Practical Security Analysis of Zero-Knowledge Proof Circuits

Hongbo Wen
hongbowen@ucsb.edu
UCSB

Jon Stephens
jon@cs.utexas.edu
UT Austin

Yanju Chen
yanju@cs.ucsb.edu
UCSB

Kostas Ferles
kostas@veridise.com
Veridise

Shankara Pailoor
spailoor@cs.utexas.edu
UT Austin

Kyle Charbonnet
kylecharbonnet@gmail.com
Ethereum Foundation

Isil Dillig
isil@cs.utexas.edu
UT Austin & Veridise

Yu Feng
yufeng@cs.ucsb.edu
UCSB & Veridise

Abstract

As privacy-sensitive applications based on zero-knowledge proofs (ZKPs) gain increasing traction, there is a pressing need to detect vulnerabilities in ZKP circuits. This paper studies common vulnerabilities in Circom (the most popular domain-specific language for ZKP circuits) and describes a static analysis framework for detecting these vulnerabilities. Our technique operates over an abstraction called the *circuit dependence graph (CDG)* that captures key properties of the circuit and allows expressing *semantic vulnerability patterns* as queries over the CDG abstraction. We have implemented 9 different detectors using this framework and perform an experimental evaluation on over 258 circuits from popular Circom projects on Github. According to our evaluation, these detectors can identify vulnerabilities, including previously-unknown ones, with high precision and recall.

1 Introduction

Zero knowledge protocols (ZKP) have gained traction as a mechanism for proving that a statement is true without providing any other information beyond the correctness of the statement itself. Since their inception, ZKPs have found numerous applications in authentication systems [17, 33], online voting [13], privacy-preserving cryptocurrencies [11], and scaling solutions for blockchains [31].

A particularly attractive zero-knowledge proof protocol is a so-called zkSNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge), which allows one party (the *prover*) to generate a *succinct* proof that allows the other party (the *verifier*) to *efficiently* check the correctness of the proof. What makes zkSNARKs particularly appealing is that the prover and the verifier can be automatically generated from a so-called *arithmetic circuit* describing a set of polynomial equations over a finite field. Hence, given some computation P on input x, developers of ZK applications only need to construct an arithmetic circuit C such that $y = f(x)$ if and only if $C(x,y)$ is true. Then, given such a circuit C, a zkSNARK proof generator can be used to generate a prover and verifier, thereby allowing the prover to establish that $y = f(x)$ without revealing any information about x to the verifier.

Given the importance of arithmetic circuits in the ZKP domain, several domain-specific languages, such as Circom [9] and Leo [16], have emerged to facilitate the construction of arithmetic circuits. However, despite language and compiler support, building *correct* arithmetic circuits remains a challenge. In particular, given some computation f on input x, the developer still needs to generate a set of constraints C such that $C(x,y)$ is true if $y = f(x)$. If the constraints crafted by the developer do not possess this property, then the resulting ZKP can allow the verifier to accept a *bogus* proof that $y = P(x)$ even though this is actually not the case. Indeed, bugs in arithmetic circuits can result in catastrophic consequences, such as allowing attackers to forge signatures [6] or mint counterfeit cryptocoins [21]. Even worse, attacks on ZK applications are extremely hard to detect, even once they occur, because the only information provided by the attacker, namely the proof, is already accepted as being valid by the application. Hence, there is a pressing need for developing *static* (i.e., compile-time) techniques for ensuring that arithmetic circuits do not contain subtle vulnerabilities.

Motivated by this problem, this paper describes a static analysis framework for automatically detecting common vulnerabilities in arithmetic circuits written in Circom, which is currently the most popular domain-specific language for zkSNARKs. Based on our manual inspection of real-world vulnerabilities in Circom programs, we identify several semantic patterns that underlie common vulnerabilities and propose a static analysis framework for detecting such patterns. Our framework allows describing semantic vulnerability patterns using a domain-specific language, at the heart of which lies an abstraction of Circom programs called a *circuit dependence graph*. Given a semantic vulnerability pattern expressed in this DSL, our framework can statically check whether a Circom program exhibits that vulnerability.

We have implemented a library of several vulnerability checkers on top of this framework and used them to detect vulnerabilities in 258 arithmetic circuits taken from popular

Circom projects on Github. Our detectors have high precision and recall and have collectively uncovered 32 unique previously-unknown vulnerabilities.

To summarize, this paper makes the following key contributions:

- We present a taxonomy of common vulnerabilities in Circom arithmetic circuits.
- We introduce a program abstraction called *circuit dependence graph (CDG)*, which is useful for expressing and detecting vulnerabilities.
- We present a *vulnerability description language (VDL)* for expressing semantic vulnerability patterns in arithmetic circuits.
- We describe a library of static checkers based on this VDL and evaluate them on 258 arithmetic circuits. Our evaluation shows that our checkers have a low false positive rate (16.4%) and collectively detect over 98.6% of vulnerabilities in these circuits. We also show that our approach can detect previously unknown vulnerabilities.

2 Background

In this section, we provide some necessary background on Zero Knowledge Proofs and give an overview of the syntax and semantics of Circom programs.

2.1 Zero-Knowledge Protocols

A zero-knowledge protocol allows one party, the *prover* $\mathcal{P}$, to convince another party, *the verifier* $\mathcal{V}$, that they know some secret information without revealing what it actually is. In more detail, given some computation f that operates over public inputs x and secret information y, the prover needs to convince the verifier that some output z corresponds to the result of $f(x,y)$ without revealing any information whatsoever about y. What makes such a system *zero-knowledge* is that the prover does not reveal anything about y beyond proving that they have access to it.

While there are several different types of zero-knowledge protocols, *zkSNARK* (*Zero-Knowledge Succinct Non-interactive ARgument of Knowledge*) protocols have become increasingly popular due to their succinct proof size and sub-linear verification time. An appealing property of zkSNARKs is that a suitable prover and verifier can be automatically generated from an *arithmetic circuit* representation of the computation. Hence, several domain-specific languages, such as Circom and Leo, have been designed to facilitate the construction of arithmetic circuits.

An arithmetic circuit is essentially a set of polynomial equations over a finite field. In particular, an arithmetic circuit takes some input signals that are values in the range $[0,p)$

```
1  pragma circom 2.0.0;
2
3  template Multiplier2(){
4    //Declaration of signals
5    signal input in1;
6    signal input in2;
7    signal output out;
8    out <== in1 * in2;
9  }
10
11 component main {public [in1,in2]} = Multiplier2();
```

Figure 1: A simple example written in Circom.

and performs additions and multiplications modulo a prime number p. The output of every addition and multiplication produces a signal, which is characterized as either an *intermediate* signal or an *output* signal, which denotes the final public output.

2.2 The Circom Language

Because this work primarily targets arithmetic circuits expressed in Circom, we provide some background on the Circom DSL. At a high level, Circom allows expressing computations in a manner that is low-level enough to be amenable to the generation of a zero-knowledge proof system. Given a program written in Circom, the compiler generates two artifacts:

- **Witness calculator program:** Given some input, this program computes the values of all the other signals (both intermediate and output). We refer to the witness generator as the computation expressed by the Circom program and denote it by f.
- **Rank-1 constraint system (R1CS)**: Because zkSNARK protocols operate over Rank-1 constraint systems, the Circom compiler translates the input program into its corresponding R1CS representation. Then, a zkSNARK generator (e.g., snark-js) is used to automatically produce both the prover and the verifier. We refer to the R1CS representation as the arithmetic circuit expressed by the Circom program and denote it by C_f.

