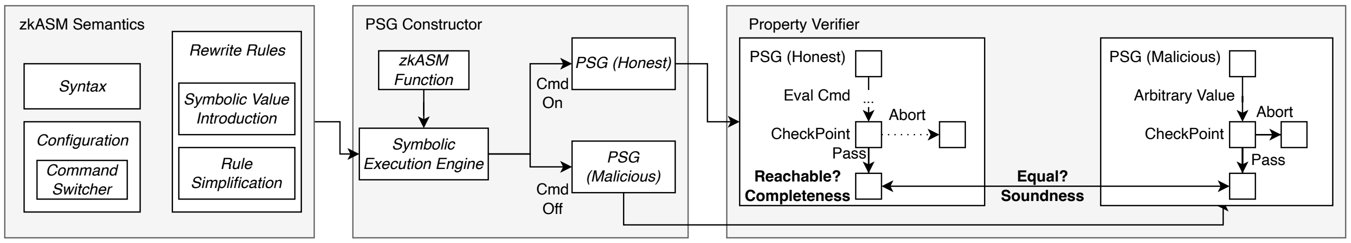

Figure 4: The workflow of FreeVer

prover, we implement a command switcher to toggle the behavior of free input commands. When the switch is on, the semantics evaluate the commands faithfully, like an honest prover. When it is off, the semantics supply a symbolic value representing arbitrary inputs to mimic malicious provers (G2). Subsequently, we add support for symbolic value introduction to facilitate reasoning with symbolic inputs (G3). Finally, we optimize the semantics to reduce the complexity of symbolic expressions and simplify rules that are incompatible with symbolic execution (G4).

4.2.1 Configuration

The configuration, as illustrated in Figure 5, is a data structure that represents the state of the zkASM virtual machine (Note that Polygon zkEVM modularizes the virtual machine into 13 state machines to optimize proof generation performance. However, for the verification task, viewing it as a single virtual machine is more intuitive and convenient). It consists of units named *cells*, which are labeled and can be nested.

The top-level `T` cell contains two sub-cells: the `k` cell, which is a default cell in the K Framework, and the `ctx` cell, which encapsulates the actual virtual machine state. The `k` cell is a special cell containing a sequence of computations.

The `ctx` cell encodes the state of the zkASM virtual machine and contains five sub-cells. The `pc` cell holds the program counter. The `mem` cell encodes memory as a mapping from addresses to 256-bit words. The `Rom` cell contains the parsed zkASM program. The `ProverInputs` cell provides essential information for EVM execution, including transaction inputs, block attributes, etc. The `pols` cell is of particular interest, as it contains a mapping from the name of each polynomial to its corresponding value, an element in the Goldilocks prime field [12]. This cell encapsulates all elements of the state that must be verified by the prover (i.e., registers, free inputs, and current PC value, etc.). For example, the 256-bit register `A` is encoded as eight 32-bit unsigned integers, fitting within the Goldilocks field ($2^{64}-2^{32}+1$), and labeled as `A0`, `A1`, ..., `A7` in the `pols` cell, respectively.

4.2.2 Syntax

The zkASM syntax in K is defined using a variation of Backus-Naur Form, directly translated from the `JISON` definition in the official zkASM compiler [38].

Listing 2 provides an overview of the syntax. It defines five types of statements for zkASM (line 1). A `Label` statement consists of an `Id` followed by a colon (line 3). It is used to mark a jump or call target. A `VarDef` statement consists of the keyword `VAR`, a scope definition (i.e., global or local), and the name of the variable (line 4). Similarly, the `ConstDef` statement is used to define a constant value (line 5). The `Step` statement includes most of operations in zkASM. It can contain an assignment (line 7), an opcode list (line 8), or both (line 6). The assignment assigns the value of the LHS expression of the operator => to each register listed at the RHS. The LHS expression is a linear combination of registers, constants, and free inputs (e.g., `3A + 1`). The opcode list includes zkASM instructions, such as arithmetic instructions (`ADD`, `SUB`, etc.), boolean instructions (`LT`, `SLT`, etc.), bitwise instructions (`AND`, `OR`, etc.), control flow instructions (`CALL`, `RETURN`, `JMP`, etc.), and constraint-checking instructions (`ASSERT`, `ARITH`, etc.). The `Command` statements are expressions wrapped with `${}` or `$${}`. `Commands` can either be standalone on separate lines or integrated into a `Step` statement, serving as the `RegsSum` part.

```
1  syntax Stmt ::= Label | VarDef | ConstDef | Step | Cmd
2
3  syntax Label    ::= Id ":"
4  syntax VarDef   ::= "VAR" Scope Id
5  syntax ConstDef ::= "CONST" Id "=" NExpr
6  syntax Step     ::= RegsSum "=>" Regs ":" OpList
7                    | RegsSum "=>" Regs
8                    | ":" OpList
9  syntax Cmd      ::= "$${" CmdExp "}"
10                   | "${" CmdExp "}"
```

Listing 2: zkASM syntax

4.2.3 Rewrite Rules

Rewrite rules encode the semantics of zkASM. Each rule rewrite the configuration matching the left part to the right part, following the matching logic [8,25]. FreeVer implements rules regarding program preprocessing, single step, command evaluation, and instruction interpretation.

Preprocessing. The execution of zkASM code begins with

$$\Big\langle \langle \text{KSequence} \rangle_{\text{k}} \Big\langle \langle \text{Int} \rangle_{\text{pc}} \langle \text{Addr} \mapsto W_{256} \rangle_{\text{mem}} \langle \text{ID} \mapsto E_{goldilocks} \rangle_{\text{pols}} \langle \ldots \rangle_{\text{Rom}} \langle \ldots \rangle_{\text{ProverInputs}} \Big\rangle_{\text{ctx}} \Big\rangle_{\text{T}}$$

Figure 5: zkASM configuration

preprocessing. The initial `k` cell contains the zkASM program (`$PGM`). '`:T`' represents the type of a term. Here, the program is of type `Stmts`, which is a list of `Stmt` separated by new lines.

```
syntax Stmts ::= List{Stmt, "\n"}
<k> $PGM :Stmts </k>
```

Preprocessing involves a loop that fetches the first statement from the list and processes it. Once all the statements have been processed (represented by `.Stmts`, which means an empty statement list), the execution loop begins. The preprocessing loop is defined with the following two rules:

```
rule <k> S:Stmt Ss:Stmts => #process(S) ~> Ss ... </k>
rule <k> .Stmts => #exec ... </k>
```

The '=>' symbol represents a reduction of a cell from the original expression on the LHS to the new expression on the RHS. Cells without '=>' are read-only and left unchanged. The function `#process` processes a statement according to its type and stores the corresponding information into subcells of the `Rom` cell. Specifically, it resolves the offsets for each `Label`, `Step`, and `Cmd`. `Step` statements and `Cmd` statements are stored to the `pgm` cell and the `cmd` cell, respectively. For `VarDef` statements, it allocates memory space and creates a mapping between variable identifiers and the allocated memory offsets. Similarly, `ConstDef` statements are processed to establish a mapping between constant variable names and their corresponding values.

