Fuzzing Processing Pipelines for Zero-Knowledge Circuits

Christoph Hochrainer
TU Wien
Vienna, Austria
christoph.hochrainer@tuwien.ac.at

Anastasia Isychev
TU Wien
Vienna, Austria
anastasia.isychev@tuwien.ac.at

Valentin Wüstholz
Consensys
Vienna, Austria
valentin.wustholz@consensys.net

Maria Christakis
TU Wien
Vienna, Austria
maria.christakis@tuwien.ac.at

Abstract

Zero-knowledge (ZK) protocols have recently found numerous practical applications, such as in authentication, online-voting, and blockchain systems. These protocols are powered by highly complex pipelines that process deterministic programs, called circuits, written in one of many domain-specific programming languages, e.g., Circom, Noir, and others. Logic bugs in circuit-processing pipelines could have catastrophic consequences and cause significant financial and reputational damage. As an example, consider that a logic bug in a ZK pipeline could result in attackers stealing identities or assets. It is, therefore, critical to develop effective techniques for checking their correctness.

In this paper, we present the first systematic fuzzing technique for ZK pipelines, which uses metamorphic test oracles to detect critical logic bugs. We have implemented our technique in a tool called Circuzz. We used Circuzz to test four significantly different ZK pipelines and found a total of 16 logic bugs in all pipelines. Due to their critical nature, 15 of our bugs have already been fixed by the pipeline developers.

ACM Reference Format:
Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 20XX. Fuzzing Processing Pipelines for Zero-Knowledge Circuits. In *Proceedings of XYZ (XYZ 'XX)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Zero-knowledge (ZK) protocols have recently evolved to enable a wide range of practical scenarios and applications [17], including authentication, online voting, and blockchain systems. These protocols are called "zero knowledge" because they allow one party, the *prover*, to prove to another party, the *verifier*, that they know a secret without revealing it. More specifically, consider a deterministic program C, called a *circuit*, that performs a computation over public and private (or secret) inputs, I_P and I_S respectively. Given C, the prover must show to the verifier that the computed output O is indeed produced by executing C with I_P and I_S, without however revealing I_S.

Under the hood, the typical workflow of a ZK pipeline, shown in Fig. 1, is as follows. A circuit C is specified in one of many domain-specific programming languages, such as Circom [7], Noir [3], Corset [1], and others. The *compiler* of the ZK pipeline produces a constraint system as well as a *witness generator*. (Note that, for some pipelines, the witness generator already exists and is not generated by the compiler.) Given inputs I_P and I_S, the witness generator produces a *witness*, which is essentially an assignment satisfying the constraint system. On a high level, the witness may also be understood as a trace through C. The witness is used by the *prover* of the pipeline to generate a proof, which, together with I_P, can be passed to the *verifier* of the pipeline to verify its correctness.

Hence, unlike for regular programs, the pipelines for processing circuits comprise components for witness and proof generation as well as for proof verification. They are, therefore, highly complex, and given their growing applicability, correctness of these pipelines is highly critical [13]. More concretely, consider that the Gnark [2] and Corset ZK pipelines are core components of the Linea blockchain, which stores crypto-assets worth ca. 745M USD as of January 2025. Bugs in these pipelines could have catastrophic consequences, potentially causing significant financial and reputational damage. For this reason, pipeline developers typically follow very strict development processes. In the case of Gnark [2], both internal and external security teams perform regular audits; 8 external audits alone have been performed in the last two years (2022–24).

Still, bugs in ZK pipelines are extremely hard to detect just like for regular compilers and execution environments. There is, thus, a pressing need to develop automated and effective techniques for validating their correctness. *Verifying* the absence of bugs in their implementations is overly demanding. Consider, for instance, the verification efforts for CompCert [29], a compiler for a subset of C—it is about 15K lines of code and required 6 person years to write 100K lines of specifications. So, for the much more complex ZK pipelines, such an endeavor is practically infeasible.

Our approach. Unlike verification, automated test-generation techniques have been previously used to detect bugs in real-world compilers (see [14] for an overview) and program analyzers (e.g., [20, 22, 24, 31, 35, 44, 45]) without providing absolute correctness guarantees, that is, without promising that all bugs are found. In this paper, we present *the first fuzzing technique for finding critical*

XYZ 'XX, June 03–05, 20XX, Woodstock, NY
© 20XX Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

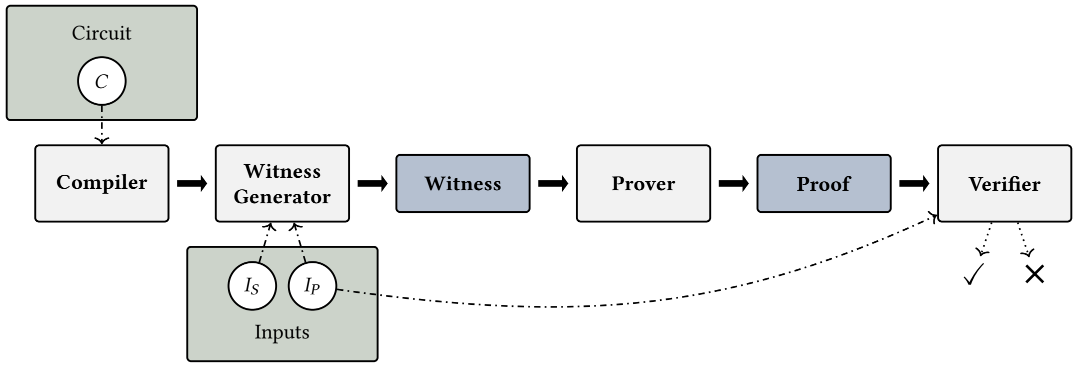


Figure 1: Overview of zero-knowledge pipeline stages.

logic bugs in circuit-processing pipelines and its implementation in a fuzzer called Circuzz.

To check the expected behavior of circuit-processing pipelines and effectively detect such bugs, automated testing techniques must solve the so-called *oracle problem* [6]. For example, differential testing [32] could address the oracle problem by running several pipelines on the same inputs and comparing their outputs for differences. More specifically, the inputs are circuits, and the outputs are the outputs of the various pipeline stages, e.g., compilation, witness generation, etc. In our context however, it would be difficult to ensure that the input circuits are semantically equivalent across diverse domain-specific languages with different levels of expressiveness.

Our approach, therefore, addresses the oracle problem using an alternative technique, called *metamorphic testing* [15]. Metamorphic testing [15] typically checks the correctness of a program P by running the program on an input i_1, observing the output o_1, and transforming i_1 to obtain i_2. This transformation is such that we know what output o_2 to anticipate (in practice, often the same as for i_1), thereby providing an *oracle* for the correct behavior of P. When actually running P on i_2, if o_2 contradicts the anticipated output, the oracle is violated and a bug in the program has been found. In our context, inputs i_1 and i_2 are circuits, the program is a ZK pipeline, and the outputs are the outputs of the various pipeline stages.

On a high level, Circuzz generates a configurable number of random circuits in an intermediate language, which we designed to capture essential features of many existing ZK languages, such as Circom or Noir. Next, Circuzz applies a sequence of metamorphic transformations to each generated circuit, say C_1, to obtain circuit C_2. After creating transformed circuit C_2, e.g., by swapping the arguments of a commutative operator in C_1, Circuzz translates C_1 and C_2 from the intermediate language to a specific ZK language. It then tests the entire processing pipeline, that is, including compilation, witness generation, proof generation, and proof verification.

To obtain (public and private) inputs I_P and I_S for the circuits, Circuzz generates them randomly. Hence, when executing the pipeline for each circuit using the same inputs, if any pipeline stage generates an unexpected output, a bug has been detected. For the example of swapping the arguments of a commutative operator, we would not expect any stage outputs to diverge.

Our approach constitutes the first application of metamorphic testing for circuit-processing pipelines. In addition, we observe that these pipelines are slower than most other fuzzing targets that have been tested using techniques such as metamorphic testing, e.g., parsers, compilers, or program analyzers. We, therefore, develop several optimizations to increase the unusually low test throughput when testing ZK pipelines.

In general, the bugs that Circuzz aims to detect are logic bugs. For ZK pipelines, such logic bugs can expose two types of critical issues, namely soundness and completeness issues. A soundness issue typically manifests through a lack of input validation in one of the pipeline stages; for instance, by finding a circuit and a circuit input for which the pipeline produces a witness or proof when, instead, the circuit input should be rejected. In contrast, a completeness issue typically manifests through overly strong input validation in one of the pipeline stages; for instance, by finding a circuit and a circuit input for which the pipeline does not produce a witness or proof when it should. These two types of issues can be seen as duals. However, note that certain logic bugs may compromise both soundness and completeness at the same time.