Hence, given a Circom program, one typically uses the outputs of the Circom compiler in the following way: One party, say Alice, uses the witness generator program f and the prover P to generate a proof that they have correctly computed $f(x,y) = z$ for some public input x, secret information y, and output z. To do so, they first use f to compute the values of *all signals* s of the corresponding arithmetic circuit $C_f(x,y,z)$. Next, they construct a proof that they performed this computation by passing s as input to P which in turn generates a short proof π that $z = f(x,y)$. Then, another party, say Bob, can use π along with the public input x to check whether $x = f(x,y)$ without having access to y. This usage scenario is illustrated in Figure 2.

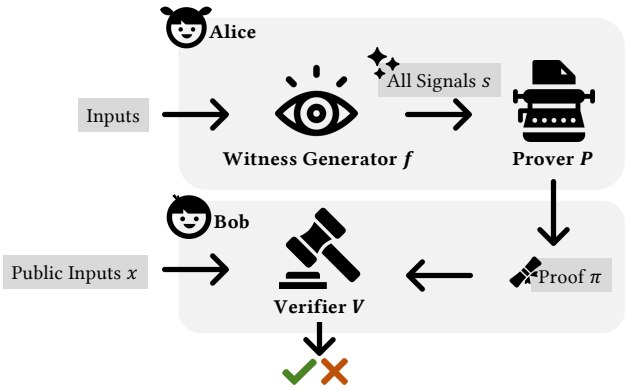

Figure 2: Usage of zero-knowledge proof: Alice generates a proof using witness generator and prover, and Bob verifies the proof using public inputs.

The crucial point here is that the proof π should only be accepted by the verifier iff the output z is indeed equal to $f(x,y)$. However, if there is a bug in the original Circom program, i.e, the arithmetic circuit C_f does not correspond to the computation f, then the verifier may end up accepting bogus proofs. In particular, a malicious party can exploit bugs in the Circom program to produce a *bogus* witness i.e, one which cannot be generated by f, and then use the prover to generate a "proof" that is accepted by the verifier.

Since understanding how bugs can arise in this context requires some background about Circom syntax and semantics, we provide a brief Circom tutorial that is needed for understanding the rest of the paper.

Circom signals. Circuits operate over *signals*, which are finite field elements (or arrays thereof) and are declared using the keyword `signal`. Signals can be qualified as input or output, and any signals without input or output annotations are considered to be intermediate signals. Given an input signal, the witness generator produced by the Circom compiler computes the values of all intermediate and output signals.

Circom operators. Because the Circom compiler is used to generate both the witness calculator program as well as the R1CS constraints, it provides operators that can be used to express both computation (for witness calculation) as well as constraint generation. In particular, the Circom operators `<--` and `-->` are used for signal assignment and are crucial for witness generation. In contrast, the `===` operator is used for *constraint generation*; for example, the statement `x === y` tells the Circom compiler to generate a circuit that only accepts signals where the value of `x` is equal to that of `y`. In practice, however, there is a close corrrespondence between witness computation and constraint generation: typically, one needs to add a constraint whenever they use the assignment operator. For this reason, Circom also provides two additional operators, namely `<==`, `==>`, to perform both assignment and constraint generation simultaneously. But there is one key complication: the operands of `===` must obey certain restrictions (e.g., involving only quadratic expressions) in order to facilitate the generation of a Rank-1 constraint system. Hence, it is not always possible to emulate every assignment in Circom directly using the `<==` operator. This introduces one of the challenges of expressing computation in Circom.

Templates. Generic circuits in Circom are referred to as *templates*, and their parametric values must be instantiated when the template is used. Templates can be instantiated using the keyword *component* and by providing the necessary parameters like so:

```
1 component c = templateId(v1,...,vn);
```

As an example, consider the Circom program shown in Figure 1. First, we use the reserved keyword `template` to define a generic circuit `Multiplier2`. This circuit contains two input signals `in1` and `in2` and a public output signal called `out`. In line 8, we use `<==` to set the value of `out` to be the result of multiplying the values of `in1` and `in2`. Equivalently, we could have also used the operator `==>`, e.g., `in1 * in2 ==> out`. Finally, the values of `in1` and `in2` are public through the instantiation of the component in line 11.

2.3 Threat Model

Since we focus on vulnerabilities in the design and implementation of ZK circuits, we make no assumptions about the quality and robustness of their underlying codebase. For attacks on ZK circuits, we consider a trustless set-up where an attacker can access all public information, including but not limited to (a) on-chain blockchain states such as blocks, transactions, accounts, and balances; (b) deployed smart contract bytecode that interacts with ZK circuits; (c) source code of ZK circuits. Furthermore, beyond directly interacting with the victim ZK applications themselves, we assume attackers can deploy their own contract and ZK application on the blockchain, which can in turn invoke public transactions of the target victim contract and circuits.

3 Taxonomy of Circom Vulnerabilities

Since our goal in this work is to understand common Circom vulnerabilities and develop effective techniques to address them, we started out by studying dozens of existing vulnerabilities in Circom programs. As a result of this manual inspection, we were able to come up with a taxonomy of different vulnerability patterns, which we classify into three main classes:

- **Nondeterministic signals**, where the input or output signals of a circuit are not properly constrained,

```
1 template ArrayXOR(n) {
2     signal input a[n];
3     signal input b[n];
4     signal output out[n];
5     for (var i = 0; i < n; i++) {
6         out[i] <-- a[i] ^ b[i];
7     }
8 }
9 component main = ArrayXOR(2);
```

Figure 3: A code snippet demonstrating an underconstrained circuit caused by nondeterministic signals in array `out`.

- **Unsafe component usage**, where signals of a component are not used correctly,
- **Constraint-computation discrepancies**, where there is a divergence between the witness generation code and the generated constraints.

Based on our manual inspection, we conjecture that these three classes cover over 90% of existing vulnerabilities in Circom programs. In the remainder of this section, we explain each vulnerability class in more detail and provide examples.

3.1 Nondeterministic Signals

We found that many bugs in Circom programs are caused by *non-deterministic signals*. We say that a signal x is nondeterministic if it can take multiple values for a given assignment to the input signals. Intuitively, if a signal is nondeterministic, the Rank-1 constraint system generated by the compiler will not properly constrain its value, often causing it to accept bogus proofs.

A common source of non-determinism is due to signals that are *unconstrained*, meaning that the signal is not even mentioned in the generated Rank-1 constraint system. If such an unconstrained signal corresponds to an output, this is guaranteed to be a bug.

For example, Figure 3 shows an example from the `circom-pairing`[1] library which implements core cryptographic pairing algorithms and their corresponding computational infrastructure to support signature verification. The `ArrayXOR` template can be instantiated to compute the exclusive disjunction of pairs of signals from the two input arrays, `a` and `b`. However, the circuit does not generate a constraint for `out[i]` in line 6: In particular, because the operator `<--` only performs signal assignment, the circuit ends up *not* constraining the value of the output signal, thereby allowing attackers to forge signatures for potential malicious behavior.

