

zkFUZZ: Foundation and Framework for Effective Fuzzing of Zero-Knowledge Circuits

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

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

Jihwan Kim
jk4908@columbia.edu
Columbia University
New York, USA

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

Abstract

Zero-knowledge (ZK) circuits enable privacy-preserving computations and are central to many cryptographic protocols. Systems like Circom simplify ZK development by combining witness computation and circuit constraints in one program. However, even small errors can compromise security of ZK programs—under-constrained circuits may accept invalid witnesses, while over-constrained ones may reject valid ones. Static analyzers are often imprecise with high false positives, and formal tools struggle with real-world circuit scale. Additionally, existing tools overlook several critical behaviors, such as intermediate computations and program aborts, and thus miss many vulnerabilities.

Our theoretical contribution is the Trace-Constraint Consistency Test (TCCT), a foundational language-independent formulation of ZK circuit bugs that defines bugs as discrepancies between the execution traces of the computation and the circuit constraints. TCCT captures both intermediate computations and program aborts, detecting bugs that elude prior tools.

Our systems contribution is zkFUZZ, a novel program mutation-based fuzzing framework for detecting TCCT violations. zkFUZZ systematically mutates the computational logic of Zk programs guided by a novel fitness function, and injects carefully crafted inputs using tailored heuristics to expose bugs. We evaluated zkFUZZ on 354 real-world ZK circuits written in Circom, a leading programming system for ZK development. zkFUZZ successfully identified 66 bugs, including 38 zero-days—18 of which were confirmed by developers and 6 fixed, earning bug bounties.

Keywords

Zero Knowledge Proof, Fuzzing, Circom

1 Introduction

Zero-knowledge (ZK) circuits have emerged as a foundational technology for privacy-preserving computation with succinct proofs and efficient verification [24, 33, 40]. Their adoption spans a variety of critical sectors, including anonymous cryptocurrencies [49], confidential smart contracts [55, 62], secure COVID-19 contact tracing protocols [39], privacy-preserving authentication [5], and verifiable machine learning systems [61, 65]. Industry forecasts predict that the ZK market will reach \$10 billion by 2030, with Web3 applications alone expected to execute nearly 90 billion proofs [45].

Manually constructing ZK circuits is complex and error-prone, so developers often use programming systems like Circom [7], which allow them to express both the *computation* for deriving the secret witness from the input signals and the *constraints* of the finite-field arithmetic circuit (used to generate and verify proofs later) within one ZK program written in a high-level, domain-specific language (DSL). The DSL compiler then translates the program into (1) an executable (e.g., in WASM) for witness generation and (2) a set of constraints that define the circuit and support proof verification.

Even with high-level programming systems, developing correct circuits remains notoriously difficult due to two core challenges. First, the computation and the constraints must align exactly, yet they operate in different domains: general-purpose logic vs. polynomial constraints optimized for efficiency. Developers must often manually translate logic into polynomial form, and any mismatch results in incorrect or unverifiable proofs [59]. Second, the non-intuitive behavior of modular arithmetic in finite fields often confuses developers. For example, function `iszero(x)` should return 1 if $x = 0$ and 0 otherwise—but in a field of prime order 5, expression `iszero(2 + 3)` evaluates to 1, since $2 + 3 \equiv 0 \pmod{5}$. Such surprises often trip up developers unfamiliar with finite fields.

Unsurprisingly, these challenges lead to vulnerabilities: *under-constrained* circuits that allow false proofs, and *over-constrained* circuits that reject valid witnesses. Both can result in serious security breaches [12, 36]. For instance, an under-constrained bug in zkSync—a widely used zk-Rollup—allowed a malicious prover to extract \$1.9 billion worth of funds at the time of disclosure [11, 56].

Existing methods for automatically detecting vulnerabilities in ZK programs face several limitations. Static analysis [42, 59] is primarily pattern-based and fails to capture deep semantic inconsistencies, suffering from high false positives, which waste developer time and lose their trust [9]. Formal methods [13, 31, 32, 38, 43] improve precision, but their reliance on SMT solvers limits scalability to real-world circuits with thousands of constraints. (Existing dynamic methods [19, 27, 34, 60] target compilation, proving, or verification systems, and do not extend to individual ZK programs.)

Furthermore, current formal definitions of ZK vulnerabilities are incomplete, leaving numerous bugs undetected. In particular, they ignore or mishandle intermediate computations [13, 32, 43], missing an entire class of over-constrained bugs. They also fail to account for abnormal termination in computation—bugs that allow inputs to crash witness generation while still satisfying circuit

constraints, despite such cases accounting for nearly 50% of real-world ZK bugs in our evaluation.

Our first contribution in this paper is a new theoretical formulation, the Trace-Constraint Consistency Test (TCCT), that models the vulnerabilities in ZK programs as discrepancies between (1) all possible execution traces that a computation may produce and (2) the set of input, intermediate, and output values permitted by the circuit constraints. It captures both under-constrained, or *soundness*, vulnerabilities—when an invalid trace is allowed by the circuit constraints—and over-constrained, or *completeness*, vulnerabilities when a valid trace is incorrectly rejected. TCCT correctly accounts for intermediate computations, and is the first comprehensive model to define the semantics of over-constrained ZK vulnerabilities. It is also the first, to the best of our knowledge, to handle abnormal termination in witness computation. Our formulation is general and language-agnostic, laying a solid foundation for tools that detect both soundness and completeness ZK vulnerabilities.

Our second contribution is the design and implementation of zkFUZZ, a novel dynamic analysis framework that leverages fuzzing with program mutations to detect TCCT violations in ZK programs. To uncover over-constrained vulnerabilities, zkFUZZ systematically fuzzes inputs to the witness computation in search of valid execution traces incorrectly rejected by the circuit constraints. To uncover under-constrained vulnerabilities, it mutates the witness computation and searches for inputs that produce different outputs accepted by the circuit constraints. Since these outputs are produced by a mutated computation that semantically differs from the original, they are, by construction, invalid and should not be accepted. zkFUZZ is fully automated, and for every bug detected, it provides a concrete counterexample, greatly simplifying diagnosis.

A key challenge for zkFUZZ is the vast search space. Although ZK program inputs are finite, they belong to large prime fields, and behavior in small fields does not generalize to larger fields (e.g., $\text{iszero}(2 + 3)$ is true in \mathbb{F}_5 but false in \mathbb{F}_7). Blindly searching these large fields with many input signals is computationally expensive. The program mutation space is even larger—practically infinite—since arbitrary programs may produce outputs accepted by buggy circuits. Traditional heuristics like coverage-guided fuzzing are ineffective, as zkFUZZ seeks specific traces that expose vulnerabilities, not broad code coverage.

To search this vast space efficiently, zkFUZZ adopts a joint input and program mutation-based evolutionary fuzzing algorithm. Given a witness computation program P and circuit constraints C , zkFUZZ generates multiple mutants of P and heuristically samples multiple input values likely to lead to bugs. Unlike traditional fuzzing, which typically mutates only the inputs, zkFUZZ mutates both the program and inputs. It executes all program mutants on all sampled inputs and checks the resulting traces against C for TCCT violations.

In each iteration, zkFUZZ scores program mutants using a novel min-sum fitness function that estimates the likelihood of triggering a violation. For each input, it sums constraint errors across all constraints and assigns the mutant the minimum such sum across inputs. Unlike prior sum-only methods [14], this approach is better suited for joint program and input fuzzing. zkFUZZ then performs crossover, favoring lower-scoring mutants, and evaluates the resulting offspring on newly sampled inputs. This process repeats until a violation is found or a timeout occurs. Each iteration is

independent once crossover mutants are generated, and zkFUZZ further improves efficiency via heuristics that prioritize mutation of program statements and input regions most likely to cause bugs.

In our evaluation, we implement zkFUZZ for Circom [7], the most popular programming system for developing ZK circuits [59], and compare it with four state-of-the-art detection tools on 354 real-world ZK circuits. Our results show that zkFUZZ consistently outperforms all other methods, finding 30% to 300% more bugs, without any false positives. In total, zkFUZZ found 66 bugs, including 38 previously unknown zero-days—18 of which were confirmed by developers and 6 fixed. For example, 11 under-constrained bugs in *zk-regex* [66], a popular ZK regex verification tool, were confirmed and awarded bug bounties. These bugs allow malicious provers to generate bogus proofs claiming that they possess a string with certain properties (e.g., a valid email address) without actually having such a string. Another confirmed bug in *passport-zk-circuits* [46], a ZK-based biometric passport project, allowed attackers to forge proofs of possessing data matching a required encoding. Our code is publicly available at <https://github.com/Koukyosyumei/zkFuzz>.

In summary, this paper makes the following key contributions:

- Introduce TCCT, a language-independent formal framework that captures both under- and over-constrained ZK bugs through trace-constraint inconsistencies.
- Develop zkFUZZ, a dynamic analysis tool that detects TCCT violations by jointly mutating programs and inputs.
- Propose a joint evolutionary fuzzing algorithm with a novel min-sum fitness function and guided crossover.
- Evaluate zkFUZZ on 354 real-world ZK circuits, finding 30–300% more bugs than prior tools with zero false positives.

2 Background

This section provides an overview of ZK Proof systems (§ 2.1), vulnerabilities in ZK circuits (§ 2.2), and Circom, the most popular programming system for ZK circuits [8, 59] (§ 2.3).

2.1 ZK Proof Systems

ZK proofs enable a *prover* to convince a *verifier* of the validity of a statement (e.g., knowledge of a secret value) without revealing any information beyond the statement’s truth [35]. Formally, ZK proof systems satisfy three properties:

- **Completeness:** If the statement is true, an honest prover convinces an honest verifier.
- **Soundness:** A malicious prover cannot convince the verifier of a false statement.
- **Zero-Knowledge:** The proof reveals nothing about the secret input beyond the statement’s validity.

These properties enable the construction of privacy-preserving verifiable systems, such as anonymous cryptocurrencies (e.g., Zcash’s zk-SNARKs), confidential smart contracts, and verifiable voting protocols. Modern applications extend to blockchain scaling (e.g., ZK-Rollups) and verification of machine-learning integrity [1, 21, 33].

At a high level, a ZK system operates on arithmetic circuits that perform computations over a finite field, where all variables and operations (e.g., addition, multiplication) are defined modulo a large prime. It exposes two primitives: Prove and Verify. The ZK

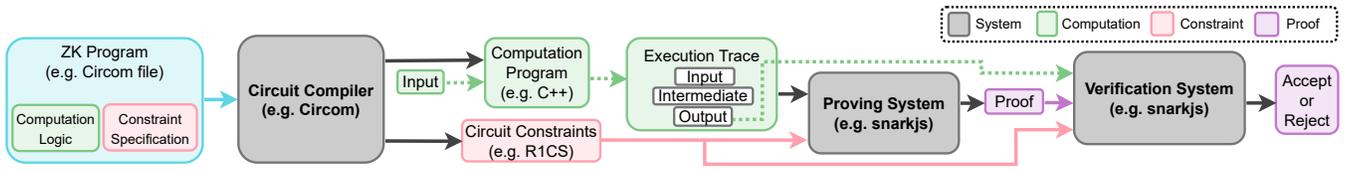


Figure 1: Overview of ZK proof systems. The circuit compiler processes a ZK program into a witness computation program and circuit constraints. Prover executes the witness program to obtain a trace (witness and public values) and creates a proof using the proving system. The verifier validates the proof using the verification system with the public output, the constraint, and the proof. Our fuzzer checks the inconsistencies between the computation logic and the circuit constraints in the ZK program.

proof process begins with the prover running `Prove` to generate a proof: $\pi \leftarrow \text{Prove}(C, w)$ where C is the circuit and w the prover’s secret value, known as the *witness*. Importantly, the proof π is constructed to reveal no information about w . Once π is generated, the prover sends it to the verifier who then calls `Verify`(C, π) which returns either *accept* or *reject*. For performance reasons, it is often desirable for the proof to be succinct and non-interactive.

ZK Programming Infrastructure. ZK protocols assume a static secret w but, in practice, developers frequently want to make claims about computations, such as proving membership in a Merkle tree for off-chain transactions—where the witness w includes not just a secret value, but also a computation trace (e.g., a Merkle path) that must satisfy the circuit constraints. For efficient proof generation and verification, the constraints are often expressed in quadratic form like in R1CS (Rank-1 Constraint System). Therefore, general computations do not directly map to the circuit constraints.

Manually constructing circuits for such computations and determining which values belong in the witness is complex and error-prone, so developers often use ZK programming systems [33] like Circom, which allow them to express the witness computation and the circuit constraints in a single ZK program written in a high-level DSL. The DSL compiler then translates the program into a computation program \mathcal{P} and the corresponding set of circuit constraints C . When \mathcal{P} executes, it produces a full execution trace, including not only the input x and output y , but also important intermediate values z , to match the required constraint format. For example, computing $y = x^4$ typically requires two quadratic constraints: $z = x^2$ and $y = z^2$. The resulting execution trace (x, z, y) becomes the witness (typically, y is public, while x and z are secret; however, developers sometimes make x public as well, depending on the application and desired level of privacy.) Fig. 1 illustrates this process, along with the subsequent use of proving and verification systems like snarkjs [28].