Single step. Single step is also defined as a loop with two rules:

```
rule
  <k>
    #exec => PGM[PC] ~> CMDS[PC] orDefault .Command ~> #exec
  </k>
  <pc> PC => PC +Int 1 </pc>
  <pgm> PGM </pgm>
  <cmd> CMDS </cmd>
  requires PC >=Int 0 andBool PC <Int size(PGM)
rule <k> #exec => .K </k> [owise]
```

The first rule specifies that if the program counter (`PC`) is in a valid range and the next computation is the start of a new single step cycle (represented by `#exec`), then a step from the `PGM` cell and a command from the `CMD` cell (or an empty command denoted by `.Command`) at the address pointed to by the program counter are fetched. The content of `k` is then replaced with a computation sequence, separated by the '~>' symbol. This sequence includes the step, the command, and the start of the next cycle, to be processed sequentially. The second rule states that if `PC` points to an invalid address (specified with the `owise` attribute, meaning any other conditions that do not satisfy the first rule), the single step loop ends with empty computation (denoted by `.K`).

Step interpreting. The rule below defines the interpretation of a full step with both assignment and opcode lists. The interpretation contains three sequential computations. The first is to calculate the sum of all input registers (`IS`) with the `#sum` function. The second is to execute the opcodes (`OL`) one by one. The last is to assign the summed value to output registers (`RL`) with the `#assign` function. The '`...`' symbol means that the left content in the `k` cell remains unchanged. Note that the first '=>' symbol is the assign operator in zkASM, and the second is the rewrite operator in K.

```
rule <k>
    IS:RegsSum => RL:Regs : OL:OpList => #sum(IS) ~> OL ~> #assign(RL) ...
    </k>
```

Command evaluation. The command evaluation differs when acting as an honest prover and as a malicious prover. To model both behaviors, we first add a command switcher in the configuration:

```
configuration <T> ... <switcher> Bool </switcher> </T>
```

The two rules below show how the switcher controls command evaluation. When the switcher is on (i.e., the `switcher` cell stores true), the command is handled by the first rule, which fetches the command expression `E` and puts it at the beginning of the computation sequence. The expression is then handled by other rules that encode the semantics of command expressions to derive a result value. When the switcher is off, the command is handled by the second rule, in which a symbolic value `?X` (the question mark indicates it is a symbolic value introduced in this rule) representing an arbitrary value from malicious provers is provided as the evaluation result.

```
rule <k> ${ E:CmdExp } => E ... </k>
     <switcher> true </switcher>
rule <k> ${ _:CmdExp } => ?X:Int ... </k>
     <switcher> false </switcher>
```

Symbolic value introduction. Though executing zkASM code with concrete values allows us to verify certain paths with certain inputs, it cannot explore all the states. Therefore, we want to execute the code symbolicly. The syntax and rules below add support for introducing symbolic values.

```
1  syntax Stmt ::= "sym" "(" Reg, Int, Int ")"
2               | "sym" "(" Var, Int, Int ")"
3  rule <k> sym(R:Reg, I1:Int, I2:Int) => setReg(R, ?X) ...
     ↪ </k>
4       ensures ?X >=Int I1 andBool ?X <Int I2
5  rule <k> sym(R:Reg, I1:Int, I2:Int) => setVar(R, ?X) ...
     ↪ </k>
6       ensures ?X >=Int I1 andBool ?X <Int I2
```

Lines 1-2 define two new statements for introducing symbolic values with upper and lower bounds into registers and variables. Lines 3-6 specify the corresponding rules. For example, the statement `sym(A, 0, 2^256)` is interpreted as setting register A to a symbolic value in the range $[0, 2^{256})$.

```
1  // Converting Scalar X to Fea and back
2  // results in a complex expression
3  rule Scalar2Fea(X) => I %Int pow32 ,  I /Int pow32 %Int
     ↪ pow32, ..., I /Int pow224 %Int pow32
4  rule Fea2Scalar(F) => F[0] +Int F[1] *Int pow32 +Int ...
     ↪ +Int F[7] *Int pow224
5
6  // optimized rules
7  rule Scalar2Fea(X) => fea(X)
8  rule Fea2Scalar(fea(X)) => X
9  // Calculate only when neccessary
10 rule fea(I:Int)[0] => I            %Int pow32
11 rule fea(I:Int)[1] => I /Int pow32  %Int pow32
12 ...
13 rule fea(I:Int)[7] => I /Int pow224 %Int pow32
```

Listing 3: Lazy evaluation

4.2.4 Optimization

There are two main obstacles for SMT solvers in symbolic execution. The first arises from word composition and decomposition. zkASM operates on 256-bit integers but stores them as eight 32-bit field elements to facilitate proving in zkSTARK [12]. As a result, each write and read of registers involves breaking down integers (Line 3 of Listing 3) and reassembling them (Line 4 of Listing 3). However, the K Framework cannot recognize that decomposing an integer and then recomposing it yields the same value. Instead, it generates a complex expression in a straightforward manner. Consequently, the complexity of symbolic expressions increases with each read and write operation, quickly rendering them unresolvable for SMT solvers. The second challenge stems from commands that utilize algorithms containing loops, which can cause symbolic execution to fail when conditioned on symbolic values. To address these issues, we optimize symbolic execution using lazy evaluation and directly provide symbolic results with constraints for algorithms involving loops.

Lazy evaluation. To mitigate the complexity introduced by word composition and decomposition, FreeVer leverages the lazy evaluation in Listing 3. Specifically, when converting a 256-bit word to field element arrays, it stores the value as `fea(X)`, where `X` is a symbolic expression and `fea` represents the breakdown of the parameter into eight 32-bit integers (line 7). The calculation is deferred and only performed when necessary (e.g., if `fea(X)[0]` is accessed, only the lowest 32 bits are extracted) (lines 10-13). As zkASM operates on 256-bit words, the value is usually fetched as a 256-bit integer. In such cases, FreeVer simply removes the `fea` wrapper and returns the value `X` (line 8).

Optimizing complex operations. In zkASM, operations such as field element inversion, square root extraction, and elliptic curve operations are challenging to handle with symbolic execution. This is due to the computation process potentially containing loops conditioned on symbolic values, which prevents symbolic execution from deriving the correct result. To address this, we optimize the semantics by directly providing the correctly constrained symbolic values for these operations based on mathematical facts, rather than calculating them as we would in concrete algorithms.