In addition to output signals, input and intermediate signals that are declared but never used in the constraints are often also problematic. For example, Figure 4 shows a code snippet from Hermez Network[2] with unused input signals, where

[1] https://github.com/yi-sun/circom-pairing
[2] https://github.com/hermeznetwork/circuits

```
1  template BitsCompressed2AySign(){
2      signal input bjj[256];
3      signal output ay;
4      signal output sign;
5      component b2nAy = Bits2Num(254);
6      var i;
7      for (i = 0; i < 254; i++) {
8          b2nAy.in[i] <== bjj[i];
9      }
10     ay <== b2nAy.out;
11     sign <== bjj[255];
12 }
13 component main {public bjj}
14     = BitsCompressed2AySign();
```

Figure 4: A code snippet demonstrating a circuit that violates functional correctness due to an unused circuit signal `bjj[254]`.

an array `bjj` is declared in line 2 but one of its elements, namely `bjj[254]`, is not used. Such unused input signals could indicate functional correctness issues but can also result in vulnerabilities in the application that verifies the proof. This is the case because, to verify a circuit, one must provide the set of outputs and public input signals that were used to generate the proof. But, in cases where a public input is completely unused, any input could be used to verify the proof regardless of the value provided to the prover. For the code in Figure 4, this means that a user can generate a proof where `bjj[254]` is 0 and still verify the resulting proof with a non-zero value for `bjj[254]`.

3.2 Unsafe Component Use

Recall that a component is an instantiated template that can be used like a function. In practice, many templates are written in such a way that they expect callers to constrain their inputs and outputs. Hence, if a component is invoked without any constraints on their inputs or outputs, this usage pattern is highly suspicious.

For instance, Figure 5 shows an unsafe component use where the template *input* is unconstrained. Here, `hash` is instantiated from the template `HashMsg` from the MACI[3] project at line 7. In order to correctly verify the hash of the current message `msg`, the call to the `hash` component requires its input signal `in` to be properly constrained. However, as discussed earlier in Section 2, the assignment operator `<--` used at line 8 only performs assignment without introducing a constraint. Thus, the input to the hash computation is completely constrained, resulting in an underconstrained output signal `msgHash`, which can in turn cause the acceptance of an arbitrary message.

Just as unconstrained inputs often correlate with bugs, unconstrained component *outputs* are also very suspicious, as most templates are written in a way that expects their users to constrain the output signal at the call site. For instance,

[3] https://github.com/privacy-scaling-explorations/maci

```
template ProcessMessages(msgBits) {
    signal input msg;
    signal input secretKey;
    signal output msgHash;
    component msgChecker = Num2Bits(msgBits);
    msgChecker.in <== msg;
    component hash = HashMsg(msgBits);
    hash.in <-- msg;
    hash.secretKey <== secretKey;
    msgHash <== hash.out;
}
component main = ProcessMessages(64);
```

Figure 5: A code snippet showing a bug caused by an unconstrained component input `hash.in`, which causes acceptance of arbitrary `msg`.

```
template CoreVerifyPubkeyG1(n, k){
    // ...
    component lt[10];
    for(var i=0; i<10; i++){
        lt[i] = BigLessThan(n, k);
        for(var idx=0; idx<k; idx++)
            lt[i].b[idx] <== q[idx];
    }
    for(var idx=0; idx<k; idx++){
        lt[0].a[idx] <== pub[0][idx];
        // ...
    }
    // ...: lt.out is not used
}
component main = CoreVerifyPubkeyG1(55, 7);
```

Figure 6: A code snippet that contains unused component output signals `lt[i].out`, which causes the absence of constraint for restricting inputs `lt[i].a` and `lt[i].b`, thus making the circuit underconstrained.

Figure 6 shows an example of an unused component output vulnerability in `circom-pairing`, a Circom implementation of elliptic curve pairings for the widely-adopted BLS12–381 curve. This bug is caused by using the `BigLessThan` template incorrectly. In particular, note that each `lt[i]` in this example is a component instantiating `BigLessThan` at line 5. Because the `BigLessThan` template requires the caller to explicitly constrain its output signal `lt[i].out` in order to properly compare its two inputs `lt[i].a` and `lt[i].b`, `lt[i].out` is unconstrained at the call site. As a result, `lt[i].a` and `lt[i].b` can take arbitrary values, causing the whole circuit to be underconstrained.

3.3 Constraint-Computation Discrepancies

Recall that Circom programs are simultaneously used for generating witness calculation programs as well as a Rank-1 constraint system. However, even though these two aspects are intermingled in Circom code, there could nonetheless be unintended discrepancies between the two aspects. For example, certain operators that are valid for witness generation are actually often invalid for constraint generation. Similarly, recall that *not* every computation can be directly expressed as a constraint; hence, two expressions that are intended to express the same value in two syntactically different ways may actually end up being semantically different, causing discrepancies between witness computation and constraint generation.

```
template Edwards2Montgomery() {
    signal input in[2];
    signal output out[2];
    out[0] <-- (1 + in[1]) / (1 - in[1]);
    out[1] <-- out[0] / in[0];
    out[0] * (1-in[1]) === (1 + in[1]);
    out[1] * in[0] === out[0];
}
component main = Edwards2Montgomery();
```

Figure 7: A code snippet showing a bug caused by an excepetion case when using a sensitive operator `/`. Here `(1 - in[1])` and `in[0]` are creating a discrepancy between the witness and constraint generation.

To gain more insight about such vulnerabilities, consider the `Edwards2Montgomery` circuit in Figure 7, which is taken from `circomlib`[4], one of the most popular libraries for constructing zero-knowledge protocols. This circuit is intended to convert a point in an Edwards curve to one in a Montgomery curve. Observe that lines 4 and 5 use the multiplicative inverse operator in the right hand side of the assignment, while lines 6 and 7 try to introduce a constraint that "emulates" this computation. Note that the code cannot combine the assignment and constraint generation via the use of the `<==` operator because R1CS constraints cannot contain multiplicative inverse operations. Hence, lines 6 and 7 try to introduce constraints that mimic the computation. However, there is a subtle bug: the constraints generated allow the inverted signal `in[0]` to be `0` and still satisfy the constraints. While the witness generator will never generate witness with `in[0] == 0`, since the multiplicative inverse is undefined in those cases, the constraints introduced at lines 6 and 7 actually accept those undefined values. This makes the overall circuit underconstrained since any assignment to `out[1]` will satisfy the constraints when when `in[0] == 0` and `out[0] == 0`.

As another example of constraint-computation discrepancies, consider the buggy circuit in Figure 8. As in the previous example, line 6 tries to introduce a constraint that emulates the computation from line 5 since the R1CS representation does not permit the integer divison operation $\backslash$ in the constraints. However, because the assignment at line 5 ignores the possible remainder of the division, there is a discrepancy between lines 5 and 6. In this case, the circuit is more constrained than the computation as the computation will accept arbitrary `inp` values, while the constraint will only be satisfied when `inp` is a multiple of `gwei`.

[4] https://github.com/iden3/circomlib

```
1 template Reward() {
2     signal input inp;
3     signal output out;
4     var gwei = 10 ** 6;
5     out <-- inp \ gwei;
6     out * gwei === inp;
7 }
8
9 component main = Reward();
```

Figure 8: A code snippet showing a witness assignment (line 5) that has different semantic meaning than its counterpart in constraint generation (line 6). Such a discrepancy leads to a violation of function correctness in the circuit.

3.4 Key Lessons

Because the goal of our manual inspection is to make it possible to automatically detect common vulnerabilities, we conclude this section by summarizing some key take-away lessons that motivate the design of our static checking framework.