Although some systems like Noir [4] support automatic generation of constraints from the computations, the resulting constraints are often significantly less efficient, increasing the number of constraints by 3x to 300x [41]. In the remainder of our paper, we focus on practical ZK systems that produce efficient constraints.

2.2 Overview of Vulnerabilities in ZK Programs

Even with high-level programming systems, general computations often involve non-quadratic operations that cannot be directly mapped to the required constraint format. As a result, it is the developer’s responsibility to ensure that the intended computation

\mathcal{P} and the corresponding circuit constraints C are aligned; any mismatch may lead to incorrect or unverifiable proofs. Unfamiliarity with finite-field arithmetic further complicates matters, as its modular behavior introduces unexpected corner cases. Consequently, ZK programs are notoriously difficult to get right [13, 43] and may contain the following two classes of vulnerabilities.

Under-Constrained Circuits. C is too loose for \mathcal{P} , allowing malicious provers to convince verifiers that “I know the input whose corresponding output is y ,” even when they do not. Such vulnerabilities violate the soundness property.

Over-Constrained Circuit. C is too strict for \mathcal{P} , preventing honest provers from generating valid proofs for some correct traces. Such vulnerabilities violate the completeness property.

Code 1 and 2 show the computation and constraints, respectively, for an under-constrained vulnerability caught by zkFuzz in a real-world 1-bit right-shift program from the *circom-monolith* library [53]. Since right shift is not a quadratic operation and cannot be directly encoded as a constraint, the developer introduces an intermediate value b , defined as the difference between x and $2y$, where $y = x \gg 1$. Unfortunately, due to quirks in finite-field arithmetic, multiple assignments to (y, b) can satisfy the constraints for a given input x . For example, in the finite field \mathbb{F}_{11} , if $x = 7$, the expected output is $y = 3$ and $b = 1$. Yet, $y = 9, b = 0$ also satisfy the constraints, because $7 - 9 \cdot 2 = -11 \equiv 0 \pmod{11}$. The circuit is thus under-constrained, allowing an invalid proof.

In contrast, the constraints in Code 3 are too strict. They enforce $b = 0$, thereby rejecting valid traces such as $\{x: 3, y: 1, b: 1\}$, and preventing an honest prover from generating a valid proof.

The safe implementation of 1-bit right shift using the bit array can be found in Appendix B.

```

1 fn RShift1(x) {
2   y = x >> 1
3   b = x - y * 2
4   assert(b*(1-b) == 0)
5   return y
6 }

```

Code 1: 1-bit R shift

```

1 (b = x-y*2) && (b*(1-b) = 0)

```

Code 2: Under-constrained circuit.

```

1 (b = x - y*2) && (b = 0)

```

Code 3: Over-constrained circuit.

Out-of-Scope Vulnerabilities. zkFuzz focuses on soundness and completeness vulnerabilities in ZK proofs that arise from developer mistakes in tying constraints to computation. The following types of vulnerabilities are out of scope: (1) vulnerabilities in the underlying compilation, proving, and verification systems (e.g., the Frozen Heart vulnerability caused by an insecure implementation

of the Fiat-Shamir transformation [56]); (2) bugs solely in the computation logic, such as unused inputs in hashing [59] (such unused inputs may be acceptable depending on the use case, as in computing the trace of a matrix where only diagonal elements are used); and (3) bugs solely in ZK protocol design that leak information and violate the zero-knowledge property [12], inline with all prior detectors [13, 32, 42, 43, 59]—what constitutes a secret is highly application-dependent: e.g., in Validity Rollups [15], ZK proofs are used to accelerate transactions rather than to conceal them.

2.3 Circom Primer

While our vulnerability definition and zkFuzz techniques apply to different ZK programming systems, this paper focuses on Circom, the most popular DSL and infrastructure for developing ZK circuits.

Value Domains and Operators. In Circom, all computations are performed over a finite field \mathbb{F}_q , where q is a large prime number. The language supports standard arithmetic operators such as addition (+), subtraction (−), multiplication (*), and division (/), all performed modulo q . Additionally, Circom provides integer division (\), bitwise, and comparison operators for defining complex circuit logic. The operands or results of these operators often need implicit conversion between integer and finite fields.

Signals and Variables. Circom enforces a strict separation between constrained and unconstrained computation through two data types. *Signals* are the primary data type and define the flow of data and the constraints that must be satisfied. They are immutable once defined and can serve as input, output, or intermediate values. In contrast, *variables*, declared using the `var` keyword, are unconstrained local values used for the computations but are not tracked by the constraint system. Unlike signals, variables are mutable and can be reassigned during execution.

Constraint and Assignment Semantics. Circom constraints follow a quadratic form: the product of two linear expressions equal a third linear expression. To express such constraints alongside computations, Circom introduces the following operators.

Weak Assignment (`<--`) assigns a value to a signal without generating a constraint, usually used for intermediate computations where constraints are unnecessary.

Strong Assignment (`<==`) assigns a value to a signal and generates a constraint, ensuring the assigned value is enforced by the circuit.

Assert (`assert`) inserts a runtime assertion into the computation without adding a constraint.

Equality Constraint (`==`) asserts equality between two signals and generates a constraint. If the `constraint_assert_disabled` flag is set, the assertion is not inserted into the computation.

Variable Assignment (`=`) assigns a value to a variable, and does not add a constraint to the circuit.

Other than `==`, Circom operators do not allow adding constraints without performing the corresponding computation, so `===` with `constraint_assert_disabled` set is the only way to introduce over-constrained bugs. Nonetheless, over-constrained circuits can arise in other languages such as halo2 [54, 63], making it important to define them formally.

Templates and Components. Circom *templates* are parameterized circuit blueprints that can be instantiated as *components* by other circuits. The parameters can be, for example, the dimensions

of input arrays, making them highly flexible. Templates and components enable modular design, but also introduce the risk that bugs in one template can propagate across all circuits that instantiate it, potentially affecting many downstream ZK programs.

3 Definitions of ZK Program Vulnerabilities

This section formulates the typical ZK circuit’s vulnerabilities as the **Trace-Constraint Consistency Test (TCCT)**, a unified model that rigorously formulates bugs within ZK circuits as the inconsistency between the computation logic and its associated circuit constraint.

3.1 Incompleteness of Prior Definitions

Prior formal definitions of ZK program vulnerabilities are incomplete on two fronts. First, when detecting under-constrained vulnerabilities, they focus on identifying nondeterministic circuit constraints [32, 43], while ignoring computation aborts. If the circuit constraints C allow the same input x to map to an output y' different from the computed output y , the circuit is clearly under-constrained. However, even if C is deterministic—accepting only one value of y for a given x —the circuit can still be under-constrained if the computation \mathcal{P} aborts on x (e.g., due to a runtime assertion). Attackers can exploit such aborts by crafting inputs that cause the computation to fail while still satisfying the circuit constraints, thereby producing bogus proofs for traces that never actually occurred. Our evaluation shows that such abort vulnerabilities account for nearly 50% of the real-world bugs zkFuzz identified.

Second, when detecting over-constrained vulnerabilities, prior work [13] considers only a special case where C permits an empty set of traces. A complete definition must consider the full set of execution traces produced by \mathcal{P} and those accepted by C . In particular, it must account for intermediate values. Suppose \mathcal{P} produces a trace (x, z, y) for input x , but C accepts only (x, z', y) where $z \neq z'$. In this case, C is over-constrained with respect to \mathcal{P} , as an honest prover with the valid trace (x, z, y) would be unable to generate a valid proof. If, however, C accepts both (x, z, y) and (x, z', y) (a common optimization in practice), there is no violation, as the prover can produce a valid proof.

3.2 Trace-Constraint Consistency Test

Let \mathbb{F}_q be a finite field of order q , where q is a prime number.

Definition 3.1 (ZK Program). A ZK program is defined as a pair of a computation and a set of circuit constraints (\mathcal{P}, C) , where:

$\mathcal{P} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^k \times (\mathbb{F}_q^m \cup \{\perp\})$ is a *computation* that takes as input a tuple $(x_1, x_2, \dots, x_n) \in \mathbb{F}_q^n$, and behaves as follows:

- On successful termination, outputs a pair (z, y) where $z = (z_1, \dots, z_k) \in \mathbb{F}_q^k$ represents intermediate values necessary to construct a witness and $y = (y_1, \dots, y_m) \in \mathbb{F}_q^m$ the final outputs. Each value is assigned exactly once.
- On abnormal termination, it returns the special value $(_, \perp)$.

$C : \mathbb{F}_q^n \times \mathbb{F}_q^k \times \mathbb{F}_q^m \rightarrow \{\text{true}, \text{false}\}$ is a set of circuit *constraints*, which evaluates to true if the given input, intermediate, and output values satisfy the required conditions; and false otherwise.

Definition 3.2 (Execution Trace). For an input $x \in \mathbb{F}_q^n$ and computation \mathcal{P} , if $\mathcal{P}(x) = (z, y)$ where $z \in \mathbb{F}_q^k$ and $y \in \mathbb{F}_q^m$, then the triplet (x, z, y) is called an *execution trace* of \mathcal{P} .

Definition 3.3 (Trace Set). For a computation \mathcal{P} , the trace set $\mathcal{T}(\mathcal{P})$ is defined as:

$$\mathcal{T}(\mathcal{P}) := \{(x, z, y) \mid x \in \mathbb{F}_q^n, z \in \mathbb{F}_q^k, y \in \mathbb{F}_q^m, \mathcal{P}(x) = (z, y)\}$$

Definition 3.4 (Constraint Satisfaction Set). For circuit constraints C , the constraint satisfaction set $\mathcal{S}(C)$ is defined as:

The trace set $\mathcal{T}(\mathcal{P})$ is the set of all possible traces, except aborts, generated by the computation \mathcal{P} while the constraint satisfaction set $\mathcal{S}(C)$ is the set of all tuples satisfying the circuit constraints C .

$$\mathcal{S}(C) := \{(x, z, y) \mid x \in \mathbb{F}_q^n, z \in \mathbb{F}_q^k, y \in \mathbb{F}_q^m, C(x, z, y) = \text{true}\}$$

In addition, we introduce an operator that projects a set of tuples of input, intermediate, and output values to a set of pairs of input and output values.

Definition 3.5 (Projection). Let $\{(x_1, z_1, y_1), \dots\} \subseteq 2^{\mathbb{F}_q^n \times \mathbb{F}_q^k \times \mathbb{F}_q^m}$ be a set of tuples of inputs, intermediates and outputs. Then, we define the projection operator Π_{xy} as $\Pi_{xy}(\{(x_1, z_1, y_1), (x_2, z_2, y_2), \dots\}) := \{(x_1, y_1), (x_2, y_2), \dots\}$.

Given the above building blocks, we formally define under-constrained and over-constrained circuits.

Definition 3.6 (Under-Constrained Circuit). We say that C is *under-constrained* for \mathcal{P} if

$$\Pi_{xy}(\mathcal{S}(C)) \setminus \Pi_{xy}(\mathcal{T}(\mathcal{P})) \neq \emptyset \quad (1)$$

Intuitively, C is under-constrained for \mathcal{P} if C accepts $(x, _ , y)$ but $\mathcal{P}(x) = (_ , y')$ and $y \neq y'$ or $\mathcal{P}(x)$ aborts. Projection out the intermediate values here is necessary for the following reason. In contrast to over-constrained vulnerabilities, if $\mathcal{P}(x) = (z, y)$ and C accepts (x, z', y) such that $z \neq z'$, it is not an under-constrained vulnerability, since malicious provers still cannot create bogus proofs asserting that $\mathcal{P}(x)$ yields an output $y' \neq y$. Similarly, even if C accepts both (x, z, y) and (x, z', y) , adversaries cannot forge proofs with outputs differing from y .

This definition naturally takes care of abnormal termination of the computation. Any input x that causes \mathcal{P} to abort will not appear in the trace set $\mathcal{T}(\mathcal{P})$. If $\mathcal{S}(C)$ includes any tuple of the form $(x, _ , _)$, then the circuit is under-constrained, as it accepts an input that the computation cannot successfully process.

Definition 3.7 (Over-Constrained Circuit). We say that C is *over-constrained* for \mathcal{P} if

$$\mathcal{T}(\mathcal{P}) \setminus \mathcal{S}(C) \neq \emptyset \quad (2)$$

Intuitively, C is over-constrained w.r.t. \mathcal{P} if any valid trace of \mathcal{P} is not accepted by C . For example, if $\mathcal{P}(x)$ yields output y , but C accepts only $y' \neq y$ for the same input x , then C is over-constrained.

