Towards Fuzzing Zero-Knowledge Proof Circuits (Short Paper)

Stefanos Chaliasos
Imperial College London &
ZKSecurity
London, United Kingdom
stefanos@chaliasos.com

Imam Al-Fath
ZKSecurity
New York, USA
imam@zksecurity.xyz

Alastair F. Donaldson
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

Abstract

Zero-knowledge proofs (ZKPs) have evolved from a theoretical cryptographic concept into a powerful tool for implementing privacy-preserving and verifiable applications without requiring trust assumptions. Despite significant progress in the field, implementing and using ZKPs via *ZKP circuits* remains challenging, leading to numerous bugs that affect ZKP circuits in practice, and *fuzzing* remains largely unexplored as a method to detect bugs in ZKP circuits. We discuss the unique challenges of applying fuzzing to ZKP circuits, examine the oracle problem and its potential solutions, and propose techniques for input generation and test harness construction. We demonstrate that fuzzing can be effective in this domain by implementing a fuzzer for `zk-regex`, a cornerstone library in modern ZKP applications. In our case study, we discovered 13 new bugs that have been confirmed by the developers.

CCS Concepts

• **Security and privacy** → **Software security engineering**; **Formal methods and theory of security**.

Keywords

Fuzzing, Zero-Knowledge Proofs

1 Introduction

Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable a prover to convince a verifier that a given statement is true without revealing any additional information [Goldwasser et al. 1989]. They are widely used in applications ranging from privacy-preserving transactions and confidential voting to secure identity verification and blockchain scalability [Ernstberger et al. 2024b]. To implement a ZKP, the statement must be transformed into an arithmetic circuit or constraint system. During the past decade, various domain-specific languages (DSLs) have been developed to assist developers in writing ZKP circuits [Sheybani et al. 2025].

Despite their promise, ZKPs are challenging to implement correctly. Errors in circuit design and implementation can compromise

the security of a ZKP [Chaliasos et al. 2024a]. Such bugs may lead to issues with completeness (failure to produce proofs for valid statements), correctness (proofs that do not accurately reflect the intended computation), or soundness (dishonest provers generating invalid proofs that are nevertheless accepted by the verifier), thereby undermining system reliability. To address these challenges, existing solutions have primarily focused on formal verification [Coglio et al. 2023a; Isabel et al. 2024; Pailoor et al. 2023] and static analysis [Soureshjani et al. 2023; Wen et al. 2024] to check the correctness of ZKP circuits. While these methods have contributed significantly to identifying vulnerabilities, formal verification tools often struggle with scalability when applied to complex circuits, and static analysis focuses on specific patterns while facing challenges in minimizing false positives [Chaliasos and Yu 2025].

Given these limitations, fuzzing emerges as a promising complementary approach for testing ZKP circuits. Yet, applying fuzzing to ZKP circuits introduces unique challenges, such as designing ZKP-specific test oracles and generating appropriate witnesses, necessitating further research.

Contributions. In this paper, we make the following contributions:

- We present a comprehensive overview of the key challenges involved in fuzzing ZKP circuits.
- We sketch *initial solutions*, to be further explored by future work, to tackle the test oracle problem and the complexities of input (witness) generation for fuzzing ZKP circuits.
- We identify and discuss several open problems and future research directions in the domain of fuzzing ZKP circuits, highlighting opportunities for further enhancement.
- We demonstrate the practical impact of fuzzing ZKP circuits by implementing a simple yet powerful fuzzer for a state-of-the-art ZKP library, which uncovered 13 new bugs.

2 Background

A zero-knowledge proof (ZKP) is a cryptographic protocol by which a prover convinces a verifier that a given statement is true without revealing anything else beyond the truth of the statement [Goldwasser et al. 1989]. A valid ZKP must satisfy three properties: (i) *completeness*, if the statement is true and both parties are honest, the verifier will accept the proof; (ii) *soundness*, if the statement is false, no cheating prover can convince the verifier to accept the proof; and (iii) *zero-knowledge (ZK)*, the verifier learns nothing beyond the fact that the statement is true. In practice, many modern ZKPs are non-interactive and succinct. For example, zk-SNARKs (zero-knowledge succinct non-interactive arguments of knowledge) [Gennaro et al. 2013] allow very short proofs and fast verification, making them practical for applications like blockchain and privacy-preserving protocols [Ernstberger et al. 2024b]. These

```
template MontgomeryAdd() {
  signal input in1[2];
  signal input in2[2];
  signal output out[2];
  var A = ...;
  var B = ...;
  signal lamda;
  lamda <-- (in2[1] - in1[1]) / (in2[0] - in1[0]);
  lamda * (in2[0] - in1[0]) === (in2[1] - in1[1]);
  out[0] <== B * lambda * lambda - A - in1[0] - in2[0];
  out[1] <== lambda * (in1[0] - out[0]) - in1[1];
}
```