Table 1 presents the summary of the optimized semantics for free input commands. Specifically, FreeVer simplifies 11 function calls by replacing the original computations with optimized rewrite rules. These rewrite rules provide a symbolic result directly, accompanied by the corresponding constraints on these symbolic results. Each row in the table describes the operation, its corresponding symbolic result, and the constraint applied to ensure the validity of the result. Note that all operations used in the table are field operations. For example, the operation `inverseFpEc(I)` computes the modular multiplicative inverse in the `FpEc` field. The zkASM Executor performs Montgomery inversion to calculate the result, which is difficult for symbolic exploration. FreeVer optimizes this by substituting the result with a symbolic value $?X$ under the constraint $I \cdot ?X = 1$. The corresponding rule is:

```
1  rule <k> inverseFpEc(I) => ?X ... </k>
2       ensures I *Int ?X modInt FPEC ==Int 1 andBool ?X
          ↪ >=Int 0 andBool ?X <Int FPEC
```

The optimized operations span several cryptographic functions, including modular inverses in various fields (e.g., `inverseFpEc`, `fpBN254inv`), modular square roots (e.g., `sqrtFpEc`), and elliptic curve point operations (e.g., `xAddPointEc`, `yDblPointEc`).

4.3 Prover State Graph Constructor

Given the state machine (i.e., the configuration in Figure 5) and the state transition rules (i.e., the rewrite rules), FreeVer can explore the reachable states of both honest and malicious provers. To facilitate the comparison between the two types of provers, FreeVer leverages a PSG constructor to produce aligned prover state graphs.

A prover state graph $G(V, E)$ is a directed acyclic graph that encodes all reachable virtual machine states during the execution of zkASM code, where a node $n \in V$ represents

Table 1: Optimized semantics for free input commands

`Operation`	Description	Result	Constraints
`inverseFpEc(I)`	Compute modular multiplicative inverse in `FpEc` field	?X	$I \cdot ?\mathrm{X} = 1$
`inverseFnEc(I)`	Compute modular multiplicative inverse in `FnEc` field	?X	$I \cdot ?\mathrm{X} = 1$
`fpBN254inv(I)`	Compute modular multiplicative inverse in `fpBN254` field	?X	$I \cdot ?\mathrm{X} = 1$
`fp2InvBN254x(I1, I2)`	Real part of inverse of (I_1, I_2)	?X	$?\mathrm{I} \cdot (I_1^2 + I_2^2) = 1$, $?\mathrm{X} = I_1 \cdot ?\mathrm{I}$
`fp2InvBN254y(I1, I2)`	Imaginary part of inverse of (I_1, I_2)	?Y	$?\mathrm{I} \cdot (I_1^2 + I_2^2) = 1$, $?\mathrm{Y} = (\mathrm{BN254P} - I_2) \cdot ?\mathrm{I}$
`sqrtFpEc(I)`	Compute modular square root in `FpEc` field	?X	$?\mathrm{X} \cdot ?\mathrm{X} = I$
`sqrtFpEcParity(I, P)`	Compute modular square root in `FpEc` field with parity matching `P`	?X	$?\mathrm{X} \cdot ?\mathrm{X} = I$, $(?\mathrm{X} \oplus P) \wedge 1 = 0$
`xAddPointEc(X1, Y1, X2, Y2)`	Add points (X_1, Y_1) and (X_2, Y_2), output x-coordinate	?X3	$?\mathrm{Inv} \cdot (X_2 - X_1) = 1$, $?\mathrm{S} = (Y_2 - Y_1) \cdot ?\mathrm{Inv}$, $?\mathrm{X3} = ?\mathrm{S} \cdot ?\mathrm{S} - X_1 - X_2$
`yAddPointEc(X1, Y1, X2, Y2)`	Add points (X_1, Y_1) and (X_2, Y_2), output y-coordinate	?Y3	$?\mathrm{Inv} \cdot (X_2 - X_1) = 1$, $?\mathrm{S} = (Y_2 - Y_1) \cdot ?\mathrm{Inv}$, $?\mathrm{X3} = ?\mathrm{S} \cdot ?\mathrm{S} - X_1 - X_2$, $?\mathrm{Y3} = ?\mathrm{S} \cdot (X_1 - ?\mathrm{X3}) - Y_1$
`xDblPointEc(X1, Y1)`	Double point (X_1, Y_1), output x-coordinate	?X3	$?\mathrm{Inv} \cdot (2 \cdot Y_1) = 1$, $?\mathrm{S} = (3 \cdot X_1^2) \cdot ?\mathrm{Inv}$, $?\mathrm{X3} = ?\mathrm{S} \cdot ?\mathrm{S} - 2 \cdot X_1$
`yDblPointEc(X1, Y1)`	Double point (X_1, Y_1), output y-coordinate	?Y3	$?\mathrm{Inv} \cdot (2 \cdot Y_1) = 1$, $?\mathrm{S} = (3 \cdot X_1^2) \cdot ?\mathrm{Inv}$, $?\mathrm{X3} = ?\mathrm{S} \cdot ?\mathrm{S} - 2 \cdot X_1$, $?\mathrm{Y3} = ?\mathrm{S} \cdot (X_1 - ?\mathrm{X3}) - Y_1$

a single state and an edge $e \in E$ represents a rewrite rule that transitions the state of the source node to the state of the target node. The PSG constructor constructs such graphs for honest and malicious provers, respectively. It then aligns the two PSGs by establishing a correspondence between the key nodes of the two graphs. The aligned PSGs facilitate the verification by the property verifier.

PSG construction. The constructor provides the zkASM semantics and the function to be verified as inputs to the symbolic execution engine, which executes the zkASM function according to the semantics. It runs under two settings: one with the command switcher on and another with it off, to model the different behaviors of honest and malicious provers. During execution, it records the result of each rewrite (i.e., the configuration) as a new node, with the rewrite rules forming the edges from the source node to the new node.

PSG optimization. Since each node encodes a state of the entire virtual machine, memory consumption can become prohibitively large as the size of the PSG grows. To address this issue, we optimize the PSG by keeping only the nodes of interest. Specifically, we retain the source and target nodes of branching rules, checkpoint rules, and leaf nodes that represent the end of execution. Branching rules are kept to align honest and malicious PSGs, checkpoint rules apply constraints and are kept for completeness verification, and leaf nodes are retained for soundness verification. The edges connecting to the removed nodes are merged. For example, consider three nodes n_1, n_2, n_3 connected by two edges e_1, e_2. If node n_2 is removed, the edges e_1 and e_2 are merged into a new edge e', which represents the sequential application of the two rules represented by e_1 and e_2. All PSGs referred to later are the optimized PSGs, unless explicitly stated otherwise.

PSG alignment. FreeVer constructs PSGs using the honest prover's state as a specification to check against the corresponding malicious prover's state. However, the correspondence cannot be directly established due to differences between the two PSGs in terms of the number of nodes and paths. These differences arise for two reasons. Firstly, when handling free input commands, honest provers take multiple rewrite steps to compute the free inputs, while malicious provers use a single rewrite step to provide arbitrary values. Secondly, the PSG for malicious provers contains more branches than that for honest provers. This difference arises because honest provers explore only valid execution paths, while malicious provers can explore paths corresponding to invalid input data. To address these issues, FreeVer performs alignment of the PSGs.