This definition must include intermediate values z , as they are essential for satisfying C . Given $\mathcal{P}(x) = (z, y)$, consider the two cases discussed in the previous subsection: (1) C accepts only (x, z', y) with $z' \neq z$. Since $\mathcal{T}(\mathcal{P}) \setminus \mathcal{S}(C) \neq \emptyset$, C is over-constrained. (2) C accepts both (x, z, y) and (x, z', y) . Since execution trace (x, z, y) occurs in $\mathcal{S}(C)$, this is not an over-constrained vulnerability, even though multiple traces share the same input x and output y . In fact,

allowing a broader range of intermediate values in C is a common optimization in practical ZK circuit design.

We now combine Eq. 1 and Eq. 2 and define a test for the consistency between the trace and the constraint satisfaction sets:

Definition 3.8 (Trace-Constraint Consistency Test). For computation \mathcal{P} and constraints C , the *Trace-Constraint Consistency Test* (TCCT) ascertains the following:

$$\Pi_{xy}(\mathcal{S}(C)) \setminus \Pi_{xy}(\mathcal{T}(\mathcal{P})) = \emptyset \wedge \mathcal{T}(\mathcal{P}) \setminus \mathcal{S}(C) = \emptyset \quad (3)$$

If Eq. 3 holds, we say that (\mathcal{P}, C) is *well-constrained*. This condition is equivalent to $\mathcal{T}(\mathcal{P}) \subseteq \mathcal{S}(C) \wedge \Pi_{xy}(\mathcal{T}(\mathcal{P})) = \Pi_{xy}(\mathcal{S}(C))$, meaning that all execution traces satisfy the constraints, and the set of input-output pairs is identical between the execution traces and the constraint satisfaction set.

The computational complexity of TCCT is as follows:

- THEOREM 3.9 (COMPLEXITY).** *Let (\mathcal{P}, C) denote a TCCT instance.*
- Determining whether C is under-constrained for \mathcal{P} is NP-complete.*
 - Determining whether C is over-constrained for \mathcal{P} is NP-complete.*
 - The Trace-Constraint Consistency Test is co-NP-complete.*

Proof is based on the reduction from the Boolean Satisfiability problem (SAT) and the Boolean tautology problem and can be found in Appendix A.

Comparison to Prior Definitions. Tab. 1 compares our model, TCCT, with prior definitions. For under-constrained bugs, prior work detect only non-deterministic behaviors [32, 43], and fail to capture those caused by computation aborts. For over-constrained bugs, prior work considers only a special case where the constraint satisfaction set is empty [13]. In contrast, TCCT is more general and complete, capturing all bugs detectable by existing models, as well as additional bugs that prior definitions overlook. A key advantage of TCCT is its DSL-independence: even though different DSLs may offer varied primitives or syntactical sugar for specifying constraints and computations in ZK circuits, TCCT's definitions consistently apply across all of them.

Table 1: Comparison of formal definitions of ZK vulnerabilities. Our TCCT is more general than existing models.

Model	Under-Constrained	Over-Constrained
[32, 43]	$\exists x, z, z', y, y'. y \neq y' \wedge C(x, z, y) \wedge C(x, z', y')$	-
[13]	Same as [32, 43]	$\mathcal{S}(C) = \emptyset$
TCCT	$\Pi_{xy}(\mathcal{S}(C)) \setminus \Pi_{xy}(\mathcal{T}(\mathcal{P})) \neq \emptyset$	$\mathcal{T}(\mathcal{P}) \setminus \mathcal{S}(C) \neq \emptyset$

3.3 Vulnerable and Benign Examples

This subsection shows two real-world Circom circuits, an under-constrained circuit that illustrates our definition, particularly the role of computation aborts; and a well-constrained circuit that highlight the importance of accounting for intermediate values.

Code 4 shows a real-world under-constrained circuit (slightly modified for clarity) caught by zkFuzz from *circom-zkVerify* [3]. It

```

1 template Verify() {
2   signal input x1;
3   signal input x2;
4
5   component h = Hash();
6   h.x <== x1;
7   assert(x2==h.y);
8 }

```

Code 4: Under-constrained circuit from [3] undetectable by existing formal definitions. Only our TCCT can capture it, and zkFuzz successfully identifies it.

```

1 template IsZero() {
2   signal input x;
3   signal output y;
4   signal z;
5
6   z <-- x != 0 ? 1/x : 0;
7   y <== -x*z + 1;
8   x*y == 0;
9 }

```

Code 5: Benign circuit from [29] incorrectly flagged by prior static tools. Its trace and constraint satisfaction sets are shown in Tab. 2.

had previously gone undetected until our discovery, later confirmed and fixed by the developers. The code instantiates a Hash template as component h (line 5), computes the hash value of input x_1 (line 6), and checks whether it matches x_2 using `assert` (line 7). Circom’s `assert`, however, performs runtime checks only without adding any constraint. As a result, the computation aborts when x_1 does not hash to x_2 , the constraints remain satisfiable for any combination of x_1 and x_2 . No prior formal tools can detect this bug because the circuit has no output and is thus always deterministic. While a static tool [59] does flag this code, it relies on pattern matching and produces many false positives. The fact that this bug went unfixed indicates that suggests that developers who may have run the tool did not inspect all of its warnings. To fix this circuit, `assert` should be replaced with `==`, which explicitly encodes the equality check into the constraints.

Code 5 shows a benign circuit template from Circom’s de facto standard library, *circom-lib* [29], which returns one if the input x is zero and zero otherwise. Line 6 introduces an intermediate value z using a weak assignment, which does not add any constraint since it is not a quadratic expression. Line 7 assigns the output y and introduces a constraint relating x , z , and y . Line 8 adds both a runtime assertion and an equality constraint. The full execution trace and constraint satisfaction sets are shown in Tab. 2. Notably, $\mathcal{T}(\mathcal{P})$ and $\mathcal{S}(C)$ are not identical: $\mathcal{S}(C)$ includes two additional tuples. However, TCCT correctly handles this case. After projecting out intermediate values, the sets of (x, y) pairs are identical, indicating no under-constrained bug. While $\mathcal{S}(C)$ includes three different intermediate values, $z = 0, 1, 2$, for the case where $x = 0$ and $y = 1$, it includes the valid tuple $(0, 0, 1)$ from $\mathcal{T}(\mathcal{P})$, so there is no over-constrained bug either. Despite being benign, this circuit is incorrectly flagged by prior static analyzers, which rely on pattern matching rather than a precise semantic definition of ZK bugs.

Table 2: Trace and Constraint Satisfaction Sets of IsZero in Code 5 ($q = 3$)

$\mathcal{T}(\mathcal{P})$			$\mathcal{S}(C)$		
x	z	y	x	z	y
0	0	1	0	0	1
			0	1	1
			0	2	1
1	1	0	1	1	0
2	2	0	2	2	0

More examples, including under-constrained circuits caused by insecure use of the LessThan template, a bug that only zkFuzz can detect; and over-constrained circuits can be found in Appendix B.

4 zkFuzz Design

This section introduces our novel fuzzing framework, zkFuzz, which detects bugs in ZK circuits by jointly optimizing program mutation and input generation, guided by target selectors. Fig. 2 provides an overview of the workflow. The design is DSL-agnostic, with an implementation for Circom described in § 5.

4.1 Joint Program Mutation & Input Generation

zkFuzz uses both program mutation and input fuzzing for detecting vulnerabilities. Although program mutation is widely used in mutation testing to assess test case effectiveness [44], our objective differs: we leverage mutation specifically to generate alternative traces that continue to satisfy the constraint, exposing undesired behaviors in ZK circuits.

Algo. 1 describes the key steps that examine both \mathcal{P} and its associated circuit constraints C by using two testing modes:

Detection of Over-Constrained Circuits. We first generate input data and execute the computation \mathcal{P} . If it completes without crashing, yet the resulting trace fails to satisfy the constraints C , this circuit is *over-constrained*.

Detection of Under-Constrained Circuits. To uncover under-constrained circuits, we construct a mutated version of the computation program, denoted by \mathcal{P}' . For example, the mutation algorithm can randomly substitute one operator for another. If \mathcal{P}' produces a non-crash trace whose output differs from \mathcal{P} while still meeting the circuit constraint, then the constraint is insufficient to enforce correctness—indicating an *under-constrained* circuit. Note that if the original computation \mathcal{P} crashes on the input while the mutated computation \mathcal{P}' successfully generates an execution trace that satisfies the constraint, the original output y is \perp . Since $\perp \neq y'$, this case is naturally captured by zkFuzz.

Algorithm 1 Fuzzing with Program Mutation to solve TCCT

```

1: for  $i \leftarrow 1, 2, \dots, \text{MAX\_GENERATION}$  do
2:   Generate input data  $x$ .
3:   Mutate  $\mathcal{P}$  to  $\mathcal{P}'$ 
4:   Execute both  $\mathcal{P}$  and  $\mathcal{P}'$  on  $x$ :
5:      $\mathcal{P}(x) = (z, y)$ ,  $\mathcal{P}'(x) = (z', y')$ .
6:   if  $y \neq \perp$  and  $C(x, z, y) = \text{false}$  then
7:     Report "Over-Constrained Problem."
8:   if  $y' \neq \perp$  and  $y \neq y'$  and  $C(x, z', y') = \text{true}$  then
9:     Report "Under-Constrained Problem."

```

For simplicity, Algo. 1 shows one input and one program mutant sampled per iteration. In practice, we set the number of input samples and program mutants to 30. Both input sampling and program mutation can be biased toward targets that are more likely to expose vulnerabilities. The algorithm also requires minimal bookkeeping across iterations: once program mutants are generated, the current iteration operates independently of the previous ones.

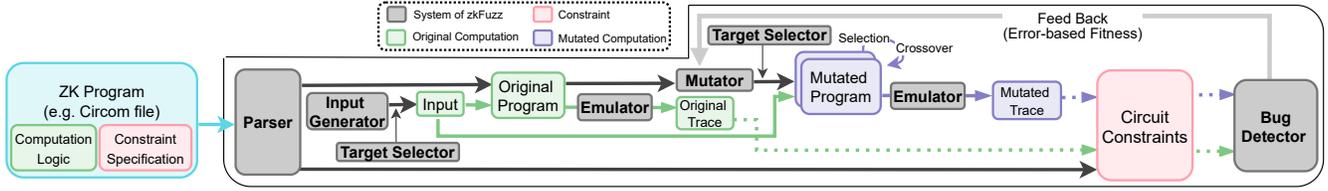


Figure 2: Basic workflow of zkFuzz. This fuzzing framework systematically mutates program logic and feeds artificially generated input data to the original and mutated programs to catch inconsistencies between the program and the constraint. The error-based fitness function is utilized to steer the selection and crossover of mutants. Several target selectors are also applied for input generation and program mutation to guide the search.

4.2 Guided Search

Since the search space of the program mutation and input generation is huge, we adopt a genetic algorithm with a novel fitness score and target selectors to achieve scalable bug detection.

Genetic Algorithm with Error-based Fitness Score. Genetic algorithm (GA) is a search heuristic commonly used in fuzzing, where candidate solutions are mutated over successive generations, with higher-performing candidates prioritized to guide the search.

zkFUZZ uses the smoothed error function transformed from the constraints C as the fitness score to find an execution trace that satisfies the constraints while producing a different output from the original. This means that we guide the mutation towards under-constrained circuits, and we adopt this design since under-constrained circuits tend to cause more critical outcomes than over-constrained circuits [13].

The constraints C usually consist of a set of polynomial equalities, $\bigwedge_{i=1}^{|C|} (a_i == b_i)$, where a_i and b_i are quadratic expressions, and $|C|$ denotes the number of those equations. Following [14], we convert each equality into an error term $|a_i - b_i|$, which evaluates to 0 when the constraint is satisfied and a positive value when violated. The total error for a given trace is defined as the sum of these errors if the trace produces a different output from the original. Otherwise, the error is set to ∞ , reflecting the failure to find a diverging execution. The final fitness score of a mutated program is determined by evaluating its error over multiple inputs and selecting the minimum score. Our *min-sum* method differs from prior max-only method [14] because zkFUZZ jointly mutates both programs and inputs.

In each iteration, zkFUZZ uses roulette-wheel selection [47], a widely adopted method in genetic algorithms, to generate new program mutants or perform crossover between mutants from the previous iteration. The selection is biased toward mutants with lower scores, as lower values indicate higher fitness. zkFUZZ represents a program mutant as a map from candidate mutation locations to the actual mutations applied, so crossover is implemented as a straightforward merging of the maps from the two parent mutants.

zkFUZZ can also optionally guide input generation via the above error-based fitness score, where the final fitness score of an input is defined as the minimum score over multiple program mutations. **Target Selectors.** We further improve the search of zkFUZZ by incorporating target selectors, which guide both input generation and program mutation. Unlike traditional targeted fuzzers—which primarily detect issues like memory access violations by finding

inputs that reach specific code blocks [58]—our target selectors are designed to maximize the odds of uncovering discrepancies between the computation program \mathcal{P} and the circuit constraints C .