Our fuzzing technique primarily works on our intermediate language instead of specific ZK languages. Note that ZK languages can be quite different, e.g., Noir is Rust-based whereas Corset is Lisp-based. The generality that comes with the intermediate language allows us to easily extend Circuzz to test new ZK pipelines by adding the corresponding circuit-generation backends, which translate circuits from the intermediate to the target ZK language. In fact, we used Circuzz to test four significantly different ZK pipelines, namely Circom [7], Corset [1], Gnark [2], and Noir [3]. As we discuss in our experimental evaluation, Circuzz detected 16 logic bugs in these four ZK pipelines, 15 of which are already fixed by the developers. Three of the detected bugs compromised soundness, seven compromised completeness, and six compromised both soundness and completeness. Note that we responsibly disclosed all issues to the development teams by either privately reporting them or receiving their permission to publicly do so.

Contributions. Overall, our paper makes the following contributions:

- We present the first systematic fuzzing technique for circuit-processing pipelines; it uses metamorphic test oracles to find critical logic bugs.
- We implement our technique in the publicly available tool Circuzz[1].
- We evaluate Circuzz by testing four different ZK pipelines, namely Circom, Corset, Gnark, and Noir; Circuzz was able to detect logic bugs in all pipelines.

Outline. The rest of the paper is organized as follows. In Sect. 2, we give an overview of Circuzz, and in Sect. 3, we describe the technical details of our technique. Sect. 4 presents our experimental evaluation. We review related work in Sect. 5 and conclude in Sect. 6.

2 Overview

In this paper, we propose the first systematic technique for fuzzing circuit-processing pipelines. An overview of our technique is shown in Fig. 2. On a high level, it consists of the following steps: (1) circuit generation (in our intermediate language), (2) circuit transformation, (3) circuit translation (to the target language), (4) input generation, and (5) bug detection. In the rest of this section, we walk the reader through each of these steps based on an actual logic bug that Circuzz found in Circom. In Sect. 3, we describe each of these steps in detail.

Fig. 3a shows a circuit, say C_1, generated by step (1) of our technique. The circuit is expressed in our intermediate language, which we call CircIL; lines 1–2 declare the inputs and outputs, line 3 computes output `out0`, and line 4 introduces the constraint that `in0 != in1`. The output is assigned a constant expression, namely the bitwise complement of p. Here, p is a 254-bit prime number and the base field of the BN254 curve (also known as alt-BN128), a common prime elliptic curve used in cryptography. (We omit the value of the prime number in the code due to its size.)

Fig. 3b shows circuit C_2, again in our intermediate language, which is generated by step (2) of our technique, i.e., it is obtained by applying metamorphic transformations on C_1. In particular, we apply three *equivalence* transformations—that is, ones that do not alter the semantics of C_1—on line 3: a multiplication with the identity element `(1 * p)`, a subtraction of the identity element `((1 - 0) * p)`, and a division by the identity element `(((1 - 0) / 1) * p)`.

Step (3) translates C_1 and C_2 into the target language, in this case Circom. The resulting circuits are shown in Fig. 3c and 3d, respectively. Next, step (4) randomly generates input values for signals `in0` and `in1`. Finally, step (5) tests the entire Circom pipeline on C_1 and C_2 using the same input values for both circuits. Given that our metamorphic transformations are semantics preserving, if any pipeline-stage outputs diverge, a bug is detected. Specifically, Circuzz checks whether each stage of the Circom pipeline (i.e., compilation, witness generation, proof generation, and proof verification), if executed, succeeds or fails (due to the same reasons) for both circuits. For the witness-generation stage, Circuzz additionally checks whether the generated witnesses are equivalent. Note that this check is only executed if the generated input values satisfy the constraint on line 4.

It is the latter check (of the generated witnesses) that fails when running Circom on C_1 and C_2 from Fig. 3. In particular, Circuzz found that `out0` of C_1 evaluates to a very large number, whereas `out0` of C_2 evaluates to zero. Note that the constraint system generated by the compiler from a given circuit operates over a *finite field*; its size may be determined by an underlying elliptic curve, and in this case, Circuzz randomly selected the BN254 curve. In other words, all arithmetic operations are computed modulo prime p, thereby wrapping around this value. The difference in the outputs for C_1 and C_2 was caused because Circom did not apply the modulo operation to constants before they were used in an expression; this can result in computing different values in the presence of bitwise operations, such as the complement in our circuits. Consequently, any one of the above metamorphic transformations alone would have also revealed the bug.

This bug may compromise both the soundness and completeness of Circom. To demonstrate this, consider inserting the statement `assert(out0 == v)`, where `v` is the correct constant value of the bitwise complement, after line 7 of Fig. 3c. This assertion is expected to hold. However, for the buggy version of Circom, it fails, exposing a completeness issue. In contrast, when inserting the statement `assert(out0 != v)`, the assertion is expected to be violated. Yet, the pipeline stages succeed, exposing a soundness issue.

The developers fixed the issue by applying the modulo operation to all constants in the abstract syntax tree.

3 Approach

We now describe our technique for circuit-processing pipelines in detail. Before delving into each of its five steps, we provide an overview of our intermediate language for circuits.

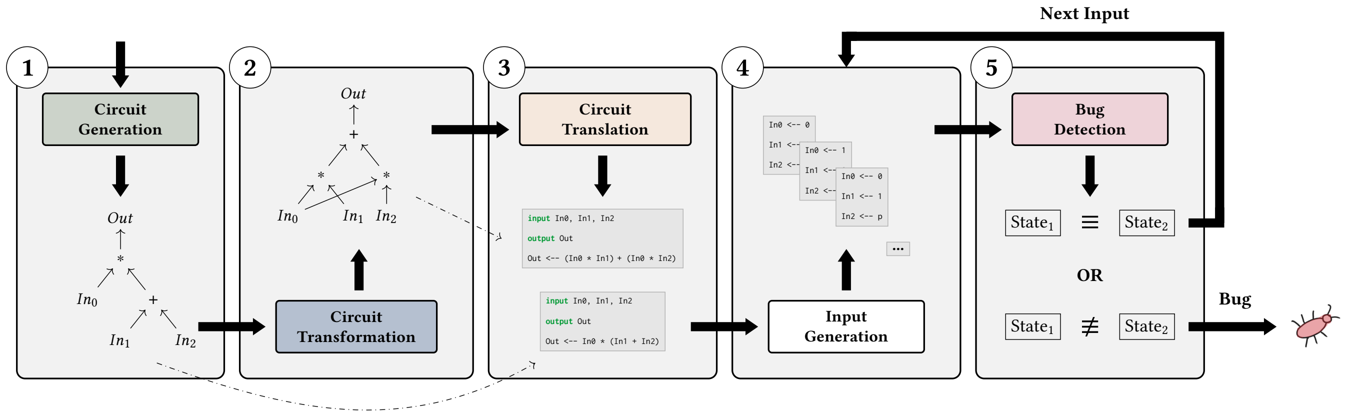
3.1 Circuit Intermediate Language

To easily support fuzzing of diverse circuit-processing pipelines, we generate and apply metamorphic transformations on circuits expressed in an intermediate language. This enables reusing the first two steps of our technique, namely circuit generation and transformation, for any new pipeline. In other words, to test a new pipeline, it is primarily the circuit-translation step that must be extended to translate to the corresponding ZK language. The bug-detection step also needs to be slightly adapted to execute the new pipeline.

On a high level, our intermediate language, CircIL, mainly captures a common subset of many existing ZK languages, such as Circom or Noir. These languages essentially provide syntactic sugar for expressing the underlying constraint system in a user-friendly way. For this reason, the simplest intermediate language could perhaps be one that directly defines a constraint system, such as R1CS (Rank-1 Constraint System). However, an intermediate language that is too simple would make it difficult to test certain features of high-level target languages, such as Circom, and their corresponding pipelines. We, therefore, designed a language that is as expressive as possible while still supporting features that can easily be translated to many popular ZK languages.

CircIL allows defining a circuit using three basic primitives: (1) a set of input variables (e.g., line 1 of Fig. 3a), (2) a set of output variables (e.g., line 2 of Fig. 3a), and (3) a sequence of statements (i.e.,

[1]https://github.com/Rigorous-Software-Engineering/circuzz



```
input In0, In1, In2
output Out
Out <-- (In0 * In1) + (In0 * In2)
```

```
input In0, In1, In2
output Out
Out <-- In0 * (In1 + In2)
```

```
In0 <-- 0
In1 <--
In2 <--
```

```
In0 <-- 1
In1 <--
In2 <--
```

```
In0 <-- 0
In1 <-- 1
In2 <-- p
```

Figure 2: Overview of our fuzzing technique for circuit-processing pipelines.

```
1 inputs : in0, in1
2 outputs: out0
3 out0 = (~ p)
4 assert(in0 != in1)
```

(a) Circuit C_1 in CircIL.

```
inputs : in0, in1
outputs: out0
out0 = (~ (((1 - 0) / 1) * p))
assert(in0 != in1)
```

(b) Circuit C_2 in CircIL.

```
pragma circom 2.0.6;

template main_template() {
  signal input in0, in1;
  signal output out0;
  out0 <-- (~ p);
  assert(in0 != in1);
}
component main = main_template();
```

(c) Circuit C_1 in Circom.

```
pragma circom 2.0.6;

template main_template() {
  signal input in0, in1;
  signal output out0;
  out0 <-- (~ (((1 - 0) / 1) * p));
  assert(in0 != in1);
}
component main = main_template();
```

(d) Circuit C_2 in Circom.