Algorithm 1: Align PSGs

```
   Input  : PSGs for honest ($G_h$) and malicious ($G_m$) provers
   Output : Tuple of aligned nodes
1  Function align($G_m$, $G_h$):
2      aligned ← $\emptyset$
3      pending ← $\{(G_m.\mathrm{root}, G_h.\mathrm{root})\}$
4      while pending $\neq \emptyset$ do
5          $(n_m, n_h)$ ← pending.pop()
6          $b_m$ ← proceedToBr ($G_m$, $n_m$)
7          $b_h$ ← proceedToBr ($G_h$, $n_h$)
8          if $isLeaf(b_m) \wedge isLeaf(b_h)$ then
9              aligned ← aligned $\cup \{(m,h)\}$
10         $T_m$ ← resolveTargets ($G_m$, $b_m$)
11         $T_h$ ← resolveTargets ($G_h$, $b_h$)
12         foreach $(m,h) \in matchRules(T_m, T_h)$ do
13             pending ← pending $\cup \{(m,h)\}$
14     return aligned
```

FreeVer leverages Algorithm 1 for aligning PSGs. It takes the two PSGs as inputs and outputs the set of aligned leaves. The algorithm maintains a pending set, which contains corresponding node pairs in the two PSGs. The pending set is initialized with the roots of the two PSGs (line 3). It then uses a loop to traverse the two PSGs simultaneously. Specifically, in each cycle, it pops a pair of nodes from the pending set and starts traversing the PSGs at the two nodes, respectively, with the `procceedToBr` function (lines 6-7). The function simply follows the edges in the PSG until it encounters an edge that represents a rule causing branches (e.g., `JMPN`, `JMPC`, etc.) or reaches a leaf node, at which point it returns the node where it stopped. If the two returned nodes are leaves, it indicates that a pair of aligned leaves has been found (lines 8-9). If they are branch points, the algorithm resolves the targets of the branches (lines 10-11) and decides the corresponding targets by checking if the applied rules are the same. Aligned target nodes are then added to the pending set for further exploration.

4.4 Property Verifier

In this section, we describe how FreeVer verifies the soundness and completeness of zkASM functions.

Algorithm 2 illustrates the soundness verification process of zkASM code. It takes as input the optimized PSGs for honest provers G_h and malicious provers G_m and outputs the soundness of each successful execution path. The `VerifySoundness` function checks each aligned pair of nodes in the PSGs (Algorithm 1) using the `equal` function. The function verifies that the two aligned nodes have equivalent states. A path is sound if the corresponding nodes are equal; otherwise, it is not.

Algorithm 2: Verify Soundness

```
Input  : PSGs for honest ($G_h$) and malicious ($G_m$) provers
Output : Soundness results of paths.
1  Function VerifySoundness($G_h$, $G_m$):
2      aligned ← align ($G_h$, $G_m$)
3      result ← $\emptyset$
4      foreach $P_h, P_m \in aligned$ do
5          if equal $(P_h, P_m) = true$ then
6              result ← result $\cup \{P_m \mapsto Sound\}$
7          else if equal $(P_h, P_m) = false$ then
8              result ← result $\cup \{P_m \mapsto Unsound\}$
9          else
10             result ← result $\cup \{P_m \mapsto Unknown\}$
11     return result
```

State equivalence. If a path in the malicious PSG is sound, the final node N_m of this path must have an equal state to its corresponding node N_h in the honest PSG. In other words, the malicious prover cannot derive a state that deviates from the state of the honest prover. Any discrepancy indicates that the constraints on this path are unsound.

A node N is represented as a matching logic predicate:

$$\mathcal{P}_N = And(\mathcal{P}_{\text{constraints}}(\mathcal{V}_{\text{sym}}), \mathcal{P}_{\text{config}}(\mathcal{V}_{\text{concrete}}, \mathcal{V}_{\text{sym}})),$$

where $\mathcal{P}_{\text{config}}$ defines an instantiation of the configuration using both concrete values $\mathcal{V}_{\text{concrete}}$ and symbolic values $\mathcal{V}_{\text{sym}}$, and $\mathcal{P}_{\text{constraints}}$ imposes constraints on $\mathcal{V}_{\text{sym}}$.

To prove that two corresponding nodes, N_m and N_h, are equal, we need to establish the following equivalence:

$$\mathcal{P}_{\text{constraints}_{N_m}} \wedge \mathcal{P}_{\text{config}_{N_m}} \iff \mathcal{P}_{\text{constraints}_{N_m}} \wedge \mathcal{P}_{\text{config}_{N_h}}$$

Given that N_m always has the same or looser constraints than N_h (as N_h imposes stricter constraints on the free inputs), FreeVer only needs to verify whether:

$$\mathcal{P}_{\text{constraints}_{N_m}} \wedge \mathcal{P}_{\text{config}_{N_m}} \implies \mathcal{P}_{\text{constraints}_{N_m}} \wedge \mathcal{P}_{\text{config}_{N_h}}$$

The inverse always holds. This implication can be transformed into a Z3 formula by attempting to prove that the following formula is unsatisfiable:

$$\left(\bigwedge_{j=1}^{x} C_{m_j}\right) \wedge \left(\bigwedge_{k=1}^{y} C_{h_k}\right) \wedge \left(\bigvee_{i=1}^{n} p_{m_i} \neq p_{h_i}\right)$$

where C_{m_j} and C_{h_k} are the individual constraints from N_m and N_h, respectively; p_{m_i} and p_{h_i} represent the corresponding elements in their configurations.

If the SMT solver returns *unsat*, it indicates that no symbolic values exist that would cause the configurations to differ. Therefore, the two states can be considered equal.

4.4.1 Completeness Verification

FreeVer verifies the completeness of a zkASM function with a heuristic as in Algorithm 3. It checks that for every successful execution path in the PSG of honest provers, every target node of edges that apply a constraint rule is reachable, meaning there exists some input that satisfies the constraints applied by the edge. More specifically, it iterates over each edge E in G_h. For edges with passing rule identifiers (i.e., $E.rule_{id} \in R_{pass}$), it attempts to solve the target node constraints $E.target_{constraints}$. If the solver finds these constraints unsatisfiable, this indicates potential overconstrained vulnerabilities or misimplementations in the zkASM executor or command evaluation, leading to incompleteness at that edge. Note that this heuristic may not always detect completeness issues, in some cases, unreachable nodes could also result from dead code. We think reporting dead code is also meaningful, as it helps developers to optimize the code.