One of the key target selectors is *skewed distribution*, which generates input data randomly while biasing the distribution to regions more likely to contain vulnerabilities. Specifically, empirical analysis reveals that many bugs stem from edge-case inputs, particularly values such as 0 and the prime modulus q . To effectively target these scenarios, our tool samples inputs and substitutes constants in unconstrained assignments using a skewed distribution that increases the likelihood of uncovering hidden vulnerabilities. Specifically, values are sampled from predefined ranges with the following probabilities: binary values (0 and 1) with 15%, small positive integers (2 to 10) with 34%, large values near the field order ($q - 100$ to $q - 1$) with 50%, and all other values (11 to $q - 101$) with 1%. By biasing input selection toward these critical edge cases, our approach enhances the detection of subtle bugs that might otherwise go unnoticed. The impact of the choice of skewed distribution can be found in § 6.4.

In addition, our zkFUZZ leverages static analysis to mutate computation programs and generate inputs more likely to trigger unexpected behaviors. For example, one of the main root causes of real-world under-constrained circuits is the *zero-division* pattern, wherein the division’s numerator and denominator may be zero [59]. In this scenario, a naive constraint for computing $y = x_1/x_2$ might be written as $y \cdot x_2 = x_1$. However, if both x_1 and x_2 are zero, any value of y will satisfy this constraint, leaving the circuit under-constrained. Therefore, when this pattern is detected, increasing the chance that generated values for x_1 and x_2 are zero can help reveal these vulnerabilities. Details of these static-analysis-based target selectors are provided in § 5.2.

5 zkFUZZ Implementation

We implemented zkFUZZ for Circom in about 4,500 lines of Rust. It begins with an enhanced parser that decomposes Circom programs into an Abstract Syntax Tree (AST) for the computation \mathcal{P} and corresponding circuit constraints C . A mutator generates program variants (\mathcal{P}'), and an emulator produces execution traces for these variants using artificial inputs. A bug detector checks if these traces satisfy the original circuit constraints based on TCCT. Target selectors strategically guide input generation and program mutation towards likely bugs. Finally, mutants’ fitness scores drive selection and crossover in subsequent generations (Fig. 2).

5.1 Circom Program Mutation

zkFuzz mutates Circom programs using the following methods.

(a) *Assertion Removal*: We begin by removing all `assert` statements from \mathcal{P} to reduce the likelihood of abnormal termination in the mutated program \mathcal{P}' .

(b) *Weak-Assignment Transformations*: We randomly alter the right-hand side of weak assignment operators (`<--`). Recall that `<--` does not introduce constraints, meaning that modifying its right-hand side does not affect the constraints derived from the original Circom file. Based on the popular mutation strategies used in mutation testing [44], we apply two types of mutations: 1) By default, we replace the right-hand expression with a random value over the finite field sampled from the skewed distribution (see § 4.2); 2) If the right-hand side contains an operator, we may also substitute it with a different operator with the same category, where we categorize the operators into arithmetic, bitwise, and logical operators. Note that mutating the strong assignment (`<==`) is ineffective, as it is likely to violate the associated condition.

We implement two optional mutation methods, random value addition and expression deletion. We show detailed results in Appendix D.2.

5.2 Target Selection with Static Analyzers

Finding bugs in ZK circuits is particularly challenging due to the vast search space imposed by large finite fields. To overcome this challenge, zkFuzz leverages static analysis for selecting promising targets that guide both program mutation and input generation.

Table 3: Summary of static target selectors used in zkFuzz.

Target Selector	Type	Pattern
Zero-Division	Input Generation	<code>y <-- x1/x2</code>
Invalid Array Subscript	Input Generation	<code>x[too_big_val]</code>
Hash-Check	Input Generation	<code>h(x1) == x2</code>
White List	Prog. Mutation	<code>IsZero, Num2Bits</code>

Zero-Division Pattern. As discussed in § 4.2, the *zero-division* pattern is a common source of under-constrained circuits where both the division’s numerator and denominator can be zero. Since Circom lacks native support for non-quadratic constraints like division, programmers must manually specify equivalent quadratic constraints. Notably, when the denominator is zero, Circom treats the division as yielding zero.

Code 6 illustrates a simple case of this issue: when both numerator and denominator are zero, the multiplication constraint (see line 7) is trivially satisfied for any value of y , due to missing checks for zero-division. Appendix B also presents a real-world circuit example from *circomlib*, where zkFuzz successfully detects an under-constrained condition using only the *zero-division* selector.

To find this problem, we introduce a target selector that specifically seeks input combinations setting both the numerator and the denominator to zero. When a numerator or denominator reduces to a polynomial of degree at most two in an input variable, we solve the simplified quadratic equation over finite fields to generate suitable inputs. More concretely, our approach is as follows:

(Degree 1) Consider the expression $x + a \equiv 0 \pmod q$, where x denotes the input variable and a is the remaining terms. The solution is then given by $x \equiv -a \pmod q$;

(Degree 2) Consider the quadratic expression $ax^2 + bx + c \equiv 0 \pmod q$. A solution for x is given by $x \equiv \frac{-b + \sqrt{b^2 - 4ac}}{2a} \pmod q$, provided that a square root of $b^2 - 4ac$ exists modulo q . Its existence can be verified using Euler’s criterion, and the Tonelli-Shanks algorithm [50, 57] is then employed to compute $\sqrt{b^2 - 4ac}$, yielding the solution.

If coefficients, a , b , and c in the above, involve additional input variables, their values are determined by substituting artificially generated values.

Hash-Check Pattern. A common pattern in ZK circuit is the *hash-check* pattern, where a circuit verifies that the hash of some input data matches a provided hash value (see Code 7). Naive fuzzing with program mutation may struggle to generate valid input-hash pairs, especially when cryptographic hash functions are involved. To overcome this limitation, we employ a heuristic: with a certain probability if an equality constraint (e.g., $A == B$) is encountered where A is an input signal that has not been assigned at this point, we update the assignment of A to match the value of B . This adjustment increases the likelihood that the input data will satisfy the hash-check constraint. A real-world example is shown in Appendix B, where zkFuzz identifies the circuit as under-constrained using only the *Hash-Check* selector.

```

1 template ZeroDiv() {
2   signal input x1;
3   signal input x2;
4   signal output y;
5
6   y <-- x1 / x2;
7   y * x2 == x1;
8 }
```

Code 6: Example of Zero Division pattern. If both $x1$ and $x2$ are 0, any value for y can satisfy the constraint.

```

1 template HashCheck() {
2   signal input x1;
3   signal input x2;
4
5   component h = Hash();
6   h.x <== x1;
7   x2 == h.y;
8 }
```

Code 7: Example of Hash Check pattern. Finding $x1$ whose hash is x requires inverting the hash function.

Invalid Array Subscript. Circom-generated programs may crash due to assert violations or out-of-range array indices. Although our mutation algorithm removes assertions, crashes can still occur from invalid array subscripts. To address this, we monitor for repeated out-of-range errors and, when detected, shrink the maximum value of generated input to the minimum value of array dimensions in that program. This constraint can reduce the chance of triggering such crashes, ensuring the fuzzing remains stable.

White List. Existing research and public audit reports have formally proven the safety of certain commonly used circuits, even if they contain weak assignments. Since mutating these circuits would immediately lead to constraint violations, zkFuzz allows specifying a *white list* of circuit templates to be skipped during the mutation phase, concentrating the mutation efforts on less well-understood or more error-prone components. In this study, we include the two most commonly used *circomlib* circuits with weak assignments, `IsZero` and `Num2Bits`, in the white list.

Collectively, these target selectors enable zkFuzz to navigate the enormous search space inherent in ZK circuits efficiently. By strategically guiding both program mutation and input generation, they significantly enhance our ability to detect critical vulnerabilities.

6 Evaluation

In this section, we present a comprehensive experimental evaluation designed to address the following research questions:

- **RQ1:** How effective is zkFuzz at detecting vulnerabilities in real-world zero-knowledge (ZK) circuits?
- **RQ2:** How quickly can zkFuzz detect bugs?
- **RQ3:** Can zkFuzz uncover previously unknown bugs in production-grade ZK circuits?
- **RQ4:** What is the individual contribution of each heuristic and static analyzer in zkFuzz to efficient bug detection?
- **RQ5:** How sensitive is zkFuzz’s performance?

We further consider the following questions in Appendix D:

- **RQ6:** Is TCCT more general than existing formulations?
- **RQ7:** Is there an alternative mutation strategy?
- **RQ8:** How effective are the other baselines (the default version of ZKAP and circomspect with recursive analysis)?
- **RQ9:** How many over-constrained circuits are observed when the `constraint_assert_disabled` flag is set?

Benchmark Datasets. We evaluate zkFuzz on 354 real-world test suites written in Circom. Our benchmark extends the dataset from ZKAP [59] by adding new test cases from 30 additional projects. Following Pailoor et al. [43], we classify the circuits into four categories based on the number of clauses in the constraint, denoted as $|C|$ (e.g., for $C(x, z, y) := (x * x = z) \wedge (z * z = y)$, $|C| = 2$). The categories are defined as follows: *Small* ($|C| < 100$); *Medium* ($100 \leq |C| < 1000$); *Large* ($1000 \leq |C| < 10000$); *Very Large* ($10000 \leq |C|$). Detailed benchmark stats are in Appendix C.

Configurations. We set a timeout of 2 hour and a maximum of 50000 generations for all our tests. We also set the number of program mutants per generation to 30, each paired with generated inputs of the same size. Program mutation and crossover operations are applied with probabilities of 0.3 and 0.5, respectively. By default, the right-hand side of a weak assignment is replaced with a random constant. However, if it consists of an operator expression, the operator is randomly substituted with a probability of 0.1. When a zero-division pattern is detected, we analytically solve for the variable and substitute it with the computed solution with a probability of 0.2. Those parameters are determined with reference to prior works on genetic algorithm mutation testing [25, 37, 52], and we empirically demonstrate the robustness of zkFuzz across different hyperparameter settings in § 6.5. We run zkFuzz with five different random seeds. All experiments were run on an Intel(R) Xeon(R) CPU @ 2.20GHz and 31GB of RAM, running Ubuntu 22.04.3 LTS.

Baseline Approaches. We compare zkFuzz against four state-of-the-art Circom bug detection tools: Circomspect [42], ZKAP [59], Picus [43], and ConsCS [32]. We use two SMT solvers, Z3 [17] and CVC5 [6] for Picus. To ensure a fair comparison, we do not count warnings related to `IsZero` and `Num2Bits` as false positives, since zkFuzz explicitly whitelists these templates. Additionally, ZKAP’s *unconstrained signal* (*US*) detector flags any circuit containing unused signals as unsafe. Although the original ZKAP paper considers

unused inputs problematic, our work does not regard them as bugs (see § 7). Moreover, as noted in [59], the primary source of ZKAP’s false positives is its *unconstrained sub-circuit output* (*USCO*) check, which targets unused component outputs. Therefore, we exclude both *US* and *USCO* from our main results to improve ZKAP’s precision. Note that the original Circomspect does not analyze internally called templates recursively. Lastly, zkFuzz-Naive is a simplified variant of zkFuzz that mutates both strong and weak assignments, uses a constant function as the fitness function, and disables all static-based target selectors listed in Tab. 3.

6.1 RQ1: Effectiveness of zkFuzz

Tab. 4 summarizes the total number of unique bugs detected by each tool across different benchmark categories. We manually check all potential bugs flagged by at least one tool and categorize them into TP (True Positive), which refers to cases where the tool correctly flags ZK circuits as unsafe, and FP (False Positive), which refers to cases where the tool incorrectly flags a test suite as unsafe. Precision is calculated as $TP / (TP + FP)$. Since there may be undetected bugs not flagged by any tool, we do not report true or false negatives. The #Bugs column shows the number of unique vulnerabilities detected by at least one tool. A bug in a Circom template (§ 2.3) may affect all programs that instantiate the template, but we count it as a single bug to avoid double counting. Moreover, when a vulnerability is detected in multiple categories (for example, if a bug is identified in both a small and a medium circuit because they use the same buggy template), it is counted only in the smallest applicable category to prevent double counting. Note this way of counting favors existing formal tools due to their limited scalability. We report the maximum and minimum TP of zkFuzz with five different random seeds. Our evaluation shows that zkFuzz significantly and stably outperforms existing methods, detecting about 96% (66/69) of all bugs without any false positives. While very large templates present a bottleneck for dynamic testing, causing zkFuzz to underperform Circomspect in the "Very Large" category slightly, zkFuzz still identifies 66% (4/6) of the bugs.