Figure 3: An example logic bug found by Circuzz in Circom.

the body), which expresses a set of constraints on the inputs and outputs (e.g., lines 3–4 of Fig. 3a). The body may contain two types of statements: (a) assignments to output variables, where the right-hand side is an expression over the underlying field (determined by an elliptic curve), such as line 3 of Fig. 3a, and (b) assertions of Boolean expressions (i.e., the field elements 0 and 1), such as line 4 of Fig. 3a. Expressions may be arbitrarily complex by using common operators, such as addition, multiplication, or equality, supported by the target ZK languages.

However, some target languages support additional operators. For instance, the bitwise complement is only used by Circom. As we saw in Fig. 3a, CircIL supports this operator; we provide a list of all CircIL operators below.

Unary operators: `-` (negation), `~` (bitwise complement), and `!` (Boolean not)

Binary operators: `+`, `-`, `*`, `/`, `%` (modulo), `**` (power), `&` (bitwise and), `|` (bitwise or), `^` (bitwise xor), `&&` (Boolean and), `||` (Boolean or), `^^` (Boolean xor), `==`, `!=`, `<`, `<=`, `>`, and `>=`

Ternary operators: `_ ? _ : _` (conditional)

To generate circuits using only operators of the target ZK language, Circuzz takes as input a language-specific configuration that enables an appropriate operator subset.

In general, it is easy to extend our intermediate language to support language-specific operators. Their translation to the target language is typically straightforward and enables generating more complex expressions. After including such operators in CircIL, we can also use them to define additional, language-specific metamorphic transformations.

3.2 Circuit Generation

The first step of our fuzzing technique randomly generates circuits based on the intermediate-language grammar.

The circuit-generation component of Circuzz is highly configurable allowing users to control the circuit size (by setting the maximum number of inputs and outputs, the maximum number of assertions, and the maximum depth of the generated expressions) as well as the circuit structure (by defining the allowed operators and setting custom weights determining how often a grammar rule should be applied). Based on the given configuration, Circuzz generates a random number of inputs, outputs, and assertions. It then generates random expressions to be assigned to outputs and used in assertions. The expressions may contain constants, inputs, outputs as well as allowed unary, binary, and ternary operators.

3.3 Circuit Transformation

The second step of our technique applies random metamorphic transformations to a circuit C_1 generated by the previous step. The transformations are designed such that the resulting circuit C_2 preserves the semantics of C_1. We can express such transformations using a set of rewrite rules that rewrite a circuit expressed in the intermediate language.

Rewrite rules. We developed a domain-specific language (DSL) for defining rewrite rules and used it to define a total of 87 rules (see Appx. A for the complete set of rules).

On a high level, the rewrite rules are based on pattern matching. Each rule is a triple, where the first element provides a unique rule identifier, the second a pattern to match in the intermediate language, and the third a rewrite template. For example, the following rule

```
{"one-plus-zero", "1", "(1 + 0)"}
```

is called `one-plus-zero` and replaces any occurrence of constant `1` by the expression `(1 + 0)`. We could also generalize the above rule as follows:

```
{"any-plus-zero", "?a", "(?a + 0)"}
```

`"?<NAME>"` (`?a` in the above rule) matches any expression and names it such that it may be referenced in both the match pattern and rewrite template. For example, the following rule

```
{"assoc-add", "((?a + ?b) + ?c)", "(?a + (?b + ?c))"}
```

employs the associative property of addition and would rewrite an expression `((in0 + 1) + in2)` to `(in0 + (1 + in2))`. Naturally, multiple occurrences of a given name are used to express structural equality, e.g., the match pattern `"?a | ?a"` would match

```
"(1 + 2) | (1 + 2)"
```

but not

```
"(1 + 2) | (2 + 1)".
```

Our DSL also allows matching expressions of a given type. Specifically, `"?<NAME>:<TYPE>"` matches any expression of a particular type, e.g., `"?a:bool"` would match `1` but not `42`. Recall that all expressions are elements of the field, and Boolean is a subtype consisting of elements 0 and 1. Therefore, we currently only need to match type `bool`, but in the future, our DSL could easily be extended to support more types if necessary. For example, the following rule

```
{"pow2-to-mul", "(?a ** 2)", "(?a * ?a)"}
```

rewrites an expression raised to the power of two to the expression multiplied by itself; no type specification is needed. The following rule, however,

```
{"double-lor-bool", "?a:bool", "(?a || ?a)"}
```

creates a logical disjunction between a Boolean expression and itself. Note that our intermediate language is untyped, and we perform Boolean-type inference for pattern matching with such rewrite rules.

Moreover, our DSL allows introducing new random expressions as part of the rewrite template (i.e., the third component of the triple). Specifically, `"$<NAME>:<TYPE>"` is used to generate a random constant of a given type in the rewrite template and name it. Again, the type specification is only necessary for generating random Booleans. For example, the following rule

```
{"sub-add-random-value", "?a", "((?a - $r) + $r)"}
```

first subtracts a random constant from an expression and then adds it again; no type specification is needed. The following rule, however,

```
{"double-lxor-bool", "0", "($r:bool ^^ $r:bool)"}
```

replaces constant `0` with the exclusive disjunction of a random Boolean and itself.

In general, using this domain-specific language, we have defined rules employing the identity, commutative, associative, and distributive properties of logical, bitwise, and arithmetic operators, De Morgan's laws, etc.

Stacked transformations. To increase the likelihood of finding logic bugs in the tested pipelines, Circuzz stacks the above circuit transformations. In other words, it may apply multiple rewrites to circuit C_1 to obtain C_2. This is possible since, in our case, all transformations have the same equivalence oracle, i.e., that no stage outputs should diverge.

3.4 Circuit Translation

Once we have two semantically equivalent, but syntactically different (after applying transformations), circuits C_1 and C_2, the next step translates these circuits from the intermediate language to the ZK language of the processing pipeline under test. Circuzz currently supports four diverse ZK languages, namely Circom, Corset, Gnark, and Noir. In particular, Circom is a low-level circuit language, which served as an inspiration when designing our intermediate language. One of its characteristics is that it is modular, thereby allowing users to define small, parameterizable circuits, called templates (see Fig. 3), that may then be combined to form larger circuits. On the other hand, Corset is Lisp-based, Gnark uses plain Go and provides a high-level API for writing circuits, and Noir is Rust-based.

This circuit-translation step of Circuzz is one of the two steps (besides the bug-detection step) that always needs to be extended when adding support for a new ZK language. The main task is to map all supported terminal and non-terminal symbols of the intermediate language to the corresponding symbols of the ZK language. For instance, for Gnark, we map `a + b` to `api.Add(a, b)` and `assert(a <= b)` to `api.AssertIsLessOrEqual(a, b)`. For certain target languages, such as Circom, this step is straightforward, but for other languages that are quite different from CircIL, such as Corset, the translation is more involved.

3.5 Input Generation

The input-generation step of Circuzz produces inputs (I_P and I_S) for circuits C_1 and C_2. We currently use blackbox fuzzing to randomly generate elements of the field. However, since the size of most supported fields is huge, we have introduced the ability to configure a set of constants that may be used during circuit generation as interesting boundary values. Examples of such constants are 0, 1, the size of the field p (typically a prime number), $p - 1$, etc. We randomly pick boundary values with a small probability (e.g., 5%).

Note that we do not know if the generated inputs should satisfy the constraints that are expressed using the generated circuit. As a result, some later pipeline stages may not always be executed since unsatisfiable inputs do not produce a witness that can be used to run the prover or verifier (see RQ3 in Sect. 4 for more details).

3.6 Bug Detection

As the final step of our technique, bug detection is specific to each processing pipeline under test and constitutes the other step of Circuzz that needs to be extended when adding support for a new ZK language. For instance, certain pipelines may combine the compilation and witness-generation stages or support several witness-generation and proving engines, so their execution requires configuration. Moreover, detecting bugs in each pipeline involves retrieving different artifacts, e.g., witnesses, error messages, return codes, etc., which are non-standard across pipelines.