Algorithm 3: Verify Completeness

```
Input  : PSG for honest provers $G_h$
Output : $Res \in \{Complete, Incomplete, Unknown\}$ for each
         checkpoint
1  $R_{pass}$ ← passing rule ids of constraint-applying instructions.
2  foreach Edge $E \in G_h$ do
3      if $E.rule_{id} \in R_{pass}$ then
4          $r$ ← solve ($E.target_{constraints}$)
5          if $r = unsat$ then
6              output (Incomplete at Edge E)
7          if $r = unknown$ then
8              output (Unknown at Edge E)
9          if $r = sat$ then
10             output (Complete at Edge E)
```

5 Evaluation

In this section we describe the results of the evaluation derived by answering the following three research questions (RQs).

- RQ1: Can FreeVer correctly identify disclosed free inputs related vulnerabilities?
- RQ2: How effective are the design choices of FreeVer in improving the accuracy of verification?
- RQ3: Can FreeVer find new vulnerabilities in Polygon zkEVM?

Benchmarks. To evaluate FreeVer and address these research questions, we collect two datasets, $\mathcal{D}_{disclosed}$ and $\mathcal{D}_{rom}$. $\mathcal{D}_{disclosed}$ contains 9 free input-related vulnerabilities and their fixes from 7 public audit reports [14, 28–32, 43] of the Polygon zkEVM ROM. To the best of our knowledge, $\mathcal{D}_{disclosed}$ is the most comprehensive dataset of vulnerabilities related to free inputs in Polygon zkEVM. The summary of these disclosed vulnerabilities is presented in Table 2. $\mathcal{D}_{rom}$ comprises 55 functions extracted from the official implementation of the Polygon zkEVM ROM (fork.5) [41].
Experimental settings. All experiments are conducted on a server equipped with dual 96-core AMD EPYC 9654 CPUs and 1.5 TB of memory. We implement FreeVer using K Framework v7.151 [44] and utilize z3 v4.13.3 [24] as the SMT solver. All verification tasks are executed in parallel, with a maximum of 16 parallel processes per task and a memory limit of 64 GB.

Table 2: Disclosed Vulnerabilities

Function ID	Root Cause	Issue	Source
IDENTITY	Missing free input constraints	Soundness	SpearBit
computeGasSendCall	Missing free input constraints	Soundness	SpearBit
divARITH	Incorrect remainder check	Soundness	SpearBit
invFp2BN254	Missing range check	Soundness	Verichains
offsetUtil	Missing range check at some paths	Soundness	SpearBit
opCALLDATACOPY	Missing free input constraints	Soundness	SpearBit
opCREATE2	Missing free input constraints	Soundness	SpearBit
saveMemGAS	Missing free input constraints	Soundness	SpearBit
subFpBN254	Misuse of free input	Soundness	Verichains

5.1 Correctness of FreeVer

To address RQ1, we apply FreeVer to $\mathcal{D}_{disclosed}$. The results presented in Table 3 demonstrate that FreeVer correctly constructs and aligns PSGs and successfully detects all the vulnerabilities in $\mathcal{D}_{disclosed}$.
PSG construction. The average size of the original PSGs for honest provers is 744 nodes, while for malicious provers, it is 719 nodes. After optimization, these sizes are significantly reduced to 64 nodes and 69 nodes, respectively. The larger original size of honest provers' PSGs is due to the additional rewrite steps required to compute free inputs, whereas malicious provers often complete this process in a single step using symbolic values. In contrast, the optimized PSGs for malicious provers are slightly larger because they include more failing paths. Despite these differences, the successful paths in the PSGs of honest and malicious provers consistently exhibit a one-to-one correspondence. This alignment serves as the foundation for soundness verification, enabling direct comparison between the states of honest and malicious provers.
Soundness verification. The soundness verification results report 13 unsound paths in the 9 buggy versions, all of which we confirm to be true positives. For the fixed versions, all functions except `subFp2BN254*` are reported as sound. After manually analyzing the implementation details, we find that `subFp2BN254*`, which performs subtraction in the BN254 prime field, assumes the input lies within the range $[0, BN254_P)$ (where $BN254_P$ is the BN254 modulus), delegating this constraint check to the caller. This design choice likely aims to improve performance by avoiding redundant checks. However, during verification, FreeVer provides symbolic inputs in the range $[0, 2^{256})$, which causes the function to be flagged as unsound because it does not fully constrain its free inputs in this case. FreeVer can verify the function as sound if the input range is adjusted appropriately. Additionally, 12 paths are reported as unknown, all of which we confirm to be sound. These cases result from SMT solver timeouts. In summary, soundness verification achieves an accuracy of 76%, identifying 25 sound paths and 13 unsound paths out of a total of 50 paths. The average time for verification is 71.08 seconds.
Completeness verification. The results show that all functions are correctly reported as complete, except for one path in `offsetUtil`, which is flagged as incomplete. Upon investigation, we find that this path corresponds to dead code, which is removed in the fixed version. The average time for completeness verification is 86.50 seconds.

Answer to RQ1: FreeVer can detect all disclosed free input related vulnerabilities.

5.2 Ablation Study

To the best of our knowledge, FreeVer is the first tool capable of detecting free input-related vulnerabilities in Polygon zkEVM. Since no existing baselines address this problem, we evaluate FreeVer against the following three ablations to assess its design:
• FreeVer-Naive: This ablation does not apply the optimizations described in § 4.2.4 and does not construct PSGs. Instead, it directly executes zkASM functions to derive symbolic expressions and constraints of the outputs. It then verifies soundness by asserting that only one assignment of the free inputs and outputs satisfies the constraints.
• FreeVer-Opt: Similar to FreeVer-Naive, but applies the optimizations described in § 4.2.4.

Table 3: Results of running FreeVer against $\mathcal{D}_{disclosed}$. * means the fixed version; H and M denote honest and malicious PSGs, respectively.