In addition, zkFuzz-Naive, which performs the search without any guidance, still detects more bugs than other existing tools, demonstrating the strength of our core approach. The default zkFuzz detects 10% more bugs overall compared to zkFuzz-Naive, especially in larger circuits.

Our benchmark shows that the performance of ZKAP is worse than that reported in [59]. This stems from not classifying potential logic issues, such as unused inputs, as vulnerabilities. Furthermore, the official implementation of ZKAP is known to crash on certain templates written in recent Circom versions—an issue acknowledged by ZKAP’s developers but yet to be fixed. The results of ZKAP with *US* and *USCO* detectors and our modified version of Circomspect with recursive analysis are provided in Appendix D.

Our analysis also reveals that 53.5% of root causes stem from computation abort bugs, while 46.5% are due to non-deterministic behavior. This result highlights the importance of comprehensively addressing both types of bugs, whereas Picus and ConsCS focus solely on non-deterministic cases.

ConsCS detects fewer bugs overall compared to Picus. Although Jiang et al. [32] reports a higher success rate for ConsCS, it is

Table 4: The number of unique bugs detected by each tool, categorized by circuit size (measured by the size of constraint, $|C|$). TP and FP denote true positive and false positive, respectively. Precision is computed as $TP / (TP + FP)$. Max / Min refers to the highest and lowest number of detected bugs across five different random seeds. The results show that zkFUZZ significantly outperforms the existing methods. The magnitude of TP and Precision, and FP are highlighted in blue and red, respectively.

Constraint Size	#Bench	#Bug	Circomspect [42]		ZKAP [59]		Picus [43] (Z3 / CVC5)		ConsCS [32]		zkFUZZ (Naive) (Max / Min)		zkFUZZ (Ours) (Max / Min)	
			TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Small	144	41	37	16	32	9	19 / 20	0	14	0	40 / 39	0	41 / 41	0
Medium	51	6	3	6	0	12	0	0	0	0	6 / 6	0	6 / 6	0
Large	76	16	6	12	3	2	0	0	0	0	12 / 12	0	15 / 15	0
Very Large	83	6	5	18	2	2	0	0	0	0	2 / 2	0	4 / 3	0
Total	354	69	51	52	37	25	19 / 20	0	14	0	60 / 59	0	66 / 65	0
Precision			0.50		0.60		1.00		1.00		1.00		1.00	

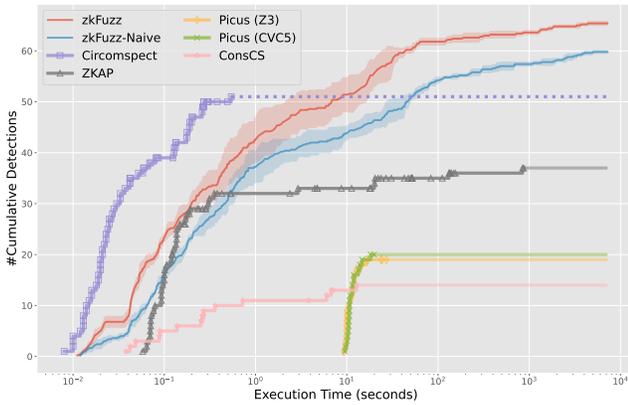


Figure 3: Detection Time Analysis. For zkFUZZ, we plot the average and standard deviation from five different random seeds. zkFUZZ is superior to other methods.

important to note that their evaluation measures both safe and unsafe circuits (demonstrating either security or vulnerability). In contrast, our experiment focuses solely on vulnerability detection and does not include formal verification of circuit safety.

Result 1: Effectiveness of zkFUZZ

zkFUZZ has efficiently detected 66 out of 69 vulnerabilities in the benchmark of real-world ZK circuits without any false positives, demonstrating its reliable ability to identify critical security flaws in ZK circuits.

6.2 RQ2: Detection Speed

Fig. 3 illustrates the cumulative number of unique vulnerabilities detected by different tools over time. For zkFUZZ, we depict the average and 1-sigma regions across five trials with different random seeds. Notably, zkFUZZ identifies over 90% of bugs within the first 100 seconds, although zkFUZZ-Naive takes more than one hour to achieve the same number of detected bugs, showing the importance of guided search with our fitness function and target selectors.

Although ZKAP is a static analyzer, it operates more slowly than zkFUZZ due to the computational overhead of compiling Circom files to LLVM and analyzing its dependency graph. Similarly, Picus exhibits slower performance, as it relies on expensive queries to SMT solvers. Our results show that ConsCS finds bugs faster than Picus confirming the findings of Jiang et al. [32]. The dashed line of Circomspect indicates that it finishes the analysis of all test cases before the timeout.

Result 2: Detection Speed of zkFUZZ

zkFUZZ identifies 90% of bugs within just 100 seconds, demonstrating its practical effectiveness over traditional formal methods. The comparison with zkFUZZ-Naive highlights the advantage of guided search, achieving a speedup of more than 30 times.

6.3 RQ3: Previously Unknown Bugs

Out of 66 identified bugs zkFUZZ, 38 were previously unknown. Of these, 18 were found in production-level services and deployed smart contracts, 18 were confirmed by developers, and 6 have already been fixed. One example is shown in Code 8, found in the *passport-zk-circuits* [46], a Web3 project that raised \$2.5 million. For instance, when $day=4$, the expected assignment is $\{dayD:0, dayR:4\}$ although $\{dayD:0, dayR:-6\}$ also satisfies the constraint.

```

1 template DateEncoder() {
2   signal output encoded;
3   signal input day, month, year;
4
5   signal dayD <-- (day \ 10);
6   signal dayR <-- (day % 10);
7   dayD * 10 + dayR == day;
8   signal dayE <== (dayD * 2**8 + dayR) + (2**4 + 2**5 +
9     2**12 + 2**13);
10  //Encode month and year in the same manner
11  encoded <== yearE * 2**32 + monthE * 2**16 + dayE; }

```

Code 8: A previously unknown bug detected by zkFUZZ in *passport-zk-circuits* [46]. This circuit is under-constrained.

Our zkFUZZ can generate counterexamples, unlike static analyzers such as ZKAP and Circomspect, significantly reducing the need for manual inspection after automated vulnerability detection.

Among the six fixed vulnerabilities identified by zkFUZZ, the average time to merge a fix after reporting was just 4.3 days—10 to 20 times faster than the typical resolution time for accepted security issues in open-source projects [10, 48]. As of now, no developer has concluded that a report from zkFUZZ was a false positive.

Result 3: Previously Unknown Bugs found by zkFUZZ

zkFUZZ can uncover 38 previously unknown bugs in production-grade ZK circuits, with 18 of them found in production-level services and deployed smart contracts.

6.4 RQ4: Ablation Study of Target Selectors

Tab. 5 shows the results of ablation studies where we assess the impact of removing individual target selectors: *error-based fitness score*, *skewed distribution*, *white list*, *invalid array subscript*, *hash-check* and *zero-division*. We report the average and standard deviation of the cumulative number of unique bugs found at each time checkpoint among five different random seeds. For skewed distribution, we use a less skewed distribution; binary values (0 and 1) with 15%, small positive integers (2 to 100) with 34%, larger values near q ($q - 1000$ to $q - 1$) with 50%, and all other values (11 to $q - 1001$) with 1%. Fig. 4 also presents the relative execution time for discovering each unique bug, normalized against the default configuration.

Among these factors, the choice of skewed distribution has the largest initial impact. Our default setting (Ours) finds 10 more bugs within the first 100 seconds compared to the less skewed distribution. The relative execution time indicates that adopting a less skewed distribution or removing the whitelist slows down detection by over 100 \times . In contrast, removing checks for invalid array subscripts, hash integrity, or division by zero prevents the fuzzer from detecting several bugs. However, after a reasonable amount of time, the performance of the less skewed distribution also converges to a similar number of unique detected bugs, demonstrating the stability and robustness of zkFUZZ regardless of the specific choice of skewed distributions. Additionally, removing static analyzers, such as checks for invalid array subscripts, hash mismatches, and division by zero, results in several bugs being missed, highlighting their importance in the system’s overall effectiveness.

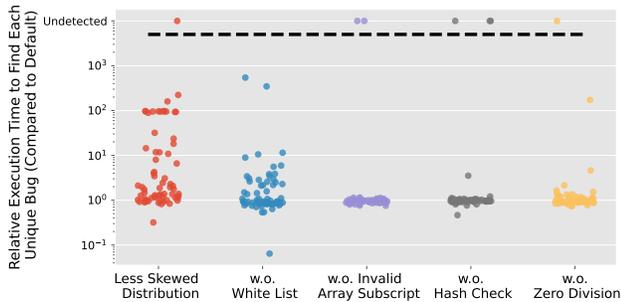


Figure 4: Relative execution time for discovering each unique bug compared to the default setting. Removing each heuristic degrades detection performance by over 100 \times .

Result 4: Individual Contribution of Each Target Selector

Each target selector in zkFUZZ significantly enhances bug detection by increasing detection speed by more than 100 times and enabling the discovery of bugs that cannot be found without them.

6.5 RQ5: Hyperparameter Sensitivity

We further examine how mutation strength and population size affect performance. Fig. 5 compares configurations using weak mutation (mutation and crossover probabilities of 0.1) versus strong mutation (both set to 0.7) and small population (the number of program mutants and generated inputs of 10) versus large population (both set to 50). Our results indicate that weak mutation and a large population yield slightly inferior performance compared to the default setting, showing the importance of sufficient exploration in navigating the vast search space since weak mutation slows the speed of exploration while larger populations cause the fuzzer to execute more programs before steering mutants toward promising directions, increasing overhead. Nonetheless, regardless of hyperparameter choices, cumulative detections converge to similar levels.

Result 5: Stability of zkFUZZ

zkFUZZ’s bug detection performance remains stable across different hyperparameter choices.

Ethical Considerations. During our research, we identified several previously unknown vulnerabilities in public projects. To ensure responsible disclosure, we contacted the project developers directly via email or social media whenever possible. We opened GitHub issues if we received no response or direct contact wasn’t available. All vulnerability reports were submitted at least 30 days before this paper, with most sent over 60 days prior.

7 Related Work

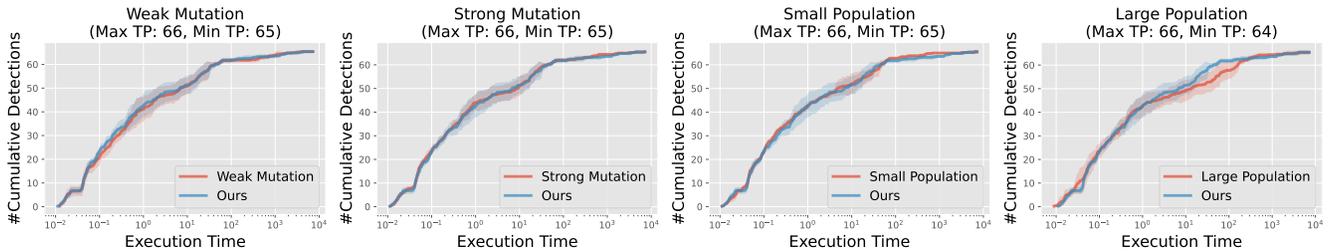
Zero-knowledge proofs are vital for privacy and trust in decentralized systems, including cryptocurrencies and smart contracts. However, flaws in circuit constraints can compromise security by permitting malicious proofs or rejecting valid ones [36, 56]. We summarize different approaches to automatically detect various types of ZKP bugs below.

Static Analysis. These approaches are widely used but suffer from high false positives. For example, Circomspect [42] flags weak assignments ($<--$) indiscriminately, leading to many false positives. ZKAP [59] improves accuracy using circuit dependence graphs, yet it still identifies unconstrained inputs as potential vulnerabilities, which may be expected behavior. Despite identifying issues, static analyzers can’t generate counterexamples, requiring manual verification of flagged bugs.

Formal Methods. Technique using formal methods rigorously verify ZK circuit correctness but struggle with scalability and require manual annotations. Tools like CIVER [31] and Constraint

Table 5: Impact of removing individual target selector on zkFUZZ’s performance. Each selector contributes significantly to the number of detected bugs and the detection speed, demonstrating their importance.