Oracles. The bug-detection step executes the processing pipeline under test for both circuits C_1 and C_2 using the same inputs and reports a bug if our metamorphic oracle is violated, i.e., if any difference is detected between the two execution behaviors. For instance, a bug is detected if a witness is obtained only for one of the two semantically equivalent circuits. Circuzz typically executes each pipeline stage for both circuits and checks for bugs before moving on to the next stage. This allows to detect any differences in execution behavior as soon as possible without wasting time on later stages, such as proving or verification, which can be computationally expensive.

Besides the metamorphic oracles, Circuzz checks additional non-metamorphic, correctness properties per execution. For instance, a bug is reported if the witness generator generates a valid witness, but the prover fails to produce a valid proof. Similarly, we check that every valid proof can be successfully verified by the verifier. In future work, we plan to extend our oracles for the verification stage, for instance, by also checking if destructive transformations of generated proofs fail to be verified.

Additional metamorphic transformations. Optionally, the bug-detection step may introduce further metamorphic transformations, not on the input circuits, but on the pipeline settings. More specifically, we observed that many pipelines support settings that should not change their functional behavior. For instance, similar to many C compilers, Circom supports a setting for selecting the level of optimizations to be applied to the generated constraint system. Such settings can help in detecting additional bugs as follows. We randomly select a pipeline setting S_1 for processing C_1 and obtain a setting S_2 for processing C_2 by applying an equivalent metamorphic transformation on S_1; we expect our oracle to still hold. For example, in Circom, the `--O2` setting, which applies Gauss elimination to remove as many linear constraints as possible, could be transformed into the `--O0` setting, which disables all optimizations. These transformations of pipeline settings were especially effective for Corset, where Circuzz found 3 logic bugs due to such transformations.

3.7 Test-Throughput Optimizations

Certain pipeline stages are computationally expensive, for instance, proof generation. Consequently, when testing ZK pipelines, the test throughput is much lower (often seconds or even minutes per test case) than for many other fuzzing targets, such as parsers (often milliseconds per test case) or compilers (often less than a few seconds per test case). However, test throughput is an important aspect of effective fuzzers. For this reason, Circuzz implements optimizations to increase the unusually low throughput when testing ZK pipelines.

Power schedule. An obvious optimization is to skip the slower prover and verifier stages for some tests, but the interesting question is when to skip. We initially skipped slower stages randomly with a (fixed) high probability, but then refined our approach to more fairly test the initial (faster) stages (i.e., compilation and witness generation) and the subsequent (slower) stages (i.e., proof generation and verification). More specifically, Circuzz is given a target ratio ρ and, for each test, it only executes the later stages if $T_2/(T_1 + T_2) < \rho$, where T_1 is the total time we have already spent executing the initial stages and T_2 is the total time we have already spent executing the later stages. By default, we use $\rho = 0.5$ to roughly balance the time that is spent in the faster and slower stages. Over time, Circuzz dynamically adjusts when to skip the slower stages, thereby converging toward the desired target ratio.

This approach can be viewed as a novel "power schedule" [9] assigning energy to different pipeline stages. The power schedule directly exploits the ZK-pipeline structure and could be applied to other systems with a similar structure; for instance, compilers with very expensive optimization stages, or program verifiers with SMT-based verification stages.

Circuit size and complexity. We also observed that the size and complexity of the generated circuits has a tremendous effect on the performance of the pipeline stages. In particular, small circuits (e.g., circuits with few assertions, expressions of small depth, etc.) are preferable since they tend to be processed much faster. We hypothesize that most bugs can be detected with small circuits (see RQ3 and RQ4 in Sect. 4 for more details). Similar observations have been made in other domains; for instance, several concurrency testing tools, such as Cuzz [12], only explore very few context switches. For this reason, Circuzz is, by default, configured to produce smaller and less complex circuits (by controlling the number of assertions, expression depth, etc.).

Another interesting side effect of large and complex circuits, especially those with several assertions, is that they make input generation more challenging—many randomly generated inputs may not satisfy the assertions. This prevents the fuzzer from effectively testing later stages of a ZK pipeline, such as proof generation and verification. For less complex circuits, we can effectively test those stages with blackbox input generation (see RQ3 and RQ4 in Sect. 4 for more details). More complex circuits would probably require feedback-guided (i.e., greybox) or whitebox [21] input generation to satisfy their constraints. Unfortunately, whitebox fuzzing would likely further reduce the test throughput, especially due to overhead from constraint solving. In future work, we plan to explore alternative input generation techniques and further optimizations.

Circuit bundling. Finally, Circuzz also implements optimizations that are specific to certain pipelines. For instance, consider that for Gnark, each circuit needs to be compiled by the regular Go compiler, which is costlier than the compilation stage of other pipelines. To amortize this overhead, Circuzz may bundle multiple circuit pairs (i.e., a generated and a transformed circuit) that are then (batch-)processed by the pipeline. In other words, Circuzz can translate hundreds of circuit pairs into a single Go test file such that all circuits are compiled together. Since Go is able to execute individual tests (i.e., circuit pairs) in parallel, this bundling optimization has the additional benefit of parallelizing the bug-detection step in Circuzz for free.

4 Experimental Evaluation

We evaluate Circuzz by testing four ZK pipelines, namely Circom, Corset, Gnark, and Noir. In our evaluation, we address the following research questions:

RQ1: How effective is Circuzz in detecting logic bugs in diverse ZK pipelines?
RQ2: What are characteristics of the detected bugs?
RQ3: How efficient is Circuzz?
RQ4: How do the design choices and settings of Circuzz affect its effectiveness?

4.1 Zero-Knowledge Pipeline Selection

For evaluating the effectiveness and generality of our approach, we selected four popular, diverse, and maintained ZK pipelines. From a user perspective, they primarily differ in the ZK language for specifying circuits, ranging from functional to imperative. However, they also differ in many technical aspects of the processing stages, such as the supported constraint systems and cryptographic curves. Moreover, we chose actively maintained pipelines to ensure that the developers would respond to any reported bugs. We, therefore, required the latest activity in their repositories (i.e., commits and responses to open issues) to be within the last two months. Next, we provide a high-level overview of each tested pipeline.

Circom. At the time of writing, the Circom pipeline has 1.4K stars on GitHub and over 280 forks. It is, for example, used to implement the Tornado cash payment mixer (storing crypto-assets worth ca. 580M USD as of January 2025). The Circom language is imperative; it allows operations on constants, input, and output signals, all of which are field elements. Circom circuits are compiled to executable witness generators that, given a set of input signals, can compute output signals and generate witnesses for the prover and verifier of the pipeline.

Corset. The Corset language is functional and Lisp-like; it provides a limited set of operations and does not support output signals. Columns constitute the basic building block of Corset circuits and may be scalar or array-like; constraints are defined over columns. In addition to the four common stages, the Corset pipeline has an optional "check" stage that, given field-element assignments to columns, checks whether the corresponding constraint system is satisfied.

Gnark. At the time of writing, the Gnark pipeline has 1.5K stars on GitHub and over 400 forks. Like Corset, it is currently used to implement the Linea blockchain (storing crypto-assets worth ca. 745M USD as of January 2025). Circuits in Gnark can be specified as functions in the (general-purpose) Go language. Similar to Corset, Gnark does not support output signals; all signals are considered inputs, which are defined as structs over field elements. Moreover, the whole pipeline is embedded in Go, and each stage must be called using its API.

Noir. At the time of writing, the Noir pipeline has over 930 stars on GitHub and over 220 forks. It provides a strongly-typed, Rust-like language for specifying circuits. Similar to Circom, Noir supports explicit output signals that are computed and returned by the circuits. Unlike the other pipelines, it allows input values other than field elements. Since this is a unique feature of Noir, we have not yet added support for it in Circuzz. Structurally, the Noir pipeline differs from others by merging the compilation and witness-generation stages.

4.2 Experimental Setup

Testing time. We started testing Circom in March 2024, and we incrementally improved and extended our fuzzer to support more ZK pipelines. We subsequently added support for Gnark (in June 2024), Corset (in July 2024), and Noir (also in July 2024). As shown by this timeline, once Circuzz was mature enough, we were able to add new pipelines without too much effort.

Due to this timeline however, we did not spend the same amount of total fuzzing time on each pipeline. We estimate that we fuzzed Circom for ~5 months, Gnark for ~4 months, and Corset and Noir for ~3 months. Note that, once a bug was detected, we typically did not continue fuzzing the corresponding pipeline until the bug was fixed to avoid reporting duplicate issues.

For all pipelines, we tested either the latest stable release or the main development branch (to potentially find bugs that were introduced more recently).