(a) Circuit

```
in1[0] = 1, in1[1] = 721428014810502002193220687200...
in2[0] = 4, in2[1] = 640610819890739540891436567510...
lambda  = 14322771264826975277158323431...
out[0]  = 27300585958536822892123621000...
out[1]  = 16504465218850074729376469432...
```

(b) Witness (field elements)

```
lamda * (in2[0] - in1[0]) - (in2[1] - in1[1]) = 0
out[0] - B*lambda*lambda - A - in1[0] - in2[0] = 0
out[1] - lamda * (in1[0] - out[0]) - in1[1] = 0
```

(c) Constraints

Figure 1: MontgomeryAdd circuit, with a witness sketch and the constraints generated from the circuit

```
template Multiplier {
  signal input a;
  signal input b;
  signal output c;
  c <-- a * b;
  c === a * b;
}
```

(a) Safe Example

```
template Multiplier {
  signal input a;
  signal input b;
  signal output c;
  c <-- a * b;
}
```

(b) Soundness Error

```
template Multiplier {
  signal input a;
  signal input b;
  signal output c;
  c <-- a + b;
  c === a * b;
}
```

(c) Completeness Error

```
template Multiplier {
  signal input a;
  signal input b;
  signal output c;
  c <-- a + b;
  c === a + b;
}
```

(d) Correctness Error

Figure 2: Four Circom multiplier examples, illustrating a correct circuit and various kinds of error

protocols require the statement to be expressed as an NP-hard problem (usually as an arithmetic circuit or constraint system) [Ernstberger et al. 2024a]. ZKPs are widely used for *private computations* (e.g., shielding transaction details in Zcash [Ben-Sasson et al. 2014]) and for *verifiable computations* (e.g., achieving blockchain scalability with validity rollups [Chaliasos et al. 2024b]). Note that in verifiable computation, the *ZK property is not required*. Instead, the key advantage is that a verifier can accept a succinct proof of a potentially large or expensive computation using significantly fewer resources than re-running the entire computation. For an in-depth introduction to ZKPs, we refer the reader to Thaler [2022].

ZKP Circuits. To obtain a ZKP of a statement, a developer must represent that statement in the form of a circuit. In particular, a *ZKP circuit* is an arithmetic representation of the computation, modeled as a collection of polynomial equations (constraints) over a finite field [Gennaro et al. 2013]. These constraints define precisely which values the circuit will accept.

A circuit C, defines a relation $R(I, W)$, where I is a public input provided by the verifier, and W is a witness computed by the prover that satisfies R. For instance, given a function, f over public input x, consider the problem of proving that $y = f(x)$ where y is claimed to be an output computed by f. The constraints of the circuit C are constructed so that they are only satisfied by valid tuples (x, y), i.e., by tuples where $y = f(x)$ actually holds. This allows the verifier to accept the proof. The witness W can be considered a trace of the computation under consideration, and constraints, i.e., the relation R, are assertions that enforce the correctness of the computation. In the provided example, the witness will contain both x, y.

Figure 1a contains the program for Montgomery curve addition in Circom (one of the most popular DSL for writing ZKP circuits).

Executing its assignment instructions (i.e., computation obtained by `<--` or `<==`) on two example curve points yields the field elements shown in Figure 1b, which compose *witness*. Each constraint in Figure 1c is obtained from an `===` or `<==` in Figure 1a. Substituting the numbers from Figure 1b into those three equations makes every left-hand side equal 0, hence the tuple $(\texttt{in1}, \texttt{in2}, \texttt{out})$ satisfies the circuit and can be turned into a valid proof.

In practice, developers use a DSL to write high-level code that describes which parts of the computation must be constrained and how. A compiler for the DSL then produces polynomial equations describing the constraints. Certain computations can be kept *out of the circuit*, with constraints merely validating that the external steps have been performed correctly. This approach can substantially reduce proof-generation costs. Ultimately, the circuit ensures that the verifier can check, with minimal overhead, that the prover indeed holds a valid witness for the asserted statement, all while preserving the zero-knowledge property.

Figure 2a shows a small circuit expressed in *Circom* [Bellés-Muñoz et al. 2023], a popular DSL for expressing ZK circuits (such languages are known as zkDSLs). Given this circuit, the Circom compiler produces both (i) a *witness generator*, which calculates c from a, b, i.e., produces the witness to be proved; and (ii) a *constraint system* ensuring any valid proof satisfies c = a × b. Then, a SNARK backend provides a *prover* (which uses the witness to produce a proof) and a *verifier* (which effectively checks a produced proof against the associated constraints and public inputs). Figure 3 depicts the pipeline for compiling a ZKP circuit and using it to generate and verify a proof.