Time (s)	Ours	Less Skewed Distribution	w.o. White List	w.o. Invalid Array Subscript	w.o. Hash Check	w.o. Zero Division
0.1	23.40 ± 1.02	18.20 ± 1.94	22.40 ± 2.87	25.00 ± 3.03	24.00 ± 1.67	20.80 ± 2.71
1.0	42.60 ± 4.50	35.20 ± 0.40	43.00 ± 2.53	43.40 ± 5.04	42.00 ± 5.06	41.40 ± 4.45
10.0	51.40 ± 3.44	43.00 ± 1.41	47.60 ± 2.33	49.60 ± 4.08	49.40 ± 3.83	50.40 ± 3.56
100.0	61.80 ± 0.75	49.20 ± 0.75	57.80 ± 2.23	60.00 ± 0.89	59.80 ± 0.75	60.80 ± 0.75
1000.0	63.60 ± 0.49	54.80 ± 2.48	62.20 ± 0.75	62.00 ± 0.63	61.60 ± 0.49	62.60 ± 0.49
3600.0	65.00 ± 0.63	62.00 ± 2.10	64.40 ± 0.49	63.20 ± 0.75	63.00 ± 0.00	64.00 ± 0.63
7200.0	65.40 ± 0.49	64.00 ± 0.89	64.60 ± 0.49	63.40 ± 0.49	63.00 ± 0.00	64.60 ± 0.49

**Figure 5: Impact of mutation strength and population size on zkFUZZ. zkFUZZ consistently maintains strong bug detection performance across various hyperparameter settings.**

Checker [19] need human input, while others like CODA [38] require rewriting circuits in another language. Fully automatic tools like Picus [43] and ConsCS [32] use SMT solvers but still face scalability challenges and offer narrower definitions of under- and over-constrained circuits than ours. For example, they don’t capture under-constrained circuits caused by unexpected inputs [59].

Dynamic Testing. Fuzzing and mutation testing have been used to detect bugs in ZKP infrastructures [27, 34, 60], typically focusing on compiler bugs. In contrast, our approach targets security vulnerabilities in individual ZK circuits.

8 Limitations and Future Work

Limitations. zkFUZZ adopts a fuzzing approach for vulnerability detection and thus inherits certain limitations. For example, the execution time of zkFUZZ is proportional to that of the target program, and slicing and partially executing the target might improve zkFUZZ. In addition, our implementation currently supports only Circom, and potential improvements include covering other DSLs.

Future Work. Integrating zkFUZZ with formal verification, static analyzers, and ML/LLM-based bug detection [20, 26, 51, 64] might further boost its scalability and flexibility. Additionally, zkFUZZ could serve as a subroutine for fuzzing ZK compilers like Cairo [22], Noir [4], and Leo [16], helping to detect discrepancies between compiled programs and circuit constraints.

9 Conclusion

We introduce zkFUZZ, a fuzzing framework for detecting ZK circuit bugs. It uses TCCT, a comprehensive and language-independent

Table 6: Comparison between zkFUZZ and existing detection methods. A partial circle represents partial capability. zkFUZZ demonstrates superior performance across all criteria, offering comprehensive automatic testing capabilities.

Method	Auto-matic	Under-Constrained	Over-Constrained	Counter Example	False Positive
CIVER [31]	✗	✓	✓	✓	No
CODA [38]	✗	✓	✓	✓	No
Constraint - Checker [19]	✗	✓	✓	✓	No
Circumspect [42]	✓	⊖	⊖	✗	Yes
Picus [43]	✓	⊖	✗	✓	No
ConsCS [32]	✓	⊖	✗	✓	No
AC4 [13]	✓	⊖	⊖	✓	No
ZKAP [59]	✓	✓	✓	✗	Yes
zkFUZZ (Ours)	✓	✓	✓	✓	No

definition of ZK circuit bugs. zkFUZZ applies program mutation and target-guided artificial inputs to find under- and over-constrained circuits without false positives. It discovered 38 previously unknown bugs in real-world circuits.

References

- [1] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. 2024. Zero-knowledge proofs of training for deep neural networks. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer*

- and Communications Security. 4316–4330.
- [2] Sanjeev Arora and Boaz Barak. 2009. *Computational complexity: a modern approach*. Cambridge University Press.
- [3] Arman Aurobindo. 2025. *circum-zkVerify*. GitHub repository. <https://github.com/armanthepythonguy/circum-zkVerify>
- [4] Aztec Network. 2024. *Noir: The Universal Language of Zero-Knowledge*. <https://azt3c-st.webflow.io/noir>. Accessed: February 23, 2025.
- [5] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindström, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. *zklogIn: Privacy-preserving blockchain authentication with existing credentials*. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3182–3196.
- [6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. *cvc5: A versatile and industrial-strength SMT solver*. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.
- [7] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. *Circum: A circuit description language for building zero-knowledge applications*. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 4733–4751.
- [8] Marta Belles-Munoz, Miguel Isabel, Jose Luis Munoz-Tapia, Albert Rubio, and Jordi Baylina. 2023. *Circum: A Circuit Description Language for Building Zero-Knowledge Applications*. *IEEE Transactions on Dependable and Secure Computing* 20, 06 (Nov. 2023), 4733–4751. doi:10.1109/TDSC.2022.3232813
- [9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53 (Feb. 2010), 66–75. Issue 2.
- [10] Noah Bühlmann and Mohammad Ghafari. 2022. How do developers deal with security issue reports on GitHub?. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (Virtual Event) (SAC ’22)*. Association for Computing Machinery, New York, NY, USA, 1580–1589. doi:10.1145/3477314.3507123
- [11] Chainlight. 2023. *zkSync-era-write-query-poc*. <https://github.com/chainlight-io/zksync-era-write-query-poc/tree/main> Accessed: 2025-04-01.
- [12] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. 2024. {SoK}: What Don’t We Know? Understanding Security Vulnerabilities in {SNARKs}. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3855–3872.
- [13] Hao Chen, Guoqiang Li, Minyu Chen, Ruibang Liu, and Sinka Gao. 2024. AC4: Algebraic Computation Checker for Circuit Constraints in ZKPs. *arXiv preprint arXiv:2403.15676* (2024).
- [14] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. *Jigsaw: Efficient and scalable path constraints fuzzing*. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 18–35.
- [15] Thomas Chen, Hui Lu, Teeramet Kumpittaya, and Alan Luo. 2022. A review of zk-snarks. *arXiv preprint arXiv:2202.06877* (2022).
- [16] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. 2021. *Leo: A programming language for formally verified, zero-knowledge applications*. *Cryptology ePrint Archive* (2021).
- [17] Leonardo De Moura and Nikolaj Bjørner. 2008. *Z3: An efficient SMT solver*. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
- [18] Privacy & Scaling Explorations. 2020. *Minimal Anti-Collusion Infrastructure (MACI)*. <https://github.com/privacy-scaling-explorations/maci>. <https://github.com/privacy-scaling-explorations/maci> Accessed: 2025-04-14.
- [19] Yongming Fan, Yuquan Xu, and Christina Garman. 2024. *Snarkprobe: An automated security analysis framework for zkSNARK implementations*. In *International Conference on Applied Cryptography and Network Security*. Springer, 340–372.
- [20] Rundong Gan, Liyi Zhou, Le Wang, Kaihua Qin, and Xiaodong Lin. 2024. *DeFiAligner: Leveraging Symbolic Analysis and Large Language Models for Inconsistency Detection in Decentralized Finance*. In *6th Conference on Advances in Financial Technologies (AFT 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 7–1.
- [21] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. 2023. *Experimenting with zero-knowledge proofs of training*. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1880–1894.
- [22] Lior Goldberg, Shahar Papimi, and Michael Riabzev. 2021. *Cairo – a Turing-complete STARK-friendly CPU architecture*. *Cryptology ePrint Archive*, Paper 2021/1063. <https://eprint.iacr.org/2021/1063>
- [23] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. *Poseidon: A new hash function for {Zero-Knowledge} proof systems*. In *30th USENIX Security Symposium (USENIX Security 21)*. 519–535.
- [24] Jahid Hasan. 2019. Overview and applications of zero knowledge proof (ZKP). *International Journal of Computer Science and Network* 8, 5 (2019), 2277–5420.
- [25] Ahmad Hassanat, Khalid Almohammadi, Esra’a Alkafaween, Eman Abunawas, Awni Hammouri, and VB Surya Prasath. 2019. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information* 10, 12 (2019), 390.
- [26] Zheyuan He, Zihao Li, Sen Yang, He Ye, Ao Qiao, Xiaosong Zhang, Xiapu Luo, and Ting Chen. 2024. *Large language models for blockchain security: A systematic literature review*. *arXiv preprint arXiv:2403.14280* (2024).
- [27] Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholtz, and Maria Christakis. 2024. *Fuzzing Processing Pipelines for Zero-Knowledge Circuits*. *arXiv preprint arXiv:2411.02077* (2024).
- [28] iden3. 2018. *snarkjs: zkSNARK implementation in JavaScript & WASM*. <https://github.com/iden3/snarkjs>. Accessed: 2025-04-14.
- [29] iden3. 2025. *Library of basic circuits for circum*. <https://github.com/iden3/circumlib>. Accessed: 2025-04-01.
- [30] iden3 contributors. 2023. *Circumlib: add SafeLessThan template to comparators with range checks over inputs (#86)*. <https://github.com/iden3/circumlib/pull/86>. Accessed: 2025-04-14.
- [31] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. 2024. *Scalable Verification of Zero-Knowledge Protocols*. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 133–133.
- [32] Jinan Jiang, Xinghao Peng, Jinzhao Chu, and Xiapu Luo. 2025. *ConsCS: Effective and Efficient Verification of Circum Circuits*. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 737–737.
- [33] Ryan Lavín, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. 2024. *A Survey on the Applications of Zero-Knowledge Proofs*. *arXiv preprint arXiv:2408.00243* (2024).
- [34] Philipp Leeb. 2024. *Metamorphic Testing of ZKC Infrastructure*. Ph.D. Dissertation. Technische Universität Wien.
- [35] Feng Li and Bruce McMillin. 2014. Chapter Two - A Survey on Zero-Knowledge Proofs. *Advances in Computers*, Vol. 94. Elsevier, 25–69. doi:10.1016/B978-0-12-800161-5.00002-5
- [36] Junkai Liang, Daqi Hu, Pengfei Wu, Yunbo Yang, Qingni Shen, and Zhonghai Wu. 2025. *SoK: Understanding zk-SNARKs: The Gap Between Research and Practice*. *arXiv preprint arXiv:2502.02387* (2025).
- [37] Wen-Yang Lin, Wen-Yung Lee, and Tzung-Pei Hong. 2003. Adapting crossover and mutation rates in genetic algorithms. *J. Inf. Sci. Eng.* 19, 5 (2003), 889–903.
- [38] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Isil Dillig, and Yu Feng. 2024. *Certifying zero-knowledge circuits with refinement types*. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1741–1759.
- [39] Joseph K Liu, Man Ho Au, Tsz Hon Yuen, Cong Zuo, Jiawei Wang, Amin Sakzad, Xiapu Luo, Li Li, and Kim-Kwang Raymond Choo. 2020. *Privacy-preserving COVID-19 contact tracing app: a zero-knowledge proof approach*. *Cryptology ePrint Archive* (2020).
- [40] Eduardo Morais, Tommy Koens, Cees Van Wijk, and Aleksei Koren. 2019. A survey on zero knowledge range proofs and applications. *SN Applied Sciences* 1 (2019), 1–17.
- [41] noir lang. 2025. *ZK Benchmark*. https://github.com/noir-lang/zk_bench. Accessed: 2025-04-03.
- [42] Trail of Bits. 2024. *Circumspect: A static analyzer and linter for the Circum zero-knowledge DSL*. <https://github.com/trailofbits/circumspect>. Accessed: 2025-02-27.
- [43] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van GEFen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. 2023. *Automated detection of under-constrained circuits in zero-knowledge proofs*. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1510–1532.
- [44] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. *Mutation testing advances: an analysis and survey*. In *Advances in computers*. Vol. 112. Elsevier, 275–378.
- [45] Protocol Labs. 2023. *The Future of ZK Proofs*. <https://www.protocol.ai/protocol-labs-the-future-of-zk-proofs.pdf>. Accessed: 2025-02-18.
- [46] Rarimo. 2025. *Passport ZK Circuits: Verifying Biometric Passports with Zero Knowledge Proofs (SNARKs)*. <https://github.com/rarimo/passport-zk-circuits> Accessed: 2025-03-09.
- [47] Noraini Mohd Razali, John Geraghty, et al. 2011. Genetic algorithm performance with different selection strategies in solving TSP. In *Proceedings of the world congress on engineering*, Vol. 2. International Association of Engineers Hong Kong, China, 1–6.
- [48] Hocine Rebatchi, Tégawendé F Bissyandé, and Naouel Moha. 2024. *Dependabot and security pull requests: large empirical study*. *Empirical Software Engineering* 29, 5 (2024), 128.
- [49] Sushmita Ruj. 2024. *Zero-knowledge proofs for blockchains*. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 67–68.
- [50] D Shanks. 1972. *Five number theoretical algorithms*. In *proceeding Second Manitoba Conference on Numerical Mathematics*, University of Manitoba, Winnipeg, Manitoba, Canada.