Circuzz settings. Over time, we refined the default setup for Circuzz based on our experience. In particular, we identified the following key settings and default values: (1) the maximum number of inputs and outputs (each defaulting to 2), (2) the maximum number of assertions (defaulting to 2), (3) the maximum expression depth (defaulting to 4), and (4) the maximum number of stacked transformations that are applied to a generated circuit (defaulting to 64). The former three may affect the size and complexity of the circuits, and thus, the test throughput. The latter aims to strike a balance between finding bugs faster (by applying more transformations) and facilitating debugging (by not producing transformed circuits that differ too much from the generated ones). As discussed in Sect. 3.7, we use a default target ratio of $\rho = 0.5$ to roughly balance the time that is spent in the initial (typically faster) stages (i.e., compilation and witness generation) and the subsequent (typically slower) stages (i.e., proof generation and verification).

In RQ4, we compare different configurations of these settings in terms of their bug-finding effectiveness. To this end, we evaluate which configurations are able to refind bugs that we reported to the pipeline developers. To ensure that a bug detected by a given configuration indeed corresponds to the original, reported bug (and not to another one), we apply the fix that was provided by the developers and check whether the buggy behavior disappears. For this reason, we only use fixed bugs for evaluating the effectiveness of different Circuzz configurations.

Fuzzing campaigns. To ensure a fair comparison and mitigate the effects of randomness in the fuzzing process, we run 10 independent fuzzing campaigns for each Circuzz configuration. We limit the duration of each campaign to 24 hours. In general, we did not limit the time per pipeline execution. However, even though Corset was generally one of the fastest pipelines, we observed that it would occasionally (i.e., only for a few circuits) use over 1TB of memory and take several hours to run. For this reason, we introduced an upper bound of 8GB memory usage per pipeline execution (only for Corset).

Hardware. We performed all experiments on a machine with an AMD EPYC 9474F CPU @ 3.60GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm). To avoid issues due to hardware resources and obtain reproducible results, we restricted each fuzzing campaign to use a single logical CPU core.

4.3 Experimental Results

We now discuss our findings for each research question.

RQ1: Effectiveness of Circuzz. Tab. 1 shows all unique bugs found by Circuzz in the ZK pipelines we tested. The first column assigns an identifier (ID) to each bug and links to the bug report. We assign a number to fixed bugs and a letter to others. The second and third columns show the ZK pipeline in which the bug was found, and the bug status (i.e., reported, confirmed, or fixed). The fourth column indicates the type of logic bug that was exposed, i.e., whether it compromised soundness, completeness, or both. The fifth and sixth columns provide the pipeline stage where the bug was detected, and the oracle that detected it. Here, "MT" denotes a metamorphic oracle, and "VC" stands for validity check, i.e., a non-metamorphic, correctness property asserting the successful execution of a pipeline stage (see Sect. 3.6). The last column includes a short description of the bug.

In total, Circuzz detected 16 unique bugs, 15 of which were previously unknown. Bug 13 was found in the latest Noir release, but the developers had independently detected and fixed it in their development branch. Recall from Sect. 1 that ZK pipelines are regularly audited, and their developers follow strict procedures; yet, Circuzz was effective in detecting previously unknown, logic bugs. *13 of the bugs were detected due to violating a metamorphic oracle and 3 due to violating a validity check.* Of the 13 that were detected due to violating a metamorphic oracle, 10 involved metamorphic transformations on the circuits and 3 on the pipeline settings (see Sect. 3). The latter transformations (on the pipeline settings) uncovered bugs 5, 6, and 7 in Corset.

Circuzz also found several compiler crashes as a by-product, but we did not report most of them to focus on critical issues. For instance, when reporting bug 5, we discovered another, less severe bug, and developers opened an independent issue to track it. Crashes in the compiler are less critical since they would be found by the circuit developer and, therefore, cannot be exploited by attackers to hack end users of the circuit. On the other hand, crashes in later stages (e.g., proof generation or verification) are critical since they typically compromise completeness; for instance, a crash in Gnark may enable a DoS attack on the Linea blockchain.

15 bugs are already fixed by the pipeline developers, attesting to their critical nature. Addressing bug A is "*quite a challenge*" for the developers, which is why it has not yet been fixed. Even though most of our bugs were fixed quickly, some of them within hours of our report, they were often non-trivial to address. For instance, for bugs 5 and 10, the initial proposed fixes addressed the underlying problem only partly, and Circuzz quickly uncovered follow-up issues 6 and 11, respectively, which required additional changes in the code.

As shown in the table, 3 bugs were found in the compilation stage, 7 in the witness-generation stage, and 2 in the proof-generation stage. Circuzz found all 4 Corset bugs when executing the optional check stage, which checks the validity of the constraint system generated from a given circuit. Note that we run the Corset check stage before the compilation stage. We find it encouraging that Circuzz found most issues in early pipeline stages, which is likely due to the fact that the proof-generation and verification stages are audited even more thoroughly. Additionally, since these stages are significantly more computationally expensive than others, we ran them less frequently (according to our target ratio ρ). While we can configure Circuzz to execute the full pipeline more often, that would significantly decrease test throughput (see RQ4).

The feedback from the pipeline developers was overwhelmingly positive. For instance, one of the main developers of Gnark responded with "*that fuzzer is killing it!*" when we reported bug 12. In response to bug 8, Corset developers called it a "*critical bug actually. Good spotting!*". It turned out that a feature to support word-wise normalization was only partially implemented, but it was used in the standard library. Overall, all teams strongly encouraged us to keep fuzzing their code.

RQ2: Detected logic bugs. In the following, we provide a more detailed description of bugs found by Circuzz in each of the tested pipelines. Note that, for simplicity, we show manually minimized versions of the generated and transformed circuits. In practice, we also manually minimized the circuits that we included in our bug reports. This tends to make it much easier for developers to debug and fix the issues.

Fig. 4a shows two circuits that revealed bug 2. Circuzz generated circuit `c1` (top) that computes the bitwise complement of 0. It then transformed `c1` into the equivalent circuit `c2` (bottom) by replacing `0` with `0 ^ 0`. Circom only allows quadratic constraints, which `~ (0 ^ 0)` is not, therefore Circuzz (internally) rewrites this expression into the intermediate assignments on lines 8 and 9. When executed, the two circuits computed different values for

Table 1: Unique logic bugs detected by Circuzz.

Bug ID	Pipeline	Status	Type	Stage	Oracle	Description
A	Circom	confirmed	Completeness	Prover	VC	“Polynomial is not divisible” error
1	Circom	fixed	Both	Witness	MT	Wrong bit-width used for constant evaluation
2	Circom	fixed	Both	Witness	MT	Incorrect evaluation of bitwise complement of zero
3	Circom	fixed	Both	Witness	MT	Inconsistent evaluation of field prime
4	Circom	fixed	Both	Witness	MT	Inconsistent evaluation of a small field prime
5	Corset	fixed	Soundness	Check	MT	Inconsistent behavior of expansion and native flags
6	Corset	fixed	Soundness	Check	MT	Incorrect evaluation of constraints using expansion
7	Corset	fixed	Soundness	Check	MT	Incorrect expansion transformation of conditionals
8	Corset	fixed	Both	Check	MT	Incorrect evaluation of normalized loobean
9	Gnark	fixed	Both	Witness	MT	Inconsistent evaluation of $\vee$ for constants and signals
10	Gnark	fixed	Completeness	Witness	MT	Incorrect evaluation of `AssertIsLessOrEqual`
11	Gnark	fixed	Completeness	Compiler	MT	Zero bit length for binary decomposition on constants
12	Gnark	fixed	Completeness	Compiler	MT	Compiler panic on branch with unchecked cast
13	Noir	fixed	Completeness	Witness	MT	Incorrect evaluation of asserted condition
14	Noir	fixed	Completeness	Prover	VC	Proof failure due to insufficiently large string
15	Noir	fixed	Completeness	Compiler	VC	Stack overflow for $<$ with nested expressions

```
template C1() {
  signal a;
  a <-- (~ 0);
}

template C2() {
  signal a, tmp, zero;
  tmp <-- (0 ^ 0);
  zero <-- (~ tmp);
  a <== zero;
}
```

(a) Bug 2 in Circom.

```
1  (defcolumns in0)
2  (defconstraint C1 ()
3    (vanishes! (let ((out0 in0))
4      (let ((out1 0))
5        (eq!
6          (is-not-zero!
7            (eq!
8              (if (is-zero out1) out1 1)
9              (neq! in0 in0)))
10         (~or! 0 out0))))))
```

(b) Bug 6 in Corset.

```
func (circuit *C1) Define(api frontend.API
                         ) error {
  api.AssertIsLessOrEqual(1, 0)
  return nil
}

func (circuit *C2) Define(api frontend.API
                         ) error {
  api.AssertIsLessOrEqual(1, api.Or(0, 0))
  return nil
}
```

(c) Bug 10 in Gnark.

```
fn main(input : Field) -> pub Field {
  let b2 : [u8; 32] = input.to_be_bytes();
  let b2_f =
    std::field::bytes32_to_field(b2);
  assert(0 != b2_f, "Assertion violated");
  0
}
```

(d) Bug 14 in Noir.