Function ID	Prover State Graph Construction										Completeness Verification				Soundness Verification			
	Size		Size Opt		Paths		Successful Paths		Time (s)		Completeness			Time (s)	Soundness			Time (s)
	H	M	H	M	H	M	H	M	H	M	C	I	U		S	U	U(Unknown)	
IDENTITY	426	423	40	42	5	7	2	2	166.76	151.56	2	0	0	86.5	1	1	0	16.54
IDENTITY*	804	801	47	54	5	10	1	1	385.58	325.21	2	0	0	104.25	1	0	0	19.88
computeGasSendCall	316	316	16	18	2	4	2	2	117.16	98.93	2	0	0	40.17	0	2	0	16.13
computeGasSendCall*	438	438	36	40	4	8	2	2	199.79	203.63	2	0	0	82.61	2	0	0	17.08
divARITH	699	693	34	38	4	8	2	2	218.4	195.36	2	0	0	72.24	1	1	0	21.29
divARITH*	699	693	34	38	4	8	2	2	220.65	194.91	2	0	0	74.03	2	0	0	17.64
invFp2BN254	2,050	1,940	234	256	26	48	12	12	861.44	669.34	12	0	0	553.93	4	4	4	377.13
invFp2BN254*	2,418	2,308	298	320	42	64	12	12	972.67	726.03	12	0	0	733.63	4	0	8	547.29
offsetUtil	130	122	22	22	4	4	3	3	45.86	49.25	2	1	0	73.25	2	1	0	16.27
offsetUtil*	261	257	36	38	5	7	3	3	106.55	98.27	3	0	0	115.81	3	0	0	18.36
opCALLDATACOPY	269	259	22	22	3	3	1	1	106.34	92.16	1	0	0	48.65	0	1	0	15.74
opCALLDATACOPY*	1,257	1,234	53	58	6	9	1	1	747.66	599.33	1	0	0	109.64	1	0	0	20.02
opCREATE2	715	708	43	50	4	9	1	1	335.27	298.3	1	0	0	82.56	0	1	0	18.43
opCREATE2*	714	711	41	48	4	9	1	1	316.4	266.78	1	0	0	82.45	1	0	0	17.41
saveMemGas	291	265	38	42	3	7	1	1	243.85	163.22	1	0	0	144.16	0	1	0	76.12
saveMemGas*	1,305	1,278	73	80	8	13	1	1	1,139	615.31	1	0	0	158.77	1	0	0	27.60
subFpBN254	207	170	24	24	4	4	1	1	111.01	113.92	1	0	0	32.38	0	1	0	17.80
subFpBN254*	392	332	54	54	10	10	2	2	198.49	145.3	2	0	0	57.96	0	2	0	21.47

• FreeVer-PSG: Verifies soundness in the same way as FreeVer but does not apply optimizations.

We evaluate these ablations on $\mathcal{D}_{disclosed}$, focusing exclusively on soundness verification, as the ablations cannot verify completeness without PSGs for honest provers.

To further evaluate the effectiveness of FreeVer, we also designed a baseline tool called FIFuzz, which employs fuzzing techniques to detect soundness vulnerabilities in zkASM code. Specifically, FIFuzz tests zkASM programs by generating random invalid free inputs that deliberately deviate from the valid inputs provided by the command evaluator. FIFuzz flags a soundness vulnerability if the system accepts these malformed free inputs. We implemented this fuzzer based on JsFuzz [1] and tested all buggy functions within $\mathcal{D}_{disclosed}$ for one hour.

The performance of each tool is summarized in Table 4. FreeVer-Naive and FreeVer-Opt do not construct PSGs, meaning they cannot distinguish constraints and outputs across different paths. Therefore, they verify soundness at the function level rather than the path level. In contrast, FreeVer-PSG and FreeVer construct PSGs, enabling soundness verification at a finer granularity by individually checking successful paths.

FIFuzz fails to detect any bugs in the buggy functions from $\mathcal{D}_{disclosed}$. Moreover, since FIFuzz cannot guarantee that the target functions are bug-free, it cannot confirm their soundness. These results demonstrate that conventional fuzzing is ineffective at identifying bugs related to free inputs.

FreeVer-Naive demonstrates the lowest accuracy (16.67%), identifying only 2 sound functions and 1 unsound function. FreeVer-PSG performs slightly better with an accuracy of 33.33% (identifying 4 sound paths out of 12). However, due to the lack of optimizations, symbolic execution timeouts occur for 11 functions during PSG construction, limiting the number of verifiable buggy and fixed functions to 4 and 3, respectively. FreeVer-Opt, which applies optimizations, achieves a significantly improved accuracy of 50.00% (identifying 5 sound functions and 4 unsound functions). FreeVer outperforms all ablations, with an accuracy of 76.00%, identifying 25 sound paths and 13 unsound paths.

Ablations without optimizations (i.e., FreeVer-Naive and FreeVer-PSG) require approximately 10 times more execution time due to the complexity of symbolic expressions. FreeVer demonstrates exceptional efficiency, taking an average of 71.08 seconds for verification, while other ablations suffer from increased complexity and longer execution times.

Table 4: Ablation results

	Sound	Unsound	Unknown	Accuracy	Exec/Construction Time (s)	Verify Time (s)
FIFuzz	0	0	9	0.00&	0	3600
FreeVer-Naive	2	1	15	16.67%	726.58	698.97
FreeVer-Opt	5	4	9	50.00%	71.55	108.50
FreeVer-PSG	4	0	8	33.33%	6,113.2/473.98	909.76
FreeVer	25	13	12	76.00%	360.72/278.16	71.08

Answer to RQ2: Optimizations and PSG construction are effective designs that improve the accuracy of verification.

5.3 Finding New Vulnerabilities

We run FreeVer against $\mathcal{D}_{rom}$ to assess its ability to identify new vulnerabilities. During PSG construction, we set an upper bound of 512 nodes for the prover state graph. In some cases, the functions contain loops with up to 256 iterations or loops without explicit upper bounds, restricted only by counters. To address these, we either provide concrete inputs to explore the loop once or manually extract the relevant code for verification.

The results show that out of the 55 functions, FreeVer successfully constructs PSGs for 47 functions. Four functions fail because they propagate symbolic values into `MLOAD` addresses, resulting in state splits up to 0x20000 (the memory size of zkEVM), making the PSG prohibitively large. Therefore, PSG construction of these functions fail because of out-of-memory. Three additional functions depend on these failing functions and thus fail as well. One function fails due to the complexity introduced by 502 sequential function calls, where the complexity of the symbolic expressions grows with execution and eventually leads to performance issues with K Framework.

After removing duplicates, FreeVer reports 6 soundness issues across 6 functions and 3 completeness issues. All the soundness issues are vulnerable to the dual execution path attack. Of the completeness issues, one is a true positive, while the other is caused by dead code. All vulnerabilities are reported to Polygon through the bug bounty program and have been confirmed as *high impact* bugs.

Case study: completeness bug. The completeness vulnerability reported in the `sqrtFpEc` function 4 arises from an inconsistency in handling the free input when no square root is found. The function uses free input command to compute the square root of a field element (line 3) and expects `%FPEC_NON_SQRT` (defined as $2^{256} - 1$ in zkASM) when no square root exists (lines 4-5). However, the command evaluator returns 0 when no square root is found. This mismatch between the expected value $2^{256} - 1$ and the actual return value 0 can lead to proof generation failures, even for valid inputs. Furthermore, this discrepancy could be exploited to launch a Denial-of-Service attack.

```
1  sqrtFpEc:
2    C                :MSTORE(sqrtFpC_tmp)
3    ${var _sqrtFpEc_sqrt = sqrtFpEc(C) } => A,C
     ↪  :MSTORE(sqrtFpC_res)
4    %FPEC_NON_SQRT => B
5    $                     :EQ,JMPC(sqrtFpEc_End)
6    ...
7  sqrtFpEc_End:
8    :RETURN
```