The role of semantic analysis. One of the key lessons from our manual inspection is that detecting many real-world vulnerabilities requires reasoning about the semantics of Circom code. For example, to detect discrepancies between the computation and the generated constraints, we need to understand the semantics of the code, at least to some extent.

Wide variety of different bugs. While our taxonomy identifies three main root causes of vulnerabilities, there are significant variations in how these vulnerabilities manifest themselves in code. Thus, from the perspective of statically finding vulnerabilities, we may need to write many different detectors to get a high coverage of all vulnerabilities.

Useful abstractions. Even though different bugs exhibit different syntactic patterns, there are nonetheless common *semantic abstractions* that underlie all of these patterns. For example, several bugs we have encountered can be understood by considering the discrepancies between variable dependencies at the computational vs. constraint level.

4 Static Analysis Framework

Drawing inspirations from the key lessons discussed in Section 3.4, we describe a static analysis framework we have developed for practical security analysis of Circom programs. At the heart of our framework lies a graph representation that we refer to as *Circuit Dependence Graph (CDG)*. In the remainder of this section, we describe our CDG abstraction, present a domain-specific language for describing semantic vulnerability patterns based on this abstraction, and explain our static analysis for constructing the CDG.

4.1 Circuit Dependence Graphs

As mentioned in Section 3.4, many vulnerabilities arise due to discrepancies in signal dependencies at the computational vs. constraint level. Thus, our proposed *circuit dependence graph (CDG)* abstraction is designed to encode such dependencies. Specifically, a CDG is a graph $G = (V, E_d, E_c)$ where V denotes the set of vertices, E_d denotes a set of arcs (i.e., directed edges) encoding dependencies at the computational level, and E_c is a set of *undirected* edges encoding dependencies at the constraint level.

In more detail, we differentiate between three types of nodes in the CDG:

- **Signal Nodes $V_s \in V$:** represent signals in the program.
- **Constant Nodes $V_c \in V$:** denote fixed literal value in the program (such as a concrete field element).

As mentioned previously, we also differentiate between two types of edges:

- **Data Flow Edges E_d:** A data flow edge $(u, v) \in E_d$ is a *directed* edge indicating direct data flow from u to v, which can be derived from the assignment operators (e.g., `<--` and `<==`).
- **Constraint Edges $E_c \in E$:** A constraint edge $(u, v) \in E_c$ is an *undirected* edge indicating that u and v appear in the same constraint, which can be derived from the constraint operators (e.g., `===` and `<==`).

At a high level, a data flow edge from u to v labeled by s indicates that the value of u directly depends on v due to an assignment to u of an expression s containing v. Similarly, a constraint edge between u and v labeled s indicates that there is an equation s directly relating u and v.

4.2 CDG Construction

Given a Circom program, our method constructs the CDG by first representing the syntax of the program using the predicates shown in Table 1 and then adding new nodes and edges to the CDG using the (Datalog-style) inference rules shown in Figure 9. As standard, a rule of the form:

$$P(\overline{x}) \Leftarrow P_1(\overline{x_1}), \ldots, P_n(\overline{x_n})$$

indicates that predicate $P(\overline{x})$ is inferred to be true if all the other predicates $P_1, \ldots, P_n$ are also true. We now briefly explain our CDG construction rules.

Nodes. The first two rules (1) and (2) in Figure 9 create nodes of the CDG abstraction. In particular, a signal (resp. constant) v in the Circom program is mapped to a signal node $\mathsf{sNode}(v)$ (resp. constant node $\mathsf{cNode}(v)$).

Predicate	Indication
`sig(v)`	`v` is declared as a signal of the circuit.
`in(v)`	`v` is declared as an input signal of the circuit.
`out(v)`	`v` is declared as an output signal of the circuit.
`const(v)`	`v` is a constant value.
`expr(e)`	`e` is an expression.
`quad(e)`	`e` is a quadratic expression.
`contains(e,v)`	`e` is an expression that contains signal `v`.
`is(x1,x2)`	`x1` is identical to `x2` in source code.
`assignedTo(e,v)`	`e` is assigned to `v`.
`equivTo(e1,e2)`	`e1` and `e2` are in the same constraint.
`sigOf(v,c)`	`v` is a signal declared in component `c`.
`inOf(v,c)`	`v` is an input signal declared in component `c`.
`outOf(v,c)`	`v` is an output signal declared in component `c`.
`exprDiv(e1,e2,e3)`	`e1` is a division expression with `e2` as dividend and `e3` as divisor.
`exprIte(e1,e2, e3,e4)`	`e1` is an if-then-else expression with `e2` as condition, `e3` as then branch and `e4` as else branch.

Table 1: Some of the predicates used for representing the syntax of Circom programs.

Data flow edges. The next four rules (3)–(6) describe how data flow edges are constructed. Rule (3) adds data flow edges to model direct assignments in the Circom program. That is, given a statement that assigns to v an expression e containing signal u, we add a data flow edge from u to v and record its label e using the dLabel predicate, as shown in rule (5). Rule (4) summarizes the data flow relations between the input and output of a component. To do so, it constructs the CDG for this component and checks whether there is a data flow dependency between the input and output. Note that, in this rule, the notation R^+ denotes the transitive closure of relation R. Since data flow edges for components only *summarize* data flow inside components, we label their edges using a special symbol $\cdot$, as shown in rule (6).

Constraint edges. Rules (7)–(12) describe the construction of constraint edges E_c. According to rules (7) and (10), if there is a constraint of the form $e_1 === e_2$ where e_1 and e_2 contain expressions u and v respectively, we add a constraint edge between these two signals and label this edge as $e_1 \equiv e_2$. Rules (8) and (11) are the counterpart of rules (4) and (6) for components, but handling constraint edges rather than data flow. In particular, these rules again conceptually construct the CDG for the relevant component and adds a constraint edge between the input and output variables if there is any constraint relating them. As before, such "summary" edges are labeled using the special symbol $\cdot$. Finally, rule (9) and (12) express that constraint edges are symmetric and add an edge (u,v) with the same label for each edge (v,u).

$$\mathsf{sNode}(v) \Leftarrow \mathsf{sig}(v) \quad (1)$$
$$\mathsf{cNode}(v) \Leftarrow \mathsf{const}(v) \quad (2)$$
$$\mathsf{dEdge}(u,v) \Leftarrow \mathsf{assignedTo}(e,v), \mathsf{contains}(e,u) \quad (3)$$
$$\mathsf{dEdge}(u,v) \Leftarrow \mathsf{inOf}(u,c), \mathsf{outOf}(v,c), \mathsf{dEdge}^+(u,v) \quad (4)$$
$$\mathsf{dLabel}(u,v,e) \Leftarrow \mathsf{assignedTo}(e,v), \mathsf{contains}(e,u) \quad (5)$$
$$\mathsf{dLabel}(u,v,\cdot) \Leftarrow \mathsf{inOf}(u,c), \mathsf{outOf}(v,c), \mathsf{dEdge}^+(u,v) \quad (6)$$
$$\mathsf{cEdge}(u,v) \Leftarrow \mathsf{contains}(e_1,u), \mathsf{contains}(e_2,v), \mathsf{equivTo}(e_1,e_2) \quad (7)$$
$$\mathsf{cEdge}(u,v) \Leftarrow \mathsf{inOf}(u,c), \mathsf{outOf}(v,c), \mathsf{cEdge}^+(u,v) \quad (8)$$
$$\mathsf{cEdge}(u,v) \Leftarrow \mathsf{cEge}(v,u) \quad (9)$$
$$\mathsf{cLabel}(u,v,e_1 \equiv e_2) \Leftarrow \mathsf{contains}(e_1,u), \mathsf{contains}(e_2,v), \mathsf{equivTo}(e_1,e_2) \quad (10)$$
$$\mathsf{cLabel}(u,v,\cdot) \Leftarrow \mathsf{inOf}(u,c), \mathsf{outOf}(v,c), \mathsf{cEdge}^+(u,v) \quad (11)$$
$$\mathsf{cLabel}(u,v,s) \Leftarrow \mathsf{cLabel}(v,u,s) \quad (12)$$

Figure 9: Datalog style inference rules describing CDG construction, where c is a component, u, v correspond to signals and constants e corresponds to an expression, and s correspond to a statement. Given relation R, R^+ denotes its transitive closure.