Figure 4: Critical bugs detected by Circuzz in Circom (top left), Corset (top right), Gnark (bottom left), and Noir (bottom right), and fixed by the developers.

signal `a`. The discrepancy came from a difference in the sign of 0 and `zero`: constant 0 was considered negative, while signal `zero` positive, resulting in different bitwise complements. The developers fixed the issue by enforcing a positive sign on all zero operands of the bitwise complement. Interestingly, bug 3 presented earlier (see Fig. 3) was discovered by a syntactically similar pair of circuits but uncovered a distinct issue, as we explain in Sect. 2.

Fig. 4b shows bug 6 found in Corset; there is no need to understand the functionality of the code other than observe that there is an if-condition nested within an expression on line 8. This issue was discovered by running Corset on a single circuit, namely `C1` from Fig. 4b, but with different flags. Circuzz first ran the pipeline with the `-N` flag, which enables native mode. By default, all operations are performed using `BigInt` objects, and native mode uses (mathematical) field values instead. Circuzz then transformed this setting into the `-Ne` flag. The `e` part of the flag enables expansion mode, which rewrites constraint expressions into a lower-level, but equivalent, form. Even though flag `-e` should not affect the constraint satisfiability, the constraints were found SAT in native mode but UNSAT when enabling both the native and expansion modes.

After closer inspection, the developers responded that "*there is a problem with the handling of if-conditions when they are nested within certain expressions*". The problematic part of the code, which was only triggered when enabling expansion mode, intended to hoist the nested if-conditions into separate constraints. The proposed fix changed the order of cases when pattern matching a nested if-condition. However, when testing the fixed version, Circuzz revealed that, even though the fix worked for the provided circuit, there were still cases where Corset did not treat if-conditions correctly. Based on this finding, we reported bug 7. The developers concluded that "*the related issue is in the same (source-code) file as the original problem, but in a different method*". To fix the issue, they had to rework the handling of nested if-conditions (in expansion mode) from scratch.

Bug 10 refers to an issue discovered by Circuzz in the evaluation of the `AssertIsLessOrEqual` primitive in Gnark. Consider circuit `C1` shown in Fig. 4c. For `C1`, Gnark failed to generate a witness since the assertion does not hold. Then, Circuzz applied the following rewrite rule

```
{"zero-or", "?a", "(?a | 0)"}
```

that transformed `0` (line 3) into the equivalent `api.Or(0, 0)` (line 9). Unlike for `C1`, Gnark successfully generated a witness for the transformed circuit `C2`. The Gnark developers modified the assertion API code to correctly handle the special case where the first argument of `AssertIsLessOrEqual` is a constant.

This fix, however, overlooked another corner case where the first constant in the assertion is zero. For instance, when replacing `1` by `0`, the proposed fix did not work, and the metamorphic oracle still failed. Circuzz found this corner case in a subsequent fuzzing campaign (bug 11). The final fix removed the code that was trying to optimize the evaluation of such assertions for constant arguments. The two iterations that were required to fix the root issue suggest that, despite thorough testing on the developer side (including unit and regression tests), no test was able to catch neither the initial issue (bug 10), nor the issue that remained after the partial fix (bug 11). This provides a glimpse into the complexity that developers of ZK pipelines are facing and highlights the need for automated test generation.

Fig. 4d shows the circuit that revealed bug 14 in the Noir prover. Before generating a proof, Noir creates a structured reference string (SRS), which records proving and verification parameters as well as a sequence of samples from some complex (secret) distribution. This string is later used to verify the correctness of the proof. The number of required samples in the SRS depends on the circuit and is estimated automatically. When proving our example circuit, the Noir pipeline crashed because the (automatically) estimated number of samples in the SRS was too small. This bug was revealed as a violation of our validity check that expects a successful witness generation to be followed by a successful proof. Developers fixed the issue by changing the algorithm to over-approximate the number of necessary samples.

Overall, the presented bugs demonstrate that Circuzz is able to identify a diverse—for instance, with respect to the different ZK pipelines, oracles, and affected pipeline stages—set of critical bugs that developers are eager to fix.

RQ3: Efficiency of Circuzz. We primarily evaluate the efficiency of Circuzz in terms of its *bug-finding time*. We additionally measure the number of circuits that had to be generated to find a given bug. We track these metrics for each *fixed* issue discovered by Circuzz (listed in Tab. 1). We only use fixed issues for this evaluation since the difference in behavior of buggy and fixed code provides a reliable way to identify if a given bug was indeed detected by the fuzzer.

Tab. 2 summarizes the results of this experiment across 10 independent fuzzing campaigns, i.e., using 10 different random seeds to randomize the fuzzing process, each with a time limit of 24 hours. We use the default configuration of Circuzz as described in Sect. 4.2, i.e., up to 2 inputs, outputs, and assertions per circuit, expressions of depth up to 4, and up to 64 stacked transformations. The first two columns of the table show the ZK pipeline under test and the unique bug IDs (from Tab. 1), and the third column presents the percentage of generated circuit inputs that satisfy the corresponding constraint system. The remaining columns show the minimum, median, and maximum bug-finding times across all campaigns as well as the minimum, median, and maximum number of circuits that were generated until each bug was found.

Circuzz reliably detects all issues in all 10 campaigns, and the median bug-finding time for 13 (out of 15) bugs is less than 1 hour. The median number of generated circuits that are needed to detect these 13 bugs is less than 850.

We observe that the bug-finding time varies greatly across different pipelines. This is expected since pipelines differ structurally, in the way they are executed as well as in the programming language in which they are implemented. The median time for a single pipeline run is 0.5s for Circom, 0.1s for Corset, 3s for Gnark, and 6s for Noir. Thus, even for the same number of generated circuits, Gnark and Noir are expected to have higher bug-finding times. For instance, Gnark is essentially a library, and a pipeline run is a sequence of API calls that need to be compiled before the circuit compilation; similarly, a Noir pipeline run performs additional analysis of circuits and executes a virtual machine. Of course, there are also differences across pipelines in how much time is spent in each stage. This affects how often the faster and slower stages are executed, which in turn impacts the efficiency of Circuzz.

It is also worth noting that *all bugs are detected using the default configuration of Circuzz, which generates small and relatively simple circuits.* Consequently, the percentage of SAT circuit inputs (shown in the fourth column of Tab. 2) is always greater than 52% despite the fact that inputs are generated using blackbox fuzzing. Such a high percentage is important for the fuzzer's effectiveness since UNSAT circuit inputs typically cannot exercise later stages of the pipeline, like proof generation and verification. A significantly smaller percentage would, therefore, introduce unwanted bias towards the earlier pipeline stages.

RQ4: Design choices and settings. In this research question, we evaluate the effectiveness of several design choices and settings in Circuzz. Specifically, we consider five variants of the default configuration, each of which modifies a single setting: (1) maximum

Table 2: Time and number of generated circuits that the default configuration of Circuzz needed to find a fixed issue across 10 independent fuzzing campaigns, each with a time limit of 24 hours.

ZK Pipeline	Bug ID	SAT inputs	Time to bug min	Time to bug med	Time to bug max	Circuits to bug min	Circuits to bug med	Circuits to bug max
Circom	1	52.88%	38s	4m14s	13m47s	7	61	212
	2	57.84%	13m08s	31m00s	1h00m30s	232	567	1265
	3	57.38%	43s	13m30s	35m17s	19	210	717
	4	57.53%	18s	7m19s	16m13s	8	108	247
Corset	5	65.08%	1s	17s	2m24s	3	11	66
	6	64.21%	<1s	20s	2m45s	5	17	117
	7	61.17%	1s	3m35s	12m11s	5	91	340
	8	63.26%	30s	3m26s	11m56s	19	112	377
Gnark	9	57.05%	1m54s	11m28s	25m12s	10	122	372
	10	55.97%	1m13s	13m13s	44m21s	9	141	663
	11	56.94%	2m01s	13m57s	39m17s	21	162	634
	12	55.68%	30m20s	2h26m54s	20h21m27s	446	2361	15329
Noir	13	59.31%	29m45s	57m12s	5h07m37s	399	843	4963
	14	58.88%	1h55m37s	10h17m13s	16h38m51s	488	2908	4780
	15	59.05%	8m19s	48m40s	1h49m52s	25	184	450

number of *inputs* (changed from 2 to 8), (2) maximum number of *outputs* (changed from 2 to 8), (3) maximum number of *assertions* (changed from 2 to 8), (4) maximum *expression depth* (changed from 4 to 16), and (5) target ratio ρ (changed from 0.5 to 1.0 to disable the optimization). For this evaluation, we again run 10 independent fuzzing campaigns for each of the five configuration variants (with the same random seeds as for the default configuration).