Listing 4: Completeness vulnerability in sqrtFpEc

Case study: soundness bug. The soundness vulnerability in `mulPointEc` arises from the underconstrained free input at line 4. The code snippet in Listing 5 performs elliptic curve point addition depending on whether the two points are the same. If they are the same, the function follows the `mulPointSameInitialPoints` branch; otherwise, it follows the `mulPointDiffInitialPoints` branch. These two branches perform different computations and thus consume a different number of counters. In the `mulPointSameInitialPoints` branch, the conditional expression for the free input is checked with `ASSERT` (note that `ASSERT` checks that `A == OP`, where `OP` is `C` at line 5). However, a similar check is missing in the other branch. As a result, malicious provers can provide 0 for this free input even when `A == C`. By examining the constraints of the instruction `ARITH_ECADD_DIFFERENT` [40] used to verify the free inputs in this branch, we find that data can be forged to pass the check, even when `A == C`. Therefore, The function exists a dual execution path vulnerability and is vulnerable to the attack introduced in § 3.

```
1  mulPointEc:
2    ...
3    ; check p1.x == p2.x
4    ${A == C}      :JMPZ(mulPointDiffInitialPoints)
5    C              :ASSERT
6
7    ; check p1.y == p2.y
8    D => A
9    $              :EQ,JMPC(mulPointSameInitialPoints)
10   1n
     ↪  :MSTORE(mulPointEc_p12_empty),JMP(mulPointEc_loop)
11
12 mulPointSameInitialPoints:
13   ; p2 == p1
14   0n
     ↪  :MSTORE(mulPointEc_p12_empty)
15   $ => A                    :MLOAD(mulPointEc_p1_x)
16   ${xDblPointEc(A,B)} => E  :MSTORE(mulPointEc_p12_x)
17   ${yDblPointEc(A,B)}       :ARITH_ECADD_SAME,
     ↪  MSTORE(mulPointEc_p12_y),JMP(mulPointEc_loop)
18
19 mulPointDiffInitialPoints:
20   ; p2.x != p1.x ==> p2 != p1
21   0n
     ↪  :MSTORE(mulPointEc_p12_empty)
22   ${xAddPointEc(A,B,C,D)} => E
     ↪  :MSTORE(mulPointEc_p12_x)
23   ${yAddPointEc(A,B,C,D)}       :ARITH_ECADD_DIFFERENT,
     ↪  MSTORE(mulPointEc_p12_y)
24   ...
```

Listing 5: Soundness vulnerability in mulPointEc

Answer to RQ3: FreeVer finds 6 soundness issues and 1 completeness issue in Polygon zkEVM, all of which are confirmed as high impact vulnerabilities.

6 Discussion

Limitations. First, we observe that K Framework currently struggles to handle symbolic bitwise operations (`AND`, `OR`, `XOR`) in its symbolic execution engine. However, free input commands use `AND` to mask higher bits of a 256-bit word (e.g., $W \ \& \ (2^n - 1)$). As a workaround, we translate `AND` operations into modulo operations (e.g., $W \ \% \ 2^n$). This approach is not complete, but it works for all functions verified in our evaluation. In cases where translation is not feasible, FreeVer may produce false positives, flagging the code as unsound or incomplete. This limitation could be resolved once K Framework addresses this issue. Second, FreeVer produces false negatives when the SMT solvers fail to solve the constraints. This issue cannot be fully addressed because SMT solvers are incomplete for polynomial equations over finite fields [3].

However, our evaluations show that by symbolically executing with optimized semantics and constructing PSGs to analyze at the path granularity, FreeVer can significantly reduce the occurrence of unsolvability.

Applicability to other zk rollups. While FreeVer is implemented and evaluated for Polygon zkEVM, the methodology of extracting specifications with executable semantics and constructing PSGs for property verification can also be applied to check non-deterministic execution in other zk rollups (e.g., the *hints* in Cairo [11]). In FreeVer, only the zkASM semantics are specialized for Polygon zkEVM; other components are language-agnostic (as they are based on K Framework [7], which provide tool sets that are language-agnostic) and can be reused. To apply FreeVerto a new zk rollup, users only need to provide the corresponding semantics.

Transition to other formalization tools. While FreeVer currently uses K for zkASM semantics, our core contributions—the abstract machine (i.e., the configuration), the state transition system (i.e., the rewrite rules), and the symbolic exploration optimizations—are tool-agnostic. These designs can be adapted to other frameworks (such as Coq or Lean4) with only syntactic adjustments. Additionally, FreeVer could be extended to support alternative formal verification tools, though we leave this as future work.

7 Related Work

Verification tools for Circom. Previous studies [16, 20, 23] primarily focus on detecting vulnerabilities in Circom, a domain-specific language for implementing zkSNARK circuits. QED^2 [23] is a tool designed to verify the soundness of a circuit. It combines a uniqueness constraint propagation approach with SMT solvers to prove that, under the same input signals, there is a unique assignment of output signals that satisfies the circuit. CODA [20] presents a refinement type system for Circom and leverages formal verification to identify potential violations. CIVER [16] offers constructions for specifying pre- and post-conditions, along with a scalable, modular technique for verifying properties of constraint systems expressed as sets of polynomial equations over a large prime field. These tools rely either on verifying uniqueness of output (which we find insufficient to capture all security issues, e.g., the dual execution path vulnerability in Polygon zkEVM) [23] or additional information manually provided by developers (but even experts can overlook constraints that cause vulnerabilities) [16, 20] to detect vulnerabilities. Consequently, they fail to correctly capture the root cause of security issues in zero-knowledge proof systems: the inconsistency between witness generation and the constraints imposed on the witness. In contrast, FreeVer adopts a novel approach in which a concrete specification is automatically extracted from honest prover PSGs and used to verify the constraints of free inputs. This approach enables FreeVer to identify common underconstrained vulnerabilities, as well as the novel dual execution path vulnerability. Furthermore, FreeVer provides a heuristic to detect completeness issues, which are not explored in previous studies.

Formal semantics in K. K Framework [7] is a tool for defining formal semantics of programming languages. A key feature of K semantics is its executability, based on Matching Logic [8, 25–27, 33]. The study in [9] presents a complete semantics for the x86-64 instruction set, enabling formal verification of all x86-64 executables. Similarly, K Framework has been used to define the semantics of EVM [7] and Solidity [17], facilitating formal verification of smart contracts. A key difference between FreeVer and these works is that, in addition to providing semantics for formal verification, FreeVer can also automatically generate specifications. Furthermore, FreeVer supports and optimizes symbolic execution, allowing verification of all possible reachable states.

8 Conclusion

In this study, we uncover a novel dual execution path vulnerability and introduce a powerful attack against it. We propose FreeVer, a framework for formally verifying the soundness and completeness of free inputs. Using FreeVer, we have detected 7 previously unknown *high impact* vulnerabilities in Polygon zkEVM.