ϕ ::= `node(`v_1`)` | `sig(`v_1`)` | `const(`v_1`)` | `in(`v_1`)` | `out(`v_1`)`
| `dEdge(`v_1`,`v_2`)` | `cEdge(`v_1`,`v_2`)`
| `dLabel(`v_1`,`v_2`,`e_1`)` | `cLabel(`v_1`,`v_2`,`s_1`)`
| `sigOf(`v_1`,`c_1`)` | `inOf(`v_1`,`c_1`)` | `outOf(`v_1`,`c_1`)`
| `matchExp`(e, *pattern*)
| ϕ, ϕ | $!\phi$ | $\phi \vee \phi$

r ::= ϕ :- ϕ

$v_i \in$ **nodes**, $c_i \in$ **components**
$s_i \in$ **statements**, $e_i \in$ **expressions**

Figure 10: Vulnerability Description Language

4.3 Vulnerability Description Language

As mentioned in Section 3.4, vulnerabilities in Circom programs take a variety of different forms despite having a few underlying root causes. Hence, in order to build a practical tool that does not suffer from many false positives, it is imperative to build detectors that can accurately identify key semantic vulnerability patterns. To this end, we have designed an extensible vulnerability description language (VDL) for expressing semantic patterns over the proposed CDG representation.

Figure 10 shows the syntax of our vulnerability description language. At a high level, the vulnerability description language has Datalog-like syntax and provides a number of built-in predicates for describing the syntax of the input Circom program as well as properties of the CDG. For example, the predicate `dEdge(`v_1`,`v_2`)` (resp. `cEdge(`v_1`,`v_2`)` denotes a dataflow (resp. constraint) edge from v_1 to v_2, and $R^+(v_1,v_2)$

denotes the transitive closure of binary relation R. The VDL also provides a number of additional predicates for checking signal types and which signals are declared in which components. For example, `sig(v)` checks whether v is a signal and `sigOf(v, c)` evaluates to true if signal v is declated in circuit c. Finally, the construct `matchExp` allows performing syntactic pattern matching on expressions labeling the edges of the circuit dependence graph.

Checking Circom programs against VDL specifications. We refer to programs in our DSL as *anti-patterns* or *semantic vulnerability patterns*. Given a library $\mathcal{L}$ of such anti-patterns and a Circom program P, our static analysis framework first constructs the CDG of the program and checks whether P is an instance of some $\alpha \in \mathcal{L}$. To do so, we first evaluate the truth value of all ground predicates in the VDL and them as facts to a Datalog program which also includes the code describing the anti-patterns. Finally, to check if the program matches anti-pattern α, we simply issue a Datalog query $\alpha(x)$?, which returns all instances of anti-pattern α for program P.

4.4 Detectors based on VDL

Drawing insights from our manual inspection described in Section 3, we have implemented a library of ZKP anti-patterns based on our VDL from Section 4.3. In what follows, we describe our detectors in more detail.

Unconstrained Circuit Output (UCO) As discussed in Section 3.1, unconstrained outputs of a circuit indicate a serious vulnerability. In particular, if an output signal is neither constrained to be a constant nor depends (transitively) on an input signal, this means it is unconstrained. This can be described in our VDL using the following anti-pattern:

```
 sigDep(v) :- sig(u), cEdge+(u,v)
isConst(v) :- cEdge+(u,v), const(u), !sigDep(v)
  inDep(v) :- in(u), cEdge+(u,v)
    UCO(v) :- out(v), !isConst(v), !inDep(v)
```

Observe that this anti-pattern specification introduces three auxiliary predicates. First, it defines a predicate `sigDep(v)` indicating that v is dependent on some other signal. Second, it defines a predicate called `isConst(v)` that evaluates to true if v is *only* dependent on a constant (and hence, it must be constrained to be a constant). Finally, `inDep(v)` checks whether v is dependent on an input signal. Using these predicates, an output signal is considered unconstrained if it is neither (a) constrained to be a constant, nor (b) dependent on an input signal.

Unconstrained Sub-circuit Input (USCI) Recall from Section 3.2 that many templates expect callers to constrain their inputs when instantiated. A component used without any constraints on its inputs is then considered to be suspicious. This can be described in our VDL using the following anti-pattern:

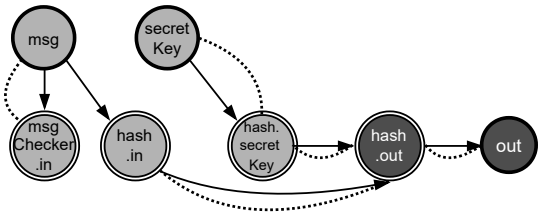

Figure 11: CDG for `ProcessMessage` example shown in Figure 5, which has an unconstrained component input `hash.in`.

```
exDep(v,c) :- sigOf(v, c), cEdge+(u,v),
              !sigOf(u,c)
   USCI(v) :- sig(v), inOf(v,c),
              !isConst(v), !exDep(v,c)
```

Here the anti-pattern introduces an extra predicate, `exDep(v,c)`, which indicates that signal v of component c has a constraint dependency on a signal outside of c. Using this predicate, we define the input signal v of component c to be unconstrained if it is neither constrained to be a constant nor satisfied the `exDep` predicate. Observe that the definition of `USCI` utilizes the same `isConst` predicate defined earlier in the Unconstrained Circuit Output pattern.

Figure 11 shows the CDG for the `ProcessMessage` example in Figure 5. Here, solid edges represent data dependencies, and dashed edges represent constraints. As we can see from the CDG, the input of component `hash` is not constrained because there is no incoming constraint edge to `hash.in`. Hence, our USCI detector will flag this circuit as being potentially vulnerable.

Dataflow-Constraint Discrepancy (DCD) As discussed in Section 3.3, one of the root causes of vulnerabilities is due to semantic discrepancies between witness calculation and constraint generation. Some of these discrepancies manifest themselves as inconsistencies between the data flow and constraint edges in the CDG. Thus, one of our static checkers looks for the following anti-pattern:

```
 nodeDepOn(v,u) :- sig(u), sig(v), dEdge+(u,v)
nodeCnstBy(v,u) :- sig(u), sig(v), cEdge+(u,v)
         DCD(v) :- nodeDepOn(v,u), !nodeCnstBy(v,u)
```

Here, two additional predicates are introduced to model the data flow and constraint relations between two signals explicitly. `nodeDepOn(v,u)` returns true if there exists a (recursive) data dependence from v to u; `nodeCnstBy(v,u)` returns true if there exists a (recursive) constraint dependence between v and u. Thus, we identify a discrepancy on a signal v by checking whether there exists another signal u such that v (recursively) depends on u via data flow but is not constrained by u.