Different Approaches to ZKP Development. There are three different ways that developers can use ZKPs. *(i) zkDSLs* such as
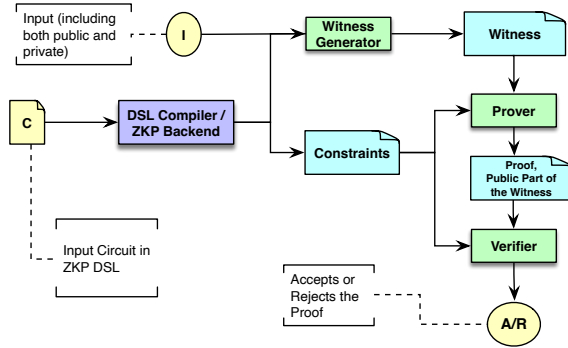
Figure 3: Simplified compilation and proof generation pipeline of ZKP circuits

Circom allow developers to write the circuit that implements their specifications directly. This offers tight control over efficiency but demands significant low-level expertise. *(ii) Domain-specific transpilers* allow developers to write higher-level statements (e.g., a regex in zk-regex [Gupta et al. 2022]) which are then transpiled into circuits (typically expressed in a zkDSL). While a good fit for specific tasks, this approach is not general-purpose and can introduce hidden inefficiencies and bugs if the transpiler is not correct. *(iii) ZK Virtual Machines (zkVMs)* like Cairo [Goldberg et al. 2021] or RISC0 [Bruestle et al. 2023] encode an entire CPU-like execution loop into some large circuits, allowing the execution of regular programs (e.g., in Rust) to be proved correct. This generality comes at the cost of performance. Notably, in all of these approaches, circuits remain a core component; what varies is whether they are written directly by a developer, or created by upstream tooling.

Common Vulnerabilities. Implementing ZKP circuits is error-prone, and various kinds of bugs can undermine the key ZKP properties of completeness, soundness and zero-knowledge.

A critical class of bugs are **soundness** issues that make a circuit *under-constrained*, such that witnesses are not fully constrained. This situation typically violates intended functionality, and allows a malicious prover to choose a bogus witness that satisfies all the given constraints but does not correspond to a trace that the circuit can actually produce. A simple example is shown in Figure 2b, a variant of Figure 2a where the result of the multiplication is unconstrained, so that the prover can convince the verifier that the circuit was used to determine bogus multiplicative relationships between numbers (e.g., that 2*5=100). Figure 1 shows a more realistic example.[1] MontgomeryAdd [iden3 2021] is used when adding two points on a Montgomery elliptic curve. Given two points (in1[0], in1[1]) and (in2[0], in2[1]) on the curve, a new point (out[0], out[1]) is calculated that effectively represents their sum under elliptic-curve arithmetic. Although the computation of the circuit of Figure 1a is correct, this circuit is under-constrained because an implicit assumption that $in2[0] - in1[0] \neq 0$. Thus, a malicious prover can convince a verifier that the addition of two points can be another value than the expected one because

of the freedom to control lambda. For example, consider the scenario where all inputs are 0, then the prover can pick any value for lambda and compute the output accordingly, regardless of whether the value constitutes a correct result from the circuit. Note that the verifier will accept the proof produced because the constraints will be satisfied.

In contrast, an *over-constrained* circuit contains extraneous constraints that make the constraint system too restrictive for some legitimate inputs. Such bugs lead to **completeness** issues. Figure 2c presents a variant of the multiplier circuit that suffers from a completeness bug. While the circuit works correctly when both *a* and *b* are 0 or 2, for other values, the constraints prevent the prover from producing a valid witness, resulting in failure to generate a proof.

Finally, **correctness** errors occur when the circuit fails to implement the intended behavior. For instance, as shown in Figure 2d, the circuit uses addition instead of multiplication to compute the product. Even if the witness satisfies the circuit's constraints, the output does not reflect the correct computation, causing either valid proofs to be rejected or invalid proofs to be accepted.

Table 1 outlines the different bug categories. Note that in contrast to prior work [Chaliasos et al. 2024a], we separate correctness and soundness issues, as they manifest differently.

3 Challenges, Solutions and Open Problems of Fuzzing ZKP Programs

Applying fuzzing techniques to ZKP circuits presents unique challenges arising from their specialized nature, including uncommon bug manifestations and the separation between constraints and computation. We now discuss the main challenges in fuzzing ZKP circuits and propose potential solutions.