- [51] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.
- [52] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695.
- [53] Ayush Shukla. 2023. circom-monolith. <https://github.com/shuklaayush/circom-monolith>. Accessed: 2025-04-14.
- [54] Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. 2023. Automated analysis of Halo2 circuits. *Cryptology ePrint Archive* (2023).
- [55] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 179–197.
- [56] Xueyan Tang, Lingzhi Shi, Xun Wang, Kyle Charbonnet, Shixiang Tang, and Shixiao Sun. 2024. Zero-knowledge proof vulnerability analysis and security auditing. *Cryptology ePrint Archive* (2024).
- [57] Alberto Tonelli. 1891. Bemerkung über die Auflösung quadratischer Congruenzen. *Nachrichten von der Königl. Gesellschaft der Wissenschaften und der Georg-Augusts-Universität zu Göttingen* 1891 (1891), 344–346.
- [58] Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. Sok: Where to fuzz? assessing target selection methods in directed fuzzing. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1539–1553.
- [59] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. 2024. Practical Security Analysis of {Zero-Knowledge} Proof Circuits. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1471–1487.
- [60] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers. In *NDSS*.
- [61] Zhibo Xing, Zijian Zhang, Jiamou Liu, Ziang Zhang, Meng Li, Liehuang Zhu, and Giovanni Russello. 2023. Zero-knowledge proof meets machine learning in verifiability: A survey. *arXiv preprint arXiv:2310.14848* (2023).
- [62] Xiaohui Yang and Wenjie Li. 2020. A zero-knowledge-proof-based digital identity management scheme in blockchain. *Computers & Security* 99 (2020), 102050.
- [63] Zcash. 2022. Halo2: The Halo2 zero-knowledge proving system. <https://github.com/zcash/halo2>. Accessed: 2025-04-14.
- [64] Kumpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. *arXiv preprint arXiv:2501.19282* (2025).
- [65] Y Zhang and Z Fan. 2024. Research on Zero knowledge with machine learning. *Journal of Computing and Electronic Information Management* 12, 2 (2024), 105–108.
- [66] zkemail. 2025. zk-regex. <https://github.com/zkemail/zk-regex>. Accessed: 2025-04-02.

A Proofs

PROOF OF THEOREM 3.9. First, we will reduce the Boolean Satisfiability Problem (SAT) to the problem of deciding whether a program is under-constrained (or over-constrained). SAT is known to be NP-complete.

(Detection of Under-Constrained Instance): Let $\phi = \bigwedge_{j=1}^m \phi_j$ be a Boolean formula in conjunctive normal form (CNF) with variables x_1, \dots, x_n , where ϕ_j is the j -th conjunction of ϕ and m is the total number of conjunctions. Then, we construct a program $\langle \mathcal{P}, C \rangle$ as follows:

$$\begin{aligned} \mathcal{P} : \mathbb{F}_2^n &\rightarrow ((\mathbb{F}_2^0 \cup \{\perp\}) \times (\mathbb{F}_2^0 \cup \{\perp\})) \\ C : \{\mathbb{F}_2^n \times \mathbb{F}_2^0 \times \mathbb{F}_2^0 &\rightarrow \{0, 1\}\} \end{aligned}$$

$$\begin{aligned} \forall (x_1, \dots, x_n) \in \mathbb{F}_2^n, \quad \mathcal{P}((x_1, \dots, x_n)) &= (\perp, \perp) \\ C &= \phi \end{aligned}$$

Since \mathcal{P} is a program that always returns (\perp, \perp) , $\mathcal{T}(\mathcal{P}) = \emptyset$.

Now, we claim that $\langle \mathcal{P}, C \rangle$ is under-constrained if and only if ϕ is satisfiable.

(\Rightarrow) If $\langle \mathcal{P}, C \rangle$ is under-constrained, there exists $(x, y) \in \Pi(\mathcal{S}(C))$ such that $(x, y) \notin \Pi(\mathcal{T}(\mathcal{P}))$. This means that there exists at least one $(x, (, y) \in \mathcal{S}(C)$. This implies that x satisfies $C = \phi$. Thus, x is a satisfying assignment for ϕ .

(\Leftarrow) If ϕ is satisfiable, let x' be a satisfying assignment. Then, we have $(x', (,) \in \Pi(\mathcal{S}(C))$ because x' satisfies all clause constraints. However, $(x', (,) \notin \Pi(\mathcal{T}(\mathcal{P}))$ because $\mathcal{T}(\mathcal{P})$ is empty. Thus, $\langle \mathcal{P}, C \rangle$ is under-constrained.

Since the entire reduction can be performed in polynomial time, we have shown that deciding whether $\langle \mathcal{P}, C \rangle$ is under-constrained is at least as hard as SAT. Therefore, deciding whether a program is under-constrained is NP-hard.

(Detection of Over-Constrained Instance): Likewise, we construct a program $\langle \mathcal{P}, C \rangle$ as follows:

$$\begin{aligned} \mathcal{P} : \mathbb{F}_2^n &\rightarrow (\mathbb{F}_2^0 \cup \{\perp\}) \times (\mathbb{F}_2^1 \cup \{\perp\}) \\ C : \{\mathbb{F}_2^n \times \mathbb{F}_2^0 \times \mathbb{F}_2^1 &\rightarrow \{0, 1\}\} \end{aligned}$$

$$\begin{aligned} \mathcal{P}(x) &= \begin{cases} ((, (1)) & \text{if all clauses in } \phi \text{ is satisfied with } x \\ (\perp, \perp) & \text{otherwise} \end{cases} \\ C &= \text{false} \end{aligned}$$

Since C always returns false, $\mathcal{S}(C) = \emptyset$.

Now, we claim that $\langle \mathcal{P}, C \rangle$ is over-constrained if and only if ϕ is satisfiable.

(\Rightarrow) If $\langle \mathcal{P}, C \rangle$ is over-constrained, there exists $(x, z, y) \in \mathcal{T}(\mathcal{P})$ such that $(x, z, y) \notin \mathcal{S}(C)$. This implies that x satisfies all clauses in ϕ , meaning x is a satisfying assignment for ϕ .

(\Leftarrow) Suppose ϕ is satisfiable, and let x' be a satisfying assignment. Then, we have $(x', (, (1)) \in \mathcal{T}(\mathcal{P})$ because x' satisfies all clauses in ϕ . However, $(x', (, (1)) \notin \mathcal{S}(C)$ because $\mathcal{S}(C)$ is empty. Thus, $\langle \mathcal{P}, C \rangle$ is over-constrained.

Since the entire reduction can be performed in polynomial time, we have shown that deciding whether $\langle \mathcal{P}, C \rangle$ is over-constrained is at least as hard as SAT. Therefore, deciding whether a program is over-constrained is NP-hard.

(Complexity of TCCT): Next, we prove the Trace-Constraint Consistency Test's co-NP-completeness by reducing the Boolean tautology problem, a known co-NP-complete problem, to it.

Let ϕ_1 and ϕ_2 be Boolean formulas with variables $x = (x_1, \dots, x_n)$. The Boolean equivalence problem asks whether $\phi_1 \equiv \phi_2$, i.e., ϕ_1 and ϕ_2 evaluate to the same value for all possible assignments. The Boolean tautology problem aims to determine whether all possible assignments to a Boolean formula yield true.

LEMMA A.1. *The Boolean equivalence problem is reducible to the Boolean tautology problem.*

PROOF. $\phi_1 \equiv \phi_2$ if and only if $(\phi_1 \wedge \phi_2) \vee ((\neg\phi_1) \wedge (\neg\phi_2))$ is a tautology. \square

LEMMA A.2. *The Boolean tautology problem is reducible to the Boolean equivalence problem.*

PROOF. ϕ_1 is a tautology if and only if $\phi_1 \equiv \text{TRUE}$ \square

By these lemmas, the Boolean tautology problem and the Boolean equivalence problem are polynomial-time reducible to each other and, thus, are of equivalent complexity.

Given two Boolean formulas ϕ_1 and ϕ_2 , we construct an instance of the Trace-Constraint Consistency Test $\langle \mathcal{P}, C \rangle$ over \mathbb{F}_2 as follows:

$$\mathcal{P} : \mathbb{F}_2^n \rightarrow (\mathbb{F}_2^0 \cup \{\perp\}) \times (\mathbb{F}_2^1 \cup \{\perp\})$$

$$C : \{\mathbb{F}_2^n \times \mathbb{F}_2^0 \times \mathbb{F}_2^1 \rightarrow \{0, 1\}\}$$

$$\mathcal{P}((x_1, \dots, x_n)) = \begin{cases} ((\perp), (1)) & \text{if } (x_1, \dots, x_n) \text{ satisfies } \phi_1 \\ ((\perp), (0)) & \text{otherwise} \end{cases}$$

$$C(x, z, y) = \begin{cases} 1 & \text{if } x \text{ satisfies } \phi_2 \text{ and } y = 1 \\ 1 & \text{if } x \text{ does not satisfy } \phi_2 \text{ and } y = 0 \\ 0 & \text{otherwise} \end{cases}$$

Since there are no intermediate values, Eq. 3 is equivalent to $\mathcal{T}(\mathcal{P}) = \mathcal{S}(C)$. By the definition of \mathcal{P} and C , we have that $\phi_1 \equiv \phi_2$ if and only if $\langle \mathcal{P}, C \rangle$ is well-constrained.

This reduction can be performed in polynomial time. Since the Boolean tautology problem is co-NP-complete [2] and we have shown a polynomial-time reduction to the Trace-Constraint Consistency Test, we conclude that it is co-NP-complete. \square

B Additional Examples of ZK Program

Correct Implementation of RShift1. Code 9 presents a secure implementation of a 1-bit right shift in Circom. The input is first converted into a bit array, the shift operation is applied to this array, and the result is then converted back into the output.

```

1 template RShift1(N) {
2   signal input x;
3   signal output y;
4
5   component x_bits = Num2Bits(N);
6   x_bits.in <= x;
7
8   signal y_bits[ N - 1 ];
9   for (var i = 0; i < N - 1; i++) {
10    y_bits[i] <= x_bits.out[i + 1];
11  }
12
13  component y_num = Bits2Num(N - 1);
14  y_num.in <= y_bits;
15  y <= y_num.out;
16 }
```

Code 9: Secure implementation of 1-bit right shift.

LessThan. While explicit constraints are readily apparent in circuit designs, implicit assumptions used in Circom’s de facto standard library, *circomlib* [29], can also introduce subtle under-constrained bugs if not properly accounted for.

Consider the ZK program in Code 11 from *zk-regex* [66] which validates the range of the given characters and is expected to abnormally terminate when they contain values larger than 255: This template uses the *LessThan* circuit from *circomlib*, which implicitly assumes that both inputs are represented by n bits or fewer. Although this assumption is well known[30, 42], the *EmailAddrRegex*

template does not validate the bitwidth of the inputs provided to *LessThan*.

The critical vulnerability arises from the potential overflow in Line 7 and 8 of *LessThan* when input a is excessively large. Note that *Num2Bits* is a template converting an input to the bit expression. To illustrate, suppose $q = 401$ and $\text{msg}[0] = 400$. The computation in Line 7 of *LessThan* becomes $400 + (1 \ll 8) - 255 \equiv 0 \pmod{401}$. Hence, *LessThan* outputs 1, erroneously passing the range check of *EmailAddrRegex*. Our fuzzer successfully discovered this bug, which the developer subsequently confirmed.

```

1 template LessThan(n) {      1 template EmailAddrRegex(n) {
2   assert(n <= 252);        2   signal input msg[n];
3   signal input in[2];      3
4   signal output out;       4   signal in_range_checks[n];
5   component n2b =          5   for (var i = 0; i < n; i++) {
      Num2Bits(n+1);         6     in_range_checks[i] <=
6                               LessThan(8)([msg[i],
7     n2b.in <= in[0] +      255]);
      (1<<n) - in[1];        7     in_range_checks[i] == 1;
8     out <= 1 - n2b.out[n  8     // the rest is omitted
      ];
9   }

```

Code 10: Implementation of LessThan template from circomlib [29]. This template assumes that the inputs are at most n bits wide. If the inputs exceed n bits, the comparison result may be incorrect.

Code 11: Example of an under-constrained circuit from *zk-regex* [66]. *LessThan* implicitly assumes that the bit length of both inputs are not longer than n , although *LessThan* itself does not check the range of inputs.

Over-Constrained Circuit. As described in § 2.3, the default configuration of the Circom compiler does not provide an operator that adds a condition to the constraints C without adding the corresponding assignment or assertion to the computation \mathcal{P} . Consequently, the trace set $\mathcal{T}(\mathcal{P})$ is always a subset of the constraint satisfaction set $\mathcal{S}(C)$, meaning that over-constrained circuits cannot exist.

```

1 template SplitReward() {
2   signal input x;
3   signal z;
4   signal output y;
5
6   z <- x \ 2;
7   z * 2 == x;
8   y <= z + 1;
9 }
```

Code 12: Example of over-constrained circuit. Suppose assert is not added in the computation for ==.

Table 7: Trace and Constraint Satisfaction Sets of Over-Constrained Circuit ($q = 5$).