We focus our comparison with the default configuration (see Tab. 2) on the following metrics: (1) bug-finding time, (2) test throughput (i.e., number of generated circuits per second), and (3) percentage of SAT inputs. For each metric, we first compute the median ratio of the variant over the default configuration for each of the fixed bugs (excluding timeouts), and then calculate the geometric mean across all bugs. The results are summarized in Fig. 5.

Unsurprisingly, configurations that generate larger and more constrained circuits—with *increased expression depth and number of assertions*—take 6.9x and 2.3x longer to find the given bugs, and there is a significant reduction in test throughput (0.4x and 0.8x) in comparison to the default configuration. Moreover, the variant with increased expression depth timed out for 7 out of 10 seeds for bug 12, and for 2 seeds for bug 14. The variant with an increased number of assertions timed out for 1 seed for bug 14. The only outliers in terms of bug-finding time are Noir’s bugs 13 and 15. Both require several nested expressions to be detected. As a result, they are found faster when increasing the expression depth.

With the default configuration, ~59% of all circuits with concrete inputs were SAT (see Tab. 2). Larger and more constrained circuits predictably result in fewer SAT inputs: when increasing the expression depth, only ~42% of all generated inputs were SAT, and when increasing the number of assertions, only ~26% of all inputs satisfied the corresponding constraint systems.

Increasing the number of circuit inputs does not have a drastic effect on any of our metrics. This is expected since our circuit-generation step does not force all inputs to be used. Therefore,

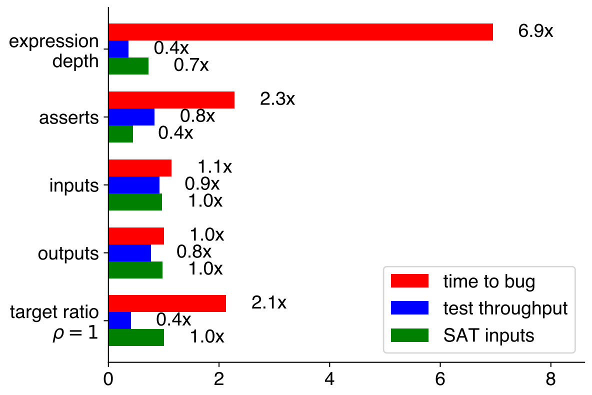


Figure 5: Comparison of the Circuzz default configuration with five variants. The bars show the relative change with respect to three metrics: (1) bug-finding time, (2) test throughput, and (3) the percentage of SAT inputs.

simply adding more signals does not necessarily make the generated circuits more complex.

Increasing the number of circuit outputs does not have a significant effect on the bug-finding time. However, it slightly reduces the test throughput. Upon closer inspection, we observed that the bug-finding time increased by 2.9x for Corset, but decreased for other pipelines (0.5x for Circom, 0.9x for Gnark, and 0.7x for Noir). Recall that Corset does not directly support outputs, and our translation simply introduces additional temporary variables. As a result, increasing the number of outputs increases the complexity of Corset circuits. For other pipelines however, the additional complexity is offset by making the oracle more effective—more output values are included in each generated witness, and thus, more data can be compared between the two witnesses of semantically equivalent circuits.

After considering the four variants that change settings in the circuit generator, let us now consider the final variant that *increases the target ratio ρ from 0.5 to 1.0*, thereby disabling the optimization that frequently skips the later, slower stages of a pipeline. In other words, this variant always runs the entire pipeline, whereas the default configuration (with the optimization) tries to balance the time that is spent in the earlier and later pipeline stages.

With the optimization, we would expect that only a small portion of all generated circuits execute the entire pipeline. For three pipelines, this expectation is also confirmed experimentally. For instance, Gnark's prover is significantly slower than the rest of the pipeline, and only 0.7% of all generated circuits execute the entire pipeline. Circom and Noir's provers are also slow, and only 3% (for Circom) and 10% (for Noir) of all circuits execute the entire pipeline. In contrast, for Corset, the full pipeline is executed for 35% of all circuits. This higher percentage results from the fact that we include the optional check stage (checking the validity of the constraint system that is generated from a given circuit) in the earlier pipeline stages, thereby making them more computationally expensive. To compensate, the later stages run more often for Corset than for other pipelines.

When disabling the optimization, we observe higher bug-finding times. The number of timeouts also increased without the optimization: for bug 12 (6 out of 10 seeds timed out), bug 13 (1 out of 10), and bug 14 (2 out of 10). This is not surprising since the test throughput dropped to 0.4x in comparison to the default configuration.

As these results show, the default configuration provides a good trade-off for effectively finding bugs across these different pipelines.

4.4 Threats to Validity

Our experimental results depend on the ZK pipelines under test, their settings (e.g., the used curve or optimization level), and the settings of Circuzz. Moreover, fuzzing per se is a random process, and Circuzz randomly generates circuits, transformations, etc. To address these potential threats, we selected four ZK pipelines that differ in their circuit programming language, their architecture, and backend components. In addition, we randomized the settings of the pipelines (via metamorphic transformations) and systematically varied settings of Circuzz during our evaluation. To mitigate effects of randomness on fuzzing, we ran 10 independent fuzzing campaigns for each evaluated Circuzz configuration.

5 Related Work

We present the first systematic fuzzing technique for testing circuit-processing pipelines. It uses metamorphic test oracles [6, 15, 38] to find logic bugs. To the best of our knowledge, there is no existing work on fuzzing *entire circuit-processing pipelines*, but the problem itself has (independently) been described in the literature as an open problem [13].

There is recent work on finding bugs in a *circuit itself* [40]. In contrast, our approach focuses on detecting bugs in a *circuit-processing pipeline*. Both types of bugs could have catastrophic consequences, but a bug in a circuit-processing pipeline may affect many, or even all, deployed circuits.

The most closely related areas to our work are fuzzing for compilers [14] and program analyzers [10, 11, 18, 20, 22–25, 27, 30, 31, 34, 37, 39, 41, 42, 44–46], such as software model checkers [8, 33] and abstract interpreters [16]. After all, the first stage in circuit-processing pipelines typically invokes a compiler for the ZK language. Similarly, most program analyzers have a compiler frontend that parses the input programs and often translates them into an intermediate language used for the analysis. The translation could generate a control-flow graph (as in many dataflow analyzers [26] and abstract interpreters), a program in an intermediate verification language [4, 19, 36] (as in many deductive verifiers, such as Dafny [28] and Spec# [5]), or a GOTO program (as in CBMC [8] and several other software model checkers).

On the other hand, regular compilers typically translate from a high-level language (such as C) to a more low-level language (such as assembly or LLVM bitcode). In contrast, ZK pipelines translate to a constraint system, and there are several later stages that make heavy use of cryptographic primitives for generating and verifying proofs. ZK pipelines are, therefore, highly complex and may contain even more subtle and hard-to-detect bugs than regular compilers. MTZK [43] is a testing framework that aims to find bugs in ZK *compilers* (and not entire pipelines) using metamorphic testing. In other words, MTZK does not target the later stages of ZK pipelines, which is a key contribution of our work.

Generally, most existing work on fuzzing for compilers and program analyzers uses one or more of the following three types of oracles: (1) specification-based oracles [6] (by comparing the actual behavior to a formal specification of the expected behavior), (2) differential oracles [6, 32] (by comparing the behavior of two or more implementations), and (3) metamorphic oracles. In this work, we have mainly focused on metamorphic oracles and have used specification-based oracles to express correctness properties of the pipelines. In the future, we plan to incorporate more metamorphic transformations for settings, for instance, by enabling different proof systems via flags.

6 Conclusion

We have presented Circuzz, the first fuzzer for detecting logic bugs in circuit-processing pipelines. It introduces CircIL, an intermediate language for circuit generation, and rewrite rules over this language for metamorphic circuit transformations. Circuzz translates the generated and transformed circuits into the ZK language of the pipeline under test and generates inputs for them using blackbox fuzzing. Bugs are detected by executing the pipeline under test on the circuits and checking for violations of the metamorphic oracles and other non-metamorphic, correctness properties. We used Circuzz to test four diverse ZK pipelines and detected critical bugs in all of them.

Despite the bug-finding effectiveness of Circuzz, there are still several (optional) components of ZK pipelines that are not being tested. For example, as an alternative to the existing verifier, Gnark allows generating a Solidity contract, which, when compiled and deployed on the blockchain, could also be invoked to verify a generated proof. In this example, even bugs in the generated Solidity contract, the Solidity compiler, or the Ethereum virtual machine could compromise the correctness of the extended ZK pipeline. As a next step, we plan to explore how to test such components.

Acknowledgments

We are grateful to the ZK-pipeline developers for their valuable help and to the anonymous reviewers for their constructive feedback. This work was supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22007] as well as Maria Christakis' ERC Starting grant 101076510.

References

[1] [n. d.]. Corset. https://github.com/Consensys/corset.

[2] [n. d.]. gnark: A Fast zk-SNARK Library that Offers a High-Level API to Design Circuits. https://docs.gnark.consensys.io.