3.1 Key Challenges

Test Oracle Problem. A critical challenge in fuzzing is the *test oracle problem*. In conventional software testing, a well-defined oracle indicates whether an output is correct for a given input [Barr et al. 2015]. For ZKPs, the objective is to define oracles that can detect completeness, correctness, and soundness bugs.
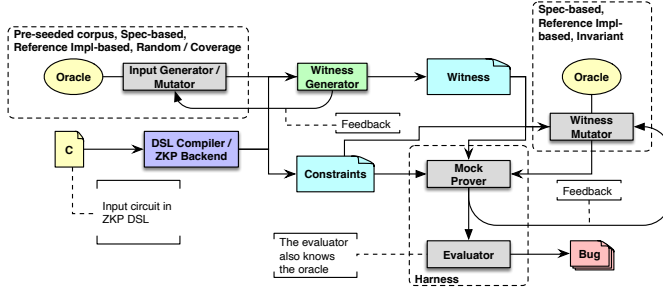
Completeness bugs are relatively straightforward to detect by verifying that valid inputs produce valid proofs, i.e., that unexpected failures are not observed. The completeness bug of Figure 2c could be detected by trying any inputs beyond 0 and 2. Identifying *correctness* bugs requires knowledge of the expected behavior, as they manifest when an honest prover outputs a proof that, while valid in the context of the circuit, conflicts with the expected behavior. Detecting the correctness bug of Figure 2d would require knowing that the circuit under test needs to perform multiplication instead of addition. *Soundness* bugs are harder to detect because they involve *invalid* proofs that the verifier erroneously accepts. Missing or insufficient constraints commonly cause such errors, as developers are working with low-level DSLs, a different programming model, and the need for optimizations. To detect the soundness bug of Figure 2b, one would need to generate a witness that satisfies the constraints but does not match the logic of the computation.

Input/Witness Generation. Input (or *witness*) generation aligns closely with oracle design. For completeness and correctness bugs,

---

[1]This example was extracted from the following report: https://veridise.com/wp-content/uploads/2023/02/VAR-circom-bigint.pdf.

Table 1: Overview of ZKP Bug Categories

Category	Manifestation	Detection Approach
Completeness	Valid inputs fail to produce valid proofs	Use known valid inputs from a spec or reference; if a proof is not generated, there is a bug
Correctness	Valid or invalid proof that contradicts intended behavior	Compare circuit outputs to expected outputs (via reference impl./formal spec)
Soundness	Dishonest prover can produce invalid proofs that the verifier accepts	Requires generating invalid witnesses that "should fail," but does not fail

Figure 4: Fuzzing ZKP circuits

Table 2: Comparison of oracle strategies; * **indicates that novel invariants can handle more bug types**

Oracle Strategy	Prerequisites	Bug Types Detected
Spec-Based	Specification	Completeness, Correctness, Soundness
Differential	Reference Implementation	Completeness, Correctness, Soundness
Invariants	None	Soundness*

3.2 Proposed Solutions

We discuss potential solutions to the challenges outlined in the previous section. Figure 4 outlines the primary components required to fuzz test ZKP circuits. The first step is to compile the circuit (i.e., the system under test) into a constraint system along with a witness generator. Next, an input generator produces candidate inputs based on an oracle (such as a specification), and these inputs are used to generate the corresponding witnesses. The witnesses are then processed by a mock prover, and an evaluator checks whether the results meet the oracle's expectations, thereby reporting any bugs. Furthermore, a witness mutator can further mutate these witnesses to produce variants that may reveal soundness issues. Finally, feedback from both the input generation/mutation and witness mutation stages is used to guide subsequent mutations.

Oracle Design Strategies. When a (machine-readable) specification or reference implementation is available, (in)correct-by-construction input generation or differential testing [McKeeman 1998] can resolve the oracle problem. If neither exists, one might leverage generic invariants (for instance, *deterministic outputs for the same inputs* [Pailoor et al. 2023]) to detect specific bug categories. Table 2 summarizes the bug types that each oracle can detect. Although common oracles such as crashes can be used, those would not be able to detect the majority of ZKP-specific bugs.

Input Generation for Completeness/Correctness vs. Soundness. Unlike completeness and correctness bugs, which can be caught by feeding valid or invalid inputs into a circuit and checking whether proofs and results match expectations, soundness bugs require crafting *witnesses* that the circuit ought to reject but instead (incorrectly) accepts. Randomized invalid witness generation is unlikely to be effective for non-trivial circuits because bugs typically occur in corner cases, e.g., in Figure 1a, the bug only manifests when `in2[0] = in1[0]`. That leaves the following options:

(i) Mutation From Valid Witnesses: A promising approach is to begin with a valid witness and then introduce slight modifications that make it invalid according to the oracle, thus exposing under-constrained soundness issues (Figure 5a). An additional improvement is to keep mutations local: if an error is introduced, adjust (i.e., patch) the remaining parts of the witness to ensure only some constraints are targeted. The intuition is that some constraints might be

producing valid inputs is relatively simple if a reference or specification exists. Generating invalid-yet-accepted witnesses (for soundness tests), however, is significantly harder: *(i)* randomly mutating a valid input often yields a proof rejected outright by the verifier; *(ii)* determining whether a mutated witness is *incorrect* yet still accepted requires advanced oracles, reference checks, or carefully designed invariants; and *(iii)* a small local mutation can invalidate other parts of the circuit unless it is carefully guided.