Figure 13 shows the CDG for the `SingleAssignment0` example from Figure 12. Here, signal `out` depends on `a` during witness generation, but it is only constrained by `b` for constraint generation. Hence, there is a discrepancy between the

```
template SingleAssignment0() {
    signal input a;
    signal input b;
    signal output out;
    out <-- a + 1;
    out === b + 1;
}
component main = SingleAssignment0();
```

Figure 12: A code snippet showing a bug caused by dataflow-constraint discrepancy. In particular, the semantics of computing and constraining `out` are different.

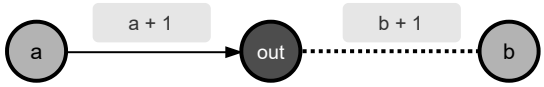

Figure 13: CDG for `SingleAssignment0` example shown in Figure 12.

dataflow and constraint dependencies of v at the CDG level, causing this example to be flagged as potentially vulnerable.

Type Mismatch (TM) There may be semantic discrepancies between witness calculation and constraint generation even when the program does not exhibit the DCD anti-pattern. This is because the CDG is only a coarse abstraction, and identifying a vulnerability may require reasoning more deeply about the underlying semantics of the code. Unfortunately, reasoning precisely about Circom semantics is very challenging because Circom constraints involve non-linear equations over finite fields. This poses a problem because more sophisticated static reasoning often requires querying an SMT solver but solving non-linear equations over large prime fields is intractable for current SMT solvers. Hence, rather than using SMT-based techniques for *precise* reasoning about program semantics, our approach is pragmatic and looks for certain *common* issues that cause a divergence between witness calculation and constraint generation. Our so-called *Type Mismatch (TM)* detector aims to identify one of these patterns.

The key idea behind the Type Mismatch detector is that some of the generated constraints enforce that a signal has a certain type but the witness computation does not, thereby causing a divergence in their semantics. For example, given an array of signals representing an integer in base-k representation, many circuits enforce that the integer representation of x is in the range $[0, N]$ for some N. However, if there is a computation that causes x to fall out of this range, it creates discrepancies between witness calculation and constraint generation. We can detect this divergence using the following anti-pattern:

```
checked(v) :- cLabel(v, _, c), rangeCheck(c).
     TM(v) :-  dEdge+(v, u),  checked(u), !checked(v)
```

Here, the first line line says that a variable is type-checked, denoted `checked`, if it is involved in a constraint that performs a range check. (We omit the definition of `rangeCheck`, as it performs syntactic pattern matching on expressions). According to the TM rule, there is a type mismatch error if there is a dataflow dependence from v to u where u is type checked but v is not.

Other Detectors. In addition to the four detectors explained in detail, ZKAP also implements five other checkers that we briefly summarize:

- **Unconstrained signal (US):** This detector checks for *any* unconstrained signals in the circuit. While unconstrained intermediate signals are not necessarily a bug, they can nonetheless be problematic, so we found it useful to generate warnings for unconstrained signals.

- **Unconstrained sub-circuit output (USCO):** This detector checks for unused component outputs. However, we found that naively checking unused component outputs results in too many false alarms, as some components are only used to constrain their input. Hence, our detector first identifies components whose output is *intended* to be checked, using the following observation: Let φ denote the logical encoding of the circuit in terms of the input and output signals. If $\exists o.\varphi$ is vacuously true, then the output o must be constrained at the call site in order for the component to have any effect on the main circuit. However, since it is intractable to perform an actual logical validity query, our detector uses CDG dependencies to over-approximate the set of relevant outputs.

- **Assignment misuse (AM):** This detector checks for misuses of the assignment operator. In particular, if an expression e is allowed as part of the constraint, developers should use `<==` instead of the assignment operator (`<--`).

- **Non-deterministic dataflow (NDD):** The key observation behind this detector is that it is extremely difficult to correctly constrain signals that are conditionally assigned based on a signal. Most conditionals in Circom are over template variables that are resolved at compile time and require no special handling. Conditionals that involve signals, however, can only be expressed in the computation and must be properly emulated as constraints, which we have found to be error-prone. Hence, this detector looks for conditional assignments that are dependent on a signal.

- **Division-by-zero (DBZ):** We found that one of the causes of computation-constraint divergence is due to a potential division-by-zero issue in assignments. Since assignments involving division are modeled using non-linear multiplication in the constraints, division-by-zero issues also introduce discrepancies between witness calculation and constraint generation. Hence, if the denominator of a division expression used in an assignment is dataflow-dependent

on an input signal, our DBZ detector flags the circuit as potentially vulnerable.

5 Evaluation

In this section, we describe the results of the experimental evaluation of ZKAP, which refers collectively to the detectors described in Section 4. Our evaluation is designed to answer the following key research questions:

- **RQ1**: How does ZKAP compare against CIRCOMSPECT (the only other tool for finding bugs in Circom) in terms of precision, recall, F1 score, and running time?
- **RQ2**: Is ZKAP effective at finding previously unknown bugs in Circom programs?
- **RQ3:** What is the relative importance of the different detectors in terms of finding bugs in real-world ZK circuits?
- **RQ4:** How do the different detectors compare in terms of false positive rate?

All experiments are conducted on a Macbook Pro(R) computer with a M1 Pro CPU and 32G of memory running on macOS 12.5.1.

5.1 Dataset Collection and Statistics

Since there is no available dataset for ZK circuits, we collected our own dataset obtained from 17 different Circom projects found on Github. We chose these projects based on popularity (i.e., number of stars on Github) and the availability of documentation and test cases. Our dataset covers a wide range of applications such as computational libraries, curve pairing, machine learning, gaming, and matrix computation. These projects contain 258 arithmetic circuits, and every benchmark has at least one test case to instantiate the circuit templates. We emphasize that these circuits were *not* used as part of our initial study. As such, neither our taxonomy nor our anti-patterns were influenced by these circuits. Table 2 provides statistics about the projects and circuits used in our evaluation.

Project	#Bench	#LOC	#Sig	#Node
circom-pairing	2	17278	37	1444
circomlib	67	31076	576	8593
circom-ecdsa	16	77092	190	20160
aes-circom	8	1553	75	96092
circom-matrix	12	260	32	233
circom-ml	12	2140	74	8487
darkforest-eth	9	922	148	524
ed25519	27	3033	306	57412
hermez-network	11	2904	485	7076
hydra	3	219	61	790
iden3-core	28	1499	888	5431
keccak256	14	1108	87	37952
maci	34	1735	871	41394
semaphore	1	126	22	99
zk-group-sigs	3	131	38	316
zk-SQL	4	634	102	5100
internal	7	13806	49	2128
Overall	258	155516	4041	293231

Table 2: Statistics of ZKAP's collected benchmark set. Here, #LOC means lines of codes, #Sig means declared signals in the whole project, #Node indicates the total number of CDG nodes across all benchmarks in the project.

	ZKAP	CIRCOMSPECT
FP rate	16.4% (8.6%)	41.4%
FN rate	1.4%	2.1%
Precision	70.9% (82.4%)	48.7%
Recall	96.6%	94.8%
F1 Score	0.82 (0.89)	0.64
Time (s)	7.2	0.2

Table 3: Comparison between ZKAP and CIRCOMSPECT. FP and FN rate indicates the false positive and false negative rates respectively. The data in parentheses is the value computed when excluding two specific circuits in circomlib.