[3] [n. d.]. Noir. https://noir-lang.org.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS, Vol. 4111)*. Springer, 364–387.

[5] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *CACM* 54 (2011), 81–91. Issue 6.

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41 (2015), 507–525. Issue 5.

[7] 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. *Trans. Dependable Secur. Comput.* 20 (2023), 4733–4751. Issue 6.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *TACAS (LNCS, Vol. 1579)*. Springer, 193–207.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *CCS*. ACM, 1032–1043.

[10] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.

[11] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.

[12] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*. ACM, 167–178.

[13] 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 *Security*. USENIX.

[14] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.

[15] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.

[16] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[17] Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic, and Arthur Gervais. 2024. Do You Need a Zero Knowledge Proof? *Cryptol. ePrint Arch.* (2024), 50.

[18] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ISSTA*. ACM, 1219–1231.

[19] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where Programs Meet Provers. In *ESOP (LNCS, Vol. 7792)*. Springer, 125–128.

[20] Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Constraint-Based Test Oracles for Program Analyzers. In *ASE*. ACM, 344–355.

[21] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 151–166.

[22] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *PACMSE* 1 (2024), 1656–1678. Issue FSE.

[23] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *ISSTA*. ACM, 556–567.

[24] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. In *ASE*. ACM, 319–330.

[25] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.

[26] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. ACM, 194–206.

[27] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.

[28] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370.

[29] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52 (2009), 107–115. Issue 7.

[30] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.

[31] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.

[32] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.

[33] Kenneth L. McMillan. 2018. Interpolation and Model Checking. In *Handbook of Model Checking*. Springer, 421–446.

[34] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.* 27 (2017). Issue 6.

[35] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *ICSE*. IEEE Computer Society, 550–562.

[36] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62.

[37] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *PACMPL* 5 (2021), 1–19. Issue OOPSLA.

[38] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *TSE* 42 (2016), 805–824. Issue 9.

[39] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.

[40] 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 *Security*. USENIX.

[41] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.

[42] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.

[43] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. MTZK: Testing and Exploring Bugs in Zero-Knowledge (ZK) Compilers. In *NDSS*. The Internet Society.

[44] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.

[45] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *ESEC/FSE*. ACM, 237–249.

[46] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *CoRR* abs/2402.14366 (2024).

A Circuzz Rewrite Rules

Rule ID	Match Pattern	Rewrite Template
comm-or	`(?a | ?b)`	`(?b | ?a)`
assoc-and	`((?a & ?b) & ?c)`	`(?a & (?b & ?c))`
comm-and	`(?a & ?b)`	`(?b & ?a)`
and-zero	`(?a & 0)`	`0`
inv-xor	`(?a ^ ?a)`	`0`
comm-xor	`(?a ^ ?b)`	`(?b ^ ?a)`
zero-or-rev	`(?a | 0)`	`?a`
zero-xor-rev	`(?a ^ 0)`	`?a`
inv-xor-rev	`0`	`($r ^ $r)`
zero-or	`?a`	`(?a | 0)`
zero-xor	`?a`	`(?a ^ 0)`
idem-and	`?a`	`(?a & ?a)`
zero-and	`0`	`($r & 0)`
comm-add	`(?a + ?b)`	`(?b + ?a)`
comm-mul	`(?a * ?b)`	`(?b * ?a)`
comm-div	`(?a / ?b)`	`((1 / ?b) * ?a)`
dist-mul-add	`((?a + ?b) * ?c)`	`((?a * ?c) + (?b * ?c))`
dist-add-mul	`((?a * ?c) + (?b * ?c))`	`((?a + ?b) * ?c)`
assoc-add	`((?a + ?b) + ?c)`	`(?a + (?b + ?c))`
assoc-add-rev	`(?a + (?b + ?c))`	`((?a + ?b) + ?c)`
assoc-mul	`((?a * ?b) * ?c)`	`(?a * (?b * ?c))`
assoc-mul-rev	`(?a * (?b * ?c))`	`((?a * ?b) * ?c)`
assoc-div	`((?a / ?b) * ?c)`	`(?a * (?c / ?b))`
assoc-div-rev	`(?a * (?c / ?b))`	`((?a / ?b) * ?c)`
zero-add-des	`(?a + 0)`	`?a`
one-mul-des	`(?a * 1)`	`?a`
one-div-des	`(?a / 1)`	`?a`
inv-zero-add-des	`(?a - 0)`	`?a`
neg-zero-add-des	`(0 - ?a)`	`(-?a)`
inv-add-des	`(?a - ?a)`	`0`
inv-assoc-neg2pos	`((?a - ?b) - ?c)`	`(?a - (?b + ?c))`
inv-assoc-pos2neg	`(?a - (?b + ?c))`	`((?a - ?b) - ?c)`
inv-addition-inl	`(?a + (-?c))`	`(?a - ?c)`
double-negation-add-des	`(-(-?a))`	`?a`
pow2-to-mul	`(?a ** 2)`	`(?a * ?a)`
pow3-to-mul	`(?a ** 3)`	`((?a * ?a) * ?a)`
mul-to-pow2	`(?a * ?a)`	`(?a ** 2)`
mul-to-pow3	`((?a * ?a) * ?a)`	`(?a ** 3)`
zero-add-con	`?a`	`(?a + 0)`
one-mul-con	`?a`	`(?a * 1)`
one-div-con	`?a`	`(?a / 1)`
inv-zero-add-con	`?a`	`(?a - 0)`
neg-zero-add-con	`(-?a)`	`(0 - ?a)`
inv-addition-exp	`(?a - ?b)`	`(?a + (-?b))`
double-negation-add-con	`?a`	`(-(-?a))`
add-sub-random-value	`?a`	`((?a - $r) + $r)`
inv-div-des	`(?a / ?a)`	`1`
zero-lor-des	`(?a || 0)`	`?a`
zero-land-des	`(?a && 1)`	`?a`
taut-lor	`(?a || 1)`	`1`
contra-land	`(?a && 0)`	`0`

Rule ID	**Match Pattern**	**Rewrite Template**
assoc-lor	((?a || ?b) || ?c)	(?a || (?b || ?c))
assoc-land	((?a && ?b) && ?c)	(?a && (?b && ?c))
comm-lor	(?a || ?b)	(?b || ?a)
comm-lan	(?a && ?b)	(?b && ?a)
dist-lor-land	((?a && ?b) || ?c)	((?a || ?c) && (?b || ?c))
dist-land-lor	((?a || ?c) && (?b || ?c))	((?a && ?b) || ?c)
de-morgan-land-con	(!(?a && ?b))	((!?a) || (!?b))
de-morgan-land-des	((!?a) || (!?b))	(!(?a && ?b))
de-morgan-lor-con	(!(?a || ?b))	((!?a) && (!?b))
de-morgan-lor-des	((!?a) && (!?b))	(!(?a || ?b))
double-negation-des	(!(!?a))	?a
double-land-des	(?a && ?a)	?a
double-lor-des	(?a || ?a)	?a
double-lxor-des	(?a ‘?a)	0
comm-lxor	(?a ‘?b)	(?b ‘?a)
lxor-to-or-and	(?a ‘?b)	(((!?a) && ?b) || (?a && (!?b)))
zero-lor-con	?a:bool	(?a || 0)
zero-land-con	?a:bool	(?a && 1)
double-negation-con	?a:bool	(!(!?a))
double-land-con	?a:bool	(?a && ?a)
double-lor-con	?a:bool	(?a || ?a)
double-lxor-con	0:bool	($r:bool ‘$r:bool)
or-and-to-lxor	(((!?a) && ?b) || (?a && (!?b)))	(?a ‘?b)
commutativity-equ	(?a == ?b)	(?b == ?a)
relation-geq-to-leq	(?a >= ?b)	(?b <= ?a)
relation-leq-to-geq	(?a <= ?b)	(?b >= ?a)
relation-leq-to-lth-and-equ	(?a <= ?b)	((?a < ?b) || (?a == ?b))
relation-lth-and-equ-to-leq	((?a < ?b) || (?a == ?b))	(?a <= ?b)
relation-geq-to-gth-and-equ	(?a >= ?b)	((?a > ?b) || (?a == ?b))
relation-gth-and-equ-to-geq	((?a > ?b) || (?a == ?b))	(?a >= ?b)
relation-leq-to-not-gth	(?a <= ?b)	(!(?a > ?b))
relation-not-gth-to-leq	(!(?a > ?b))	(?a <= ?b)
relation-geq-to-not-lth	(?a >= ?b)	(!(?a < ?b))
relation-not-lth-to-geq	(!(?a < ?b))	(?a >= ?b)
relation-neq-to-not-equ	(?a != ?b)	(!(?a == ?b))
relation-not-equ-to-neq	(!(?a == ?b))	(?a != ?b)