Efficiency and Performance Overheads. ZKP proof generation and verification entail computationally expensive operations. For example, proving could take up to hours for large real-world circuits. Fuzzing usually involves running large quantities of tests, but each proof can take significant time. This overhead renders standard fuzzing techniques infeasible in some cases.

Support for Multiple DSLs and Frameworks. ZKP circuits can be written in various DSLs (Circom, Noir, ZoKrates, Halo2, etc.), each mapping to potentially different proof systems. A fuzzing tool for one DSL might not port cleanly to another, leading to fragmented solutions and additional engineering challenges. Working on the arithmetization level (e.g., R1CS) might be possible, but critical information such as circuit separation is not preserved at this level. While not ZK-specific, this problem is magnified in the ZK space due to the numerous similar DSLs available to developers.

Working with Cryptographic Primitives. Many ZKP circuits integrate cryptographic operations such as hashes or signatures. Random mutations rarely reveal subtle flaws (e.g., partial collisions or bit misalignment). Moreover, cryptographic primitives often impose strict input conditions, such as specific preimages that should result to a hash after some operations, that are extremely unlikely to be met by randomly mutated inputs. This challenge is similar to issues encountered in fuzzing checksum-based systems [Wang et al. 2010], where valid inputs that satisfy the checksum constraints are rare, thereby hindering the exploration of critical circuit paths.

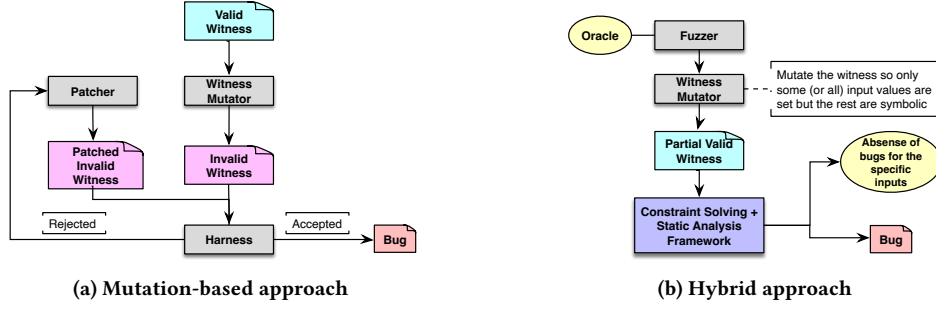(a) Mutation-based approach   (b) Hybrid approach

Figure 5: Workflow of a mutation-based approach for finding soundness bugs, and a hybrid approach that can be used to improve symbolic and static analyses to detect soundness bugs

missing or incorrect, while others may still be valid; if the mutated value conflicts with subsequent constraints and the witness is not patched, the mismatch will cause a failure there, obscuring the bug.

(ii) Hybrid Methods: Symbolic and static analysis has been used to detect soundness bugs in ZKP circuits [Pailoor et al. 2023; Wen et al. 2024], but suffers from scalability or precision issues. We envisage a hybrid approach where fuzzing is used to find *good inputs*, after which the other techniques to reason about the properties of a circuit based on these inputs; see Figure 5b. This will not provide any guarantees, but it could be a pragmatic approach to detecting bugs. The main intuition for this approach is that it will drastically reduce search space.

Optimizing Performance for ZKP Fuzzing. Because proof generation is expensive, avoiding cryptographic steps on every iteration is crucial. Pre-checking constraints in a 'mock' mode can save time if the DSL supports it (e.g., MockProver in Halo2). It is critical for DSLs to add support for mock proving to make fuzzing circuits possible. Additionally, sub-circuit fuzzing might be beneficial when having to fuzz test huge circuits. Yet, the oracle should be adapted because there might be a 'bug' (e.g., a missing constraint) in the sub-circuit, but other constraints in other circuits or the main circuit could include the missing check. Hence, when sub-circuits are targeted, the findings must be validated against the entire circuit to prevent false positives.