5.2 Comparison Against CIRCOMSPECT

In order to answer our first research question (**RQ1**), we perform a comparison against CIRCOMSPECT[5], an open-source static analyzer developed by TrailOfBits, a leading company in blockchain security. CIRCOMSPECT looks for certain syntactic patterns in the abstract syntax tree and, for example, flags *every* use of the <-- and --> operators as a potential bug.

Table 3 shows the result of our comparison between ZKAP and CIRCOMSPECT in terms of precision, recall, F1 score, and running time. Since calculating recall and precision is difficult without having access to the ground truth labels, we estimate these metrics by manually inspecting 140 of the 258 circuits in our benchmark set. In particular, for each of these 140 circuits, we do a manual security audit to identify as many potential vulnerabilities as we can, and we compute precision and recall with respect to these manually-curated labels.

As we can see from Table 3, ZKAP outperforms CIRCOMSPECT in terms of all metrics except for running time. Notably, ZKAP has an F1 score of 0.82 compared to 0.64 for CIRCOMSPECT, and ZKAP's false positive rate is significantly lower (16.4% vs 41.4%). The only metric where CIRCOMSPECT performs better on is with respect to running time: On average, ZKAP takes a median of 7.2 seconds to analyze a benchmark, whereas the median running time of CIRCOMSPECT is 0.2 seconds. This result is expected because CIRCOMSPECT only performs checks at the syntax level. However, since 7 seconds per benchmark is likely to be quite acceptable in the intended usage scenario of ZKAP, we believe this additional analysis time is more than justified by the huge reduction in

[5] https://github.com/trailofbits/circomspect

the number of false positives.

When investigating the sources of false positives, we found two commonly used circomlib templates, namely `isZero` and `BabyAdd`, which match the non-deterministic dataflow and division-by-zero vulnerability patterns but are nevertheless bug free. These templates are frequently used as components in other templates and, as a result, produce many duplicate false positives. To show a more accurate result, we recomputed the false positive rate while excluding the warnings reported from those two circuits and present the new rate in parentheses in Tables 3 and 4. As we can see in Table 3, the false positive rate of ZKAP is only around 9% if we exclude those circuits, making precision 82% and increasing the F1 score to 0.89.

5.3 Detecting Unknown Vulnerabilities

To answer **RQ2**, we inspected all vulnerabilities reported by ZKAP. In our evaluation on 258 circuits, ZKAP reported a total of 405 warnings. However, many of these warnings have the same root cause because multiple circuits in one project invoke the same buggy component. Hence, ZKAP identifies a total of 34 unique problems. Among these, 32 of them are previously unknown vulnerabilities.

To give readers some intuition about the severity of the vulnerabilities uncovered by ZKAP, we briefly describe two vulnerabilities and explain their consequence.

Vulnerability 1: One of the vulnerabilities found by ZKAP is in the ed25519-circom[6] project, which is used to perform curve operations and signature verification for the Ed25519 digital signature scheme. Since this project serves as infrastructure for other Circom circuits, it is quite security-critical. ZKAP detected a previously unknown unconstrained output signal in one of the key circuits used in this project.

Figure 14 shows the template of the `PointCompress` circuit, which implements a compression algorithm to reduce the size of the public key. The public key is represented as the input signal `P[4][3]`, which is a point on the Ed25519 curve. The output signal is declared as an array of size 256 and represents the compressed public key.

ZKAP detected a previously unknown bug located at lines `16-17`. The intention of the developer here is to constrain `out[255]` to be the parity of signal `mod_x.out[0]`. However, line `17` only constrains `out[255]` to be binary, but it remains unconstrained by `mod_x.out[0]`. Therefore, a malicious user can convince the verifier that signal `mod_x.out[0]` is odd when it is even (and vice versa). According to the project's documentation, this template implements a crucial optimization that is directly applied to two inputs of the overall algorithm. Thus, the vulnerability found by ZKAP exposes a substantial attack surface for the whole system. To fix this problem, the developer must first obtain the binary representation

```
1  template PointCompress(){
2      signal input P[4][3];
3      signal output out[256];
4      ...
5      component modinv_z = BigModInv51();
6      component mod_x = ModulusWith25519Chunked51(6);
7      component mod_y = ModulusWith25519Chunked51(6);
8
9      for(i=0;i<3;i++){
10         modinv_z.in[i] <== P[2][i];
11     }
12     ...
13     component bits_x = Num2Bits(85);
14     bits_x.in <== mod_x.out[0];
15     ...
16     out[255] <-- mod_x.out[0] & 1;
17     out[255] * (out[255] - 1) === 0;
18 }
```

Figure 14: An unconstrained output signal found in ed25519-circom project.

of `mod_x.out[0]` (via circomlib's `Num2Bits` for instance) and then constrain `out[255]` to be the least significant bit.

Vulnerability 2: ZKAP's Type Mismatch (TM) detector also detected a bug in a core circuit of the `iden3` protocol[7]. The `iden3` protocol is a ZK-based access control system. Hence, vulnerabilities in the core circuits can potentially lead to privacy leaks in the protocol.

The vulnerability detected by ZKAP originates in template `verifyExpirationTime`, which is shown in Figure 15. The `verifyExpirationTime` template is responsible for validating that a claim (line `2`) was submitted prior to its expiration date. The expiration of the claim is inferred from the `claim` input signal (lines `9-11`), whereas the submission time is represented by input signal `timestamp` (line `3`). To validate that a claim was submitted on time, `verifyExpirationTime` adds adequate constraints to check that `timestamp` is less than or equal to the claim's expiration (lines `9-15`).

ZKAP reported line `14` as vulnerable because of a missing range check on input signal `timestamp`. To see why this is vulnerable, we first need to dive into template `LessEqThan`, circom-lib's circuit that implements the $\leq$ operator for two signals. Specifically, for template `LessEqThan` to work correctly, both of its inputs must be represented using at most N bits, where N is the template's parameter provided during instantiation. For the instantiation of `LessEqThan` on line `14`, the `verifyExpirationTime` template must ensure that both `timestamp` and `expirationComp.expiration` fit in `252` bits. However, only the latter constraint is enforced (via `getClaimExpiration`'s implementation), but the `timestamp` signal is left completely unconstrained. Moreover, we have already identified instantiations of `verifyExpirationTime` in `iden3` that also omit the range check on signal `timestamp`. Therefore, malicious users can easily bypass this check by providing signals that fall out of the expected range. To fix

[6] https://github.com/Electron-Labs/ed25519-circom

[7] https://github.com/iden3/circuits

```
template verifyExpirationTime() {
    signal input claim[8];
    signal input timestamp;

    component header = getClaimHeader();
    for (var i=0; i<8; i++) {
        header.claim[i] <== claim[i]; }

    component expirationComp = getClaimExpiration();
    for (var i=0; i<8; i++) {
        expirationComp.claim[i] <== claim[i]; }

    component lt = LessEqThan(252);
    lt.in[0] <== timestamp;
    lt.in[1] <== expirationComp.expiration;
    ...
}
```

Figure 15: A type mismatch issue found in Circom core library.

this, the developer must either perform data validation within the `verifyExpirationTime` template or ensure that all users of `verifyExpirationTime` pass inputs in the expected range.