Acknowledgments

We sincerely thank the shepherd and the reviewers for their invaluable insights and constructive comments. This work is supported by Hong Kong RGC Projects (PolyU15224121, PolyU15231223), and Hong Kong RGC Grant for Theme-based Research Scheme Project (T43-513/23-N).

9 Ethics Considerations

The paper introduces a novel dual execution path attack and proposes an automated formal verification tool to detect free input-related vulnerabilities in Polygon zkEVM. It aims to ① improve the community's knowledge about vulnerable implementations, and ② enhance detection techniques to better defend against attacks that leverage free inputs.

Potential Risk 1. Attackers may improve their knowledge by learning the new dual execution path attack. Therefore, we do not directly share any exploits.

Potential Risk 2. Attackers may use the detected vulnerabilities to launch attacks. Therefore, we have submitted all detected bugs to the Polygon team and confirmed that all of them have been correctly fixed.

10 Open Science

This research complies with the open science policy. The artifacts are open-sourced at https://doi.org/10.5281/zenodo.15609121, including the following:

- The source code of FreeVer, including the zkASM semantics in K and the Python code for other components.
- The two datasets used in our evaluation, along with the relevant scripts for evaluation.
- The documentation for setting up the execution environments for FreeVer and instructions for using FreeVer.

References

[1] Jsfuzz: coverage-guided fuzz testing for javascript. https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/jsfuzz, 2025.

[2] Seyed Mojtaba Hosseini Bamakan, Amirhossein Motavali, and Alireza Babaei Bondarti. A survey of blockchain consensus algorithms performance evaluation criteria. *Expert Systems with Applications*, 154:113385, 2020.

[3] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE TDSC*, 20(6):4733–4751, 2022.

[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[5] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. Sok: what don't we know? understanding security vulnerabilities in snarks. In *Proc. USENIX Security*, 2025.

[6] Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. Analyzing and benchmarking zk-rollups. *Cryptology ePrint Archive*, 2024.

[7] Xiaohong Chen and Grigore Roşu. A language-independent program verification framework. In *Proc. ISoLA*, 2018.

[8] Xiaohong Chen and Grigore Roşu. Matching μ-logic. In *Proc. LICS*, 2019.

[9] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proc. PLDI*, 2019.

[10] Ethereum. Ethereum documentation. https://ethereum.org/en/developers/docs/scaling/zk-rollups/, 2024.

[11] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[12] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. *Cryptology ePrint Archive*, 2015.

[13] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast zk processor with cached oram for ansi c programs. In *Proc. S&P*, 2021.

[14] Hexens. Security review report for polygon zkevm. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/Hexens_Polygon_zkEVM_PUBLIC_27.02.23.pdf, 2024.

[15] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *Proc. CSF*, 2018.

[16] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Scalable verification of zero-knowledge protocols. In *Proc. S&P*, 2024.

[17] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *Proc. S&P*, 2020.

[18] Matter Labs. zksync era. https://github.com/matter-labs/zksync-era/, 2024.

[19] Zihao Li, Xinghao Peng, Zheyuan He, Xiapu Luo, and Ting Chen. famulet: Finding finalization failure bugs in polygon zkrollup. In *Proc. CCS*, 2024.

[20] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. In *Proc. S&P*, 2024.

[21] Zuchao Ma, Muhui Jiang, Feng Luo, Xiapu Luo, and Yajin Zhou. Surviving in dark forest: Towards evading the attacks from front-running bots in application layer. In *Proc. USENIX Security*, 2025.

[22] Zuchao Ma, Muhui Jiang, Xiapu Luo, Haoyu Wang, and Yajin Zhou. Uncovering nft domain-specific defects on smart contract bytecode. *IEEE TDSC*, 2025.

[23] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages*, 7:1510–1532, 2023.

[24] Microsoft Research. The z3 theorem prover v4.13.3. https://github.com/Z3Prover/z3/tree/z3-4.13.3, 2024.

[25] Grigore Rosu. Matching logic. *Logical Methods in Computer Science*, 13, 2017.

[26] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proc. OOPSLA*, 2012.

[27] Grigore Rosu, Andrei Stefanescu, Stefan Ciobâca, and Brandon M Moore. One-path reachability logic. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2013.

[28] Spearbit. Polygon zkevm security review, december 2022 engagement. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/zkEVM-engagement-1-Spearbit-27-March.pdf, 2022.

[29] Spearbit. Polygon zkevm security review: Calldata bugfix review. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/zkEVM-ROM-upgrade-2-Spearbit-21-August.pdf, 2023.

[30] Spearbit. Polygon zkevm security review: January 2023 engagement. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/zkEVM-engagement-2-Spearbit-27-March.pdf, 2023.

[31] Spearbit. Polygon zkevm security review: March 2023 engagement. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/zkEVM-engagement-3-Spearbit-6-April.pdf, 2023.

[32] Spearbit. Polygon zkevm security review: zkevm rom jun upgrade features review. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/zkEVM-ROM-upgrade-1-Spearbit-30-May.pdf, 2023.

[33] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M Moore, Traian Florin Şerbănută, and Grigore Roşu. All-path reachability logic. In *International Conference on Rewriting Techniques and Applications*, 2014.

[34] StarkWare Team. ethstark documentation–version 1.1. Technical report, IACR preprint archive 2021, 2021.

[35] Scroll Tech. Scroll. https://github.com/scroll-tech, 2024.

[36] Polygon Technology. Polygon zkevm documentation. https://docs.polygon.technology/zkEVM/, 2024.

[37] Polygon Technology. Polygon zkevm repositries. https://github.com/0xpolygonhermez, 2024.

[38] Polygon Technology. zkasm compiler repository. https://github.com/0xPolygonHermez/zkasmcom, 2024.

[39] Polygon Technology. zkevm prover in c++ repository. https://github.com/0xPolygonHermez/zkevm-prover, 2024.

[40] Polygon Technology. zkevm proverjs repository. https://github.com/0xPolygonHermez/zkevm-proverjs, 2024.

[41] Polygon Technology. zkevm rom repository. https://github.com/0xPolygonHermez/zkevm-rom, 2024.

[42] Polygon Technology. zkprover documentation. https://docs.polygon.technology/zkEVM/architecture/zkprover, 2024.

[43] Verichains. Security audit of polygon zkevm. https://github.com/0xPolygonHermez/zkevm-rom/blob/main/audits/Polygon-zkEVM-Public-v1.1-verichains-19-03-2024.pdf, 2024.

[44] Runtime Verification. K framework v7.1.151. https://github.com/runtimeverification/k/tree/v7.1.151, 2024.

[45] Runtime Verification. K framework. https://github.com/kframework, 2025.

[46] Wikipedia. Backus–naur form. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form, 2024.

[47] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151(2014):1–32, 2014.

[48] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. Deepinfer: Deep type inference from smart contract bytecode. In *Proc. ESEC/FSE*, 2023.