Cross-DSL Compatibility. A generic approach might unify circuits in a common intermediate representation (IR). However, any discrepancies between the DSL's compiler and the IR layer can introduce false positives or negatives (e.g., due to compiler bugs). Still, an IR-based framework potentially allows a single fuzzer to test multiple DSLs. A framework such as CirC [Ozdemir et al. 2022] could be helpful for this use case. Although this is not a ZK-specific issue, it is even more critical in the ZK space due to the fragmentation because of the different DSLs, which more or less allow the representation of similar programs and primitives.

Handling Cryptographic Primitives. A specialized oracle or white-box hooking of cryptographic components can help validate these sub-circuits. If the fuzzing target is a cryptographic function, domain-specific checks or differential testing against a reference implementation may suffice. The main difficulty is ensuring the fuzzer explores cryptographic logic deeply. For example, if a circuit takes as inputs a preimage of a hash and the hash value, we need to

guide the fuzzer on how to compute the hash for a random preimage to explore the whole circuit (given that it has more functionality).

3.3 Open Problems and Future Directions

Novel Invariants as Oracles. In Section 3.2 we discussed how an existing invariant [Pailoor et al. 2023] can be employed to detect soundness bugs with fuzzing. More complex invariants capturing deeper properties of ZKP circuits could enable the detection of additional classes of bugs in the absence of a full specification or reference implementation.

Practical Patching Mechanisms for Soundness Bugs. Although mutation-based approaches can expose under-constrained circuits, efficiently "patching" the rest of the witness to isolate targeted constraints remains complex. Implementations for dynamic witness adaptation could significantly improve the discovery of subtle soundness bugs.

Improved Coverage Metrics for ZKP Circuits. Traditional code-based coverage metrics do not directly apply to constraint-based logic. Developing customized coverage strategies (e.g., tracking how many constraints are actively 'exercised' in each test) may guide fuzzers to explore corner cases more systematically.

Combining Fuzzing with Static and Symbolic Analyses. Hybrid approaches that couple lightweight static checks or partial symbolic exploration with mutation-based fuzzing could amplify bug-finding capabilities. Integrating these methods seamlessly in the context of ZKP circuits remains an open challenge but offers high potential to bolster both coverage and depth of analysis.

In summary, fuzzing ZKP circuits is still in its infancy, leaving significant avenues for future research. Advancing this field is crucial to ensure that ZKPs remain secure and reliable as they continue to underpin an expanding range of modern applications.

4 Case Study: Fuzzing ZK-Email

Zk-email is a core library that provides an easy way to prove properties about emails within a circuit [Gupta et al. 2022]. One common use case is to prove ownership of an account (e.g., Twitter account) by demonstrating that a user received an email from Twitter containing their username. This is achieved by verifying the DKIM signatures and proving the inclusion of the username in the email body within a circuit. Zk-email relies on regular expressions for substring matching, and to facilitate this, it provides a transpiler
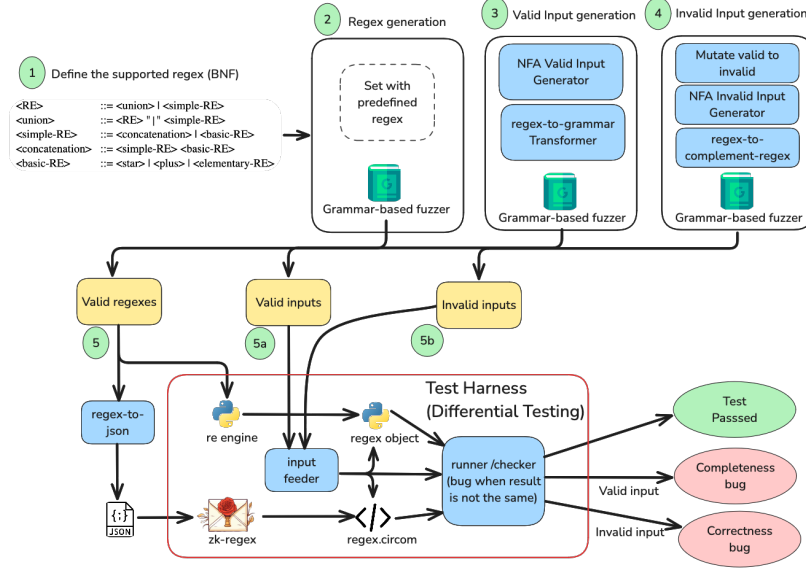
Figure 6: Fuzzing zk-regex

that converts a regex into a Circom or Noir circuit. As a core component used by many production-level ZK applications, the security of zk-regex is critical.