5.4 Effectiveness of Different Detectors

To answer **RQ3** and **RQ4**, we present a more detailed breakdown of the results for each individual detector. The results of this evaluation are shown in Table 4. Here, the first column shows the abbreviated name of the detector, and the next column shows the percentage of real bugs uncovered by each detector.[8] The next two columsn, TP and FP, report the true positive and false positive rate respectively, and the last column show the number of vulnerabilities uncovered by each checker. In what follows, we summarize some key observations.

Interestingly, around 40% of the vulnerabilities are detected by the Division-by-Zero (DBZ) checker. The other checkers that uncover more than 10% of the vulnerabilities include Dataflow-Constraint Discrepancy (DCD) checker, Unconstrained Circuit Output (UCO) detector, unconstrained signal (US) checker, assignment misuse (AM) analysis, and the type mismatch (TM) detector. While the remaining checkers also find bugs in our benchmarks, they each contribute to less than 5% of the uncovered vulnerabilities.

As we can see from Table 4, after excluding the warnings reported from two specific circuits mentioned in Section 5.2, the USCO detector has a surprisingly high false positive rate. Recall from Section 4.4 that the USCO detector needs to identify circuits whose output is intended to be constrained at the call site. However, doing this precisely requires logical reasoning about a quantified formula, which is very expensive and even intractable for existing solvers. Instead, our analysis uses heuristics to identify circuits whose output should be checked at call sites. This strategy results in a large number of false positives but is still meaningful to exhaustively expose potential vulnerabilities.

[8] Note that these numbers do not add up to 100%, as some bugs are flagged by multiple detectors.

Detector	Contribution	FP	#Vul
UCO	16%	0%	9
USCI	4%	0%	2
DCD	12%	0%	7
TM	21%	37%	12
US	24 %	30%	14
USCO	4%	93%	2
AM	14%	27%	8
NDD	5%	90% (25%)	3
DBZ	41%	25% (11%)	24

Table 4: Performance of ZKAP across different detectors. Contribution shows the percentage of real bugs uncovered by each checker. Here, #Vul means how many benchmarks are detected and their vulnerable patterns are confirmed leading to an issue by our manual review. The data in parentheses is the refined data after excluding two specific circuits in circomlib.

6 Related Work

In this section, we survey the closest related work that are relevant to our proposed approach.

ZK programming languages and compilers. Due to the growing demand for zero-knowledge proofs in many application domains, there have been several proposals for new domain-specific languages for ZKPs. While our proposed approach is primarily intended for Circom, there are other DSLs, such as Leo [16], Zokrates [19], Zinc [26], Snarky [27], and CirC [28], for facilitating the construction of zk-SNARK proofs [12]. Since most of these languages operate over arithmetic circuits, similar approaches to the one proposed here could be applied to other zkSNARK languages.

As mentioned earlier, zkSNARKs are one of several zero-knowledge protocols, and a popular alternative is the so-called zk-STARK proof system [10], which stands for Scalable Transparent ARgument of Knowledge. There are also other programming languages targeting zkSTARKs, the most popular of which is Cairo [24], a Turing complete language that allows general computation. Cairo is quite different than zkSNARK languages and is based on Algebraic Intermediate Representation (AIR) rather than R1CS. Hence, the techniques proposed in this paper are not intended for DSLs targeting zkSTARK proof systems.

Bug detection for cryptography There is a large body of work on finding bugs in cryptographic APIs and protocols. For example, the technique proposed by Manuel et al. [20] aims to find bugs in cryptography libraries for Android applications. Later work by Rahaman et al. [32], Luca et al. [30], and Rumen et al. [29] propose more sophisticated and higher-precision techniques aimed at finding crytography API mis-

uses in Java and Android applications. Recent work by Amit et al. [4] systematically evaluates cryptographic misuse detection tools and provides some insights on improving the state-of-the-art in this domain.

There is also is a substantial body of work that tries to *formally verify* cryptographic protocols and APIs [3, 5, 8, 14]. In particular, Almedia et al. developed EasyCrypt [3], a tool to specify and check functional correctness of cryptographic protocols, and used it to write machine checked proof of RSA-OESP. Likewise, Appel et al. [5] generated a machine checkable proof of the correctness of OpenSSL's SHA-256 implementation.

Despite an extensive body of work on verifying and finding bugs in cryptographic applications [7], there is relatively little work on finding bugs in domain-specific languages for ZKPs. To the best of our knowledge, the only existing tool for finding bugs in arithmetic circuits is the CIRCOMSPECT tool [18], which is an open-source tool developed by a leading security auditor. As mentioned earlier, CIRCOMSPECT flags all uses of the assignment operator (<--) as a potential bug and therefore has a high false positive rate. Our approach is guided by a manual study of existing bugs in Circom programs and aims for a pragmatic and extensible framework to catch common vulnerabilities with high precision and recall. However, our approach is neither sound nor complete, and more sophisticated techniques for precise reasoning about Circom would likely require advances in SMT solvers for reasoning about non-linear equations over finite fields.

Program abstractions for bug detection There is a rich body of work on using graph abstractions of programs to catch bugs. Some of these graph abstractions are quite general and domain-agnostic, such as program dependence graphs [23], callgraphs [1], data flow graphs [2] etc. These graph abstractions have found numerous applications for finding security vulnerabilities. As one example, Boyuan et al. [25] use program dependence graphs to detect SSL/TLS vulnerabilities in Ubuntu applications. However, many other techniques aimed at finding security vulnerabilities use domain-specific program abstractions. For example, Mu et al. propose [36] propose a weighted contextual API dependency graph to detect Android malware instances. Similarly, Feng et al. [22] utilize an Android-specific graph abstraction called an *inter-component call graph (ICCG)* to detect Android applications that match semantic malware signatures. Similarly, our proposed circuit dependence graph abstraction is fairly specific to Circom programs, but it is effective for expressing and detecting semantic vulnerability patterns.

Extensible bug detection frameworks. In this paper, we use a Datalog-like domain-specific language for expressing common vulnerability patterns. The key advantage of this approach is to make the framework extensible by building different detectors on top of a common abstraction. There is a rich history of using declarative languages for writing extensible bug detectors in the program analysis literature. For example, systems like bddbddb [34], Saturn [35], Doop [15] etc. all provide declarative Datalog-like interfaces for writing extensible static checkers. There is also prior security analysis frameworks that are based on Datalog-style domain-specific languages. For example, PQL is a program query language for expressing several classes of security vulnerabilities, such as SQL injections, leaked file handlers etc. In the same vein, Apposcopy [22] provides a Datalog-like language for writing semantic malware signatures. Similar to this prior body of work, we propose an extensible framework for building static checkers by writing semantic vulnerability patterns.

7 Discussion

While designing ZKAP, we strived to achieve a good trade-off between expressiveness and scalability. For complex arithmetic semantics that are hard to express precisely, our current static analysis leverages data- and constraint-dependencies to over-approximate the actual semantics, which in theory, can lead to false alarms. However, as discussed in Section 5, our tool achieves a low false positive rate despite our abstraction.

Our study in this paper focuses on Circom, the most prevalent language for writing ZK circuits. While ZKAP was designed based on common vulnerabilities in Circom, many DSLs for zkSNARKS are quite similar in that they provide mechanisms for witness computation and constraint generation. Hence, we believe that similar insights could be used for designing and building static checkers for other zkSNARK languages.