$\mathcal{T}(\mathcal{P})$			$\mathcal{S}(C)$		
x	z	y	x	z	y
0	0	1	0	0	1
1	0	1	1	3	4
2	1	2	2	1	2
3	1	2	3	4	0
4	2	3	4	2	3

However, enabling the constraint `_assert_dissabled` flag can lead to over-constrained circuits. Code 12 presents a modified version of the *Reward* template from [59], illustrating a ZK program that performs integer division. Note that “\” operator in Circom is

the integer division, while "*" operator is the multiplication modulo q . Suppose $q = 5$ and `constraint_assert_disabled` is turned on. Then, Tab. 7 shows the trace and constraint satisfaction sets. $S(C)$ does not contain $(x = 1, z = 0, y = 1)$ and $(x = 3, z = 1, y = 2)$ of $\mathcal{T}(\mathcal{P})$, meaning that this circuit is over-constrained ($\mathcal{T}(\mathcal{P}) \setminus S(C) \neq \emptyset$). Note that it is also under-constrained.

Zero-Division Pattern. Code 13 presents a real-world example of an under-constrained circuit from *circomlib* [29] that exhibits the zero-division vulnerability pattern. On line 13, a division is performed where the numerator is a degree-2 polynomial in `in[0]`, while the denominator is a degree-1 polynomial in `in[1]`.

```

1  template MontgomeryDouble() {
2    signal input in[2];
3    signal output out[2];
4
5    var a = 168700;
6    var d = 168696;
7    var A = (2 * (a + d)) / (a - d);
8    var B = 4 / (a - d);
9
10   signal lamda;
11   signal x1_2;
12
13   x1_2 <== in[0] * in[0];
14
15   lamda <-- (3*x1_2 + 2*A*in[0] + 1) / (2*B*in[1]);
16   lamda * (2*B*in[1]) === (3*x1_2 + 2*A*in[0] + 1);
17
18   out[0] <== B*lamda*lamda - A - 2*in[0];
19   out[1] <== lamda * (in[0] - out[0]) - in[1];
20 }

```

Code 13: Real example of Zero Division pattern in *circomlib* [29]. `lamda` can be any value when both the denominator and the numerator of Line 15 are 0, making this circuit non-deterministically under-constrained.

Hash-Check Pattern. Code 14 illustrates the real-world circuit implementing the hash-check pattern from the *maci* project [18]. Although this circuit is under-constrained due to insecure call of `LessThan` within `QuinTreeInclusionProof`, detecting this issue without the hash-check is difficult, as finding a valid input and Merkle root pair requires inverting the hash function.

```

1  template QuinLeafExists(levels){
2    var LEAVES_PER_NODE = 5;
3    var LEAVES_PER_PATH_LEVEL = LEAVES_PER_NODE - 1;
4    var i;
5    var j;
6
7    signal input leaf;
8    signal input path_elements[levels][LEAVES_PER_PATH_LEVEL];
9    signal input path_index[levels];
10   signal input root;
11
12   // Verify the Merkle path
13   component verifier = QuinTreeInclusionProof(levels);
14   verifier.leaf <== leaf;
15   for (i = 0; i < levels; i++) {
16     verifier.path_index[i] <== path_index[i];
17     for (j = 0; j < LEAVES_PER_PATH_LEVEL; j++) {
18       verifier.path_elements[i][j] <== path_elements[i][j];
19     }
20   }
21
22   root === verifier.root;
23 }

```

Code 14: Real example of Hash Check pattern from *maci* [18]. Determining a valid pair of input and Merkle root requires inverting the hash function. The circuit is under-constrained due to an unsafe invocation of `LessThan` within `QuinTreeInclusionProof`.

C Details of Benchmark Datasets

Fig. 6 presents a histogram of the number of instructions in the program \mathcal{P} after loop-unfolding, the constraint length $|C|$, and the number of weak assignments in the dataset. The x-axis is logarithmic. We observe that nearly 60% of the benchmarks contain weak assignments, which may lead to under-constrained bugs.

Fig. 7 also presents the top 10 most frequently used templates in our benchmark, all of which originate from *circomlib*[29]. The two most commonly used templates are `Num2Bits` and `IsZero`. Notably, `IsEqual` internally relies on `IsZero`, while `LessThan` builds upon `Num2Bits`. Templates ranked 5 through 9—namely `Sigma`, `MixLast`, `Ark`, and `MixS`—are mainly used within `Poseidon` [23], a widely adopted hash function in zero-knowledge (ZK) circuits.

D Additional Experiments and Analysis

D.1 RQ6: Generality of TCCT

We investigate how our TCCT framework is generalized compared to existing formulations.

Consider Code 15, which presents a lightly revised version of Code 4. Recall that `assert` does not add a constraint, leading to an under-constrained circuit. Line 4 and 9 are added by us to prevent any signal from being unused in the constraint, while the circuit is still under-constrained. Our TCCT can cover this issue since the constraints accept trace even if $b \neq h.y$, and `zkFuzz` successfully detects this bug. In contrast, none of the existing tools can detect this bug as the constraints are deterministic, and all signals are utilized in the constraints.

```

1  template Verify() {
2    signal input a;
3    signal input b;
4    signal output c;
5
6    component h = Hash();
7    h.x <== a;
8    assert(b==h.y);
9    c <== h.y * b;
10 }

```

Code 15: Example of an under-constrained circuit undetectable by other existing tools. Only our TCCT model is capable of capturing this bug, and `zkFuzz` successfully identifies it.

Another example is Code 16, a real-world circuit from the `iden3` protocol, where the `GreaterThan` internally uses `LessThan`, and the `IN` does not verify the bit-length of `GreaterThan`'s inputs. Consequently, `ZKAP` erroneously marks the circuit as unsafe. However, the first argument to `GreaterThan` is `count`—the sum of outputs from `IsEqual`. Since each `IsEqual` outputs either 0 or 1, the maximum value of `count` is bounded by `valueArraySize` (which is 3 in this case). As the second argument is 0, both inputs to `GreaterThan`

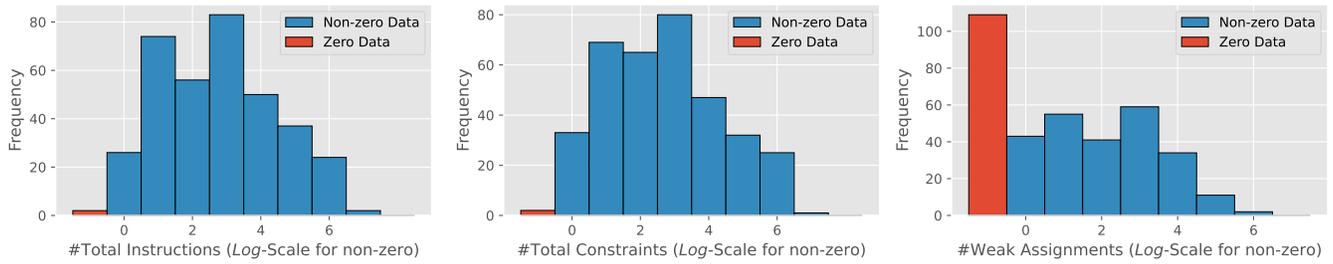


Figure 6: Distribution of circuit characteristics across 354 real-world ZKP benchmarks. The dataset includes circuits with large specifications, underscoring the need for scalable bug-detection tools. Notably, approximately 60% of circuits contain weak assignments, posing a potential risk of under-constrained vulnerabilities in real-world circuits.

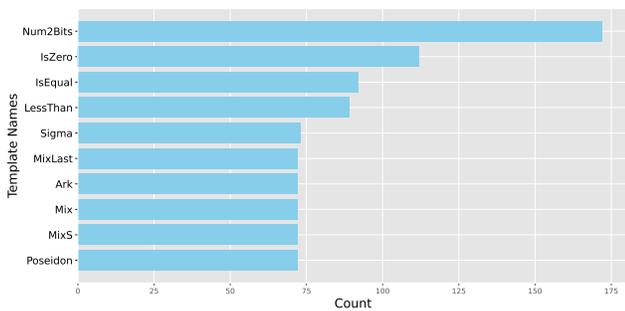


Figure 7: Top 10 Frequently Occurring Circomlib Templates.

remain sufficiently small, ensuring the circuit is safe. Notably, zkFuzz does not generate any false positive for this circuit.

```

1 template IN (valueArraySize){
2   signal input in;
3   signal input value[valueArraySize];
4   signal output out;
5
6   component eq[valueArraySize];
7   var count = 0;
8   for (var i=0; i<valueArraySize; i++) {
9     eq[i] = IsEqual();
10    eq[i].in[0] <== in;
11    eq[i].in[1] <== value[i];
12    count += eq[i].out;
13  }
14  // GreaterThan internally uses LessThan
15  component gt = GreaterThan(252);
16  gt.in[0] <== count;
17  gt.in[1] <== 0;
18  out <== gt.out;
19 }
20
21 component main = IN(3);

```

Code 16: Real-world circuit from iden3: This is safe despite misleading lack of bit-length checks

Result 6: Generality of TCCT

TCCT is more general than existing models, as it can cover bugs that others miss while avoiding false positives in real-world circuits.

D.2 RQ7: Alternative Mutation Strategies

In addition, we explore multiple strategies for input generation and program mutation. First, we apply a genetic algorithm to optimize input generation, using an error-based fitness score similar to the optimization in program mutation. Specifically, the fitness score of an input is defined as the minimum score across multiple program mutants. As shown in the left figure of Fig. 8, although performance eventually converges to the same level as the default setting, it remains worse throughout the process. This suggests that, due to the vast search space, guiding only program mutation with the fitness function, while feeding generated inputs, achieves a better balance between exploration and exploitation than guiding both program mutation and input generation with the fitness function, which skews more toward exploitation.

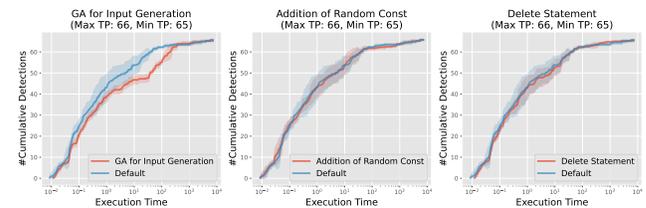


Figure 8: Various input generation and program mutation strategies.

We also investigate two additional program mutation strategies: (1) adding a random constant to the right-hand side of a weak assignment, such as transforming $z <- x$ into $z <- x + 1234$, and (2) removing the statement of the weak assignment entirely. In Circom, all signals are initialized to zero upon declaration, so eliminating a weak assignment ensures that the left-hand variable remains zero throughout execution. These new strategies are applied with a probability of 0.2. However, their performance remains nearly identical to the default setting.

Result 7: Alternative Mutation Strategies

Adding two additional mutation strategies, random constant addition and expression deletion, does not significantly improve the performance of zkFuzz, highlighting the effectiveness of its default mutation strategies.

Result 9: Over-Constrained Circuits

Since most circuits are designed for the default settings of the Circom compiler, enabling the `constraint_assert_disabled` flag causes nearly half of the circuits to be classified as over-constrained.

D.3 RQ8: Performance of Other Baselines

Tab. 8 presents the performance of the modified Circomspect, which recursively analyzes all internally called templates, and ZKAP with the Unconstrained-Signal detector. While recursive Circomspect detects all vulnerabilities, it also increases false positives, causing precision to drop sharply from 0.50 to 0.37. For ZKAP, the Unconstrained-Signal detector does not contribute to finding new bugs in our benchmarks.

Table 8: Performance of Circomspect with recursive analysis and ZKAP with the US and the USCO detector. Prec. stands for precision.

Constraint Size	Circomspect (Recursive)			ZKAP (with US and USCO)		
	TP	FP	Prec.	TP	FP	Prec.
Small	41	20	0.67	33	20	0.62
Medium	6	20	0.23	1	23	0.04
Large	16	27	0.37	3	5	0.38
Very Large	6	46	0.12	2	14	0.13
Total	69	113	0.38	39	62	0.39

Result 8: Performance of Other Baselines

The original ZKAP and Circomspect with recursive analysis detect more bugs, but at the cost of significantly increasing false positives and reducing precision.

D.4 RQ9: Over-Constrained Circuits

To validate zkFuzz’s ability to detect over-constrained circuits, we conduct experiments with the `constraint_assert_disabled` flag enabled, allowing over-constrained circuits to exist. In this mode, the equality constraint `===` adds a condition to the constraints set without inserting an `assert` into the program. Consequently, developers likely need to add the corresponding `assert` manually in the Circom file to prevent over-constrained circuits. Since most circuits are designed to function under the Circom compiler’s default configuration, enabling this flag renders many circuits over-constrained. In our benchmark, zkFuzz identifies 159 such cases out of 354 test cases. However, because these circuits typically run under the default compiler settings, such issues are unlikely to pose a security risk in practice.