To fuzz-test the circuits produced by zk-regex, we address the test oracle problem by using a reference implementation of Python's regex module combined with differential testing. For input generation, we employ correct-by-construction and incorrect-by-construction inputs based on a specification of what constitutes a valid regex and the expected behavior of a given regex. Figure 6 depicts the end-to-end workflow of our fuzzer. We begin by writing a BNF specification of the fragment of regular-expression syntax supported by zk-regex. Then we employ a grammar-based fuzzer along with a pre-seeded corpus of important regexes, to get valid regexes. For each regex we then construct two types of test strings: *valid* inputs, which should be accepted, are obtained both by grammar expansion of the regex by employing a grammar-based fuzzer and by enumerating accepting paths in the corresponding NFA, whereas *invalid* inputs, which should be rejected, are generated through random mutation of valid inputs, exploration of non-accepting NFA branches, and generation from the grammar of the regex's complement. Next, each regex is wrapped in a JSON expected by zk-regex, compiled to a Circom circuit, and executed by using valid and invalid inputs. The harness performs differential testing between Python's re module and that of the generated circuit, and any disagreement is reported as a bug based on the used oracle that depends on if the input is valid or invalid.

Our campaign was conducted over one week, during which we fuzzed the library for one hour per day on a 16-core machine, continuously enhancing our fuzzer and incorporating fixes for some of the reported bugs in zk-regex. Our approach detected completeness and correctness bugs, although it did not uncover soundness bugs since we did not fuzz the witness. Despite the simplicity of our method, we revealed a total of 13 new bugs (5 correctness and 8

completeness), underscoring that even basic fuzzing campaigns can uncover significant issues in ZKP circuits. Notably, all 13 bugs were confirmed by the developers of zk-regex. We leave as future work the implementation and evaluation of the additional techniques described in Section 3.

5 Related Work

ZKP Security and Automated Analysis. Recent studies have categorized ZKP bugs and threats to different layers, highlighting a comprehensive taxonomy of vulnerabilities [Chaliasos et al. 2024a]. Several works have also focused on attacking ZKP constructions, demonstrating how oversights can lead to severe security vulnerabilities in all ZKP implementations that use the specific constructions [Ciobotaru et al. 2024; Dao et al. 2023; Khovratovich et al. 2025]. Another line of work has focused on rigorous verification of ZKP components, for instance, verifying the verifier itself to ensure correctness [Firsov and Livshits 2024]. While formal verification has been used to prove the absence of bugs in ZKP circuits [Coglio et al. 2023b; Isabel et al. 2024; Ozdemir et al. 2023; Pailoor et al. 2023], it often faces scalability challenges. Similarly, static analysis approaches [Soureshjani et al. 2023; Wen et al. 2024] can detect certain classes of issues but may suffer from precision limitations and heuristics that miss broader categories of bugs. These hurdles underscore the need for complementary, scalable, and precise techniques, such as fuzzing, that could reveal critical bugs.

Fuzzing ZKP Implementations. Fuzzing ZKP implementations has gained traction in a few domain-specific contexts. For instance, recent work identified completeness bugs in a specific zkEVM implementation by systematically mutating and monitoring execution paths [Li et al. 2024]. Further, metamorphic testing has been employed to uncover errors in ZKP compilers [Hochrainer et al. 2024; Xiao et al. 2025]. Finally, SnarkProbe [Fan et al. 2024] focuses on revealing bugs in ZKP DSL backends. These papers either focus

on non-circuit bugs or struggle with the challenges we outlined in Section 3, such as solving the oracle problem for soundness bugs.

6 Conclusion

We have discussed the unique challenges of fuzzing ZKP circuits, including issues with test oracles, input/witness generation, and performance overheads, and sketched practical solutions such as differential testing, invariant-based oracles, and hybrid methods to enhance bug detection. Our zk-regex case study demonstrates that even a simple fuzzing campaign can reveal significant bugs, highlighting the potential of fuzzing to improve ZKP security. The remaining open problems we have identified provide fruitful ideas for future fuzzing research in this domain.

Acknowledgments

References

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina Melé. 2023. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Trans. Dependable Secur. Comput.* 20, 6 (2023), 4733–4751. doi:10.1109/TDSC.2022.3232813

Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 459–474. doi:10.1109/SP.2014.36

Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. 2023. RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity. https://dev.risczero.com/proof-system-in-detail.pdf.

Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. 2024a. SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/chaliasos

Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. 2024b. Analyzing and Benchmarking ZK-Rollups. In *6th Conference on Advances in Financial Technologies, AFT 2024, September 23-25, 2024, Vienna, Austria (LIPIcs, Vol. 316)*, Rainer Böhme and Lucianna Kiffer (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:24. doi:10.4230/LIPICS.AFT.2024.6

Stefanos Chaliasos and Chenyang Yu. 2025. zkbugs: Repository of ZK security bugs and analyses. https://github.com/zksecurity/zkbugs. Accessed: 2025-03-18.

Oana Ciobotaru, Maxim Peter, and Vesselin Velichkov. 2024. The Last Challenge Attack: Exploiting a Vulnerable Implementation of the Fiat-Shamir Transform in a KZG-based SNARK. *IACR Cryptol. ePrint Arch.* (2024), 398. https://eprint.iacr.org/2024/398

Alessandro Coglio, Eric McCarthy, Eric Smith, Collin Chin, Pranav Gaddamadugu, and Michel Dellepere. 2023b. Compositional Formal Verification of Zero-Knowledge Circuits. *IACR Cryptol. ePrint Arch.* (2023). https://eprint.iacr.org/2023/1278

Alessandro Coglio, Eric McCarthy, and Eric Whitman Smith. 2023a. Formal Verification of Zero-Knowledge Circuits. 393 (2023), 94–112. doi:10.4204/EPTCS.393.9

Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. 2023. Weak fiat-shamir attacks on modern proof systems. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 199–216.

Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. 2024a. zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs. 14973 (2024), 46–72. doi:10.1007/978-3-031-71070-4_3

Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic, and Arthur Gervais. 2024b. Do You Need a Zero Knowledge Proof? *IACR Cryptol. ePrint Arch.* (2024), 50. https://eprint.iacr.org/2024/050

Yongming Fan, Yuquan Xu, and Christina Garman. 2024. SNARKProbe: An Automated Security Analysis Framework for zkSNARK Implementations. In *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14584)*, Christina Pöpper and Lejla Batina (Eds.). Springer, 340–372. doi:10.1007/978-3-031-54773-7_14

Denis Firsov and Benjamin Livshits. 2024. The Ouroboros of ZK: Why Verifying the Verifier Unlocks Longer-Term ZK Innovation. *IACR Cryptol. ePrint Arch.* (2024), 768. https://eprint.iacr.org/2024/768

Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, 626–645. doi:10.1007/978-3-642-38348-9_37

Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo - a Turing-complete STARK-friendly CPU architecture. *IACR Cryptol. ePrint Arch.* (2021), 1063. https://eprint.iacr.org/2021/1063

Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* 18, 1 (1989), 186–208. doi:10.1137/0218012

Aayush Gupta, Sampriti Panda, and Sora Suegami. 2022. zk-regex. https://github.com/zkemail/zk-regex/.

Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Fuzzing Processing Pipelines for Zero-Knowledge Circuits. *CoRR* abs/2411.02077 (2024). doi:10.48550/ARXIV.2411.02077 arXiv:2411.02077

iden3. 2021. Montgomery circuit. https://github.com/iden3/circomlib/blob/master/circuits/montgomery.circom.

Miguel Isabel, Clara Rodriguez-Nunez, and Albert Rubio. 2024. Scalable Verification of Zero-Knowledge Protocols . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1794–1812. doi:10.1109/SP54263.2024.00133

Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. 2025. How to Prove False Statements: Practical Attacks on Fiat-Shamir. *IACR Cryptol. ePrint Arch.* (2025), 118. https://eprint.iacr.org/2025/118

Zihao Li, Xinghao Peng, Zheyuan He, Xiapu Luo, and Ting Chen. 2024. fAmulet: Finding Finalization Failure Bugs in Polygon zkRollup. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 971–985. doi:10.1145/3658644.3690243

William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler infrastructure for proof systems, software verification, and more. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2248–2266. doi:10.1109/SP46214.2022.9833782

Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark W. Barrett. 2023. Bounded Verification for Finite-Field-Blasting - In a Compiler for Zero Knowledge Proofs. 13966 (2023), 154–175. doi:10.1007/978-3-031-37709-9_8

Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez-Núñez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. 2023. Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1510–1532. doi:10.1145/3591282

Nojan Sheybani, Anees Ahmed, Michel A. Kinsy, and Farinaz Koushanfar. 2025. Zero-Knowledge Proof Frameworks: A Systematic Survey. *CoRR* abs/2502.07063 (2025). doi:10.48550/ARXIV.2502.07063 arXiv:2502.07063

Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. 2023. Automated Analysis of Halo2 Circuits. 3429 (2023), 3–17. https://ceur-ws.org/Vol-3429/paper3.pdf

Justin Thaler. 2022. Proofs, Arguments, and Zero-Knowledge. *Found. Trends Priv. Secur.* 4, 2-4 (2022), 117–660. doi:10.1561/3300000030

Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 497–512. doi:10.1109/SP.2010.37

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. (2024). https://www.usenix.org/conference/usenixsecurity24/presentation/wen

Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. MTZK: Testing and Exploring Bugs in Zero-Knowledge (ZK) Compilers. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/mtzk-testing-and-exploring-bugs-in-zero-knowledge-zk-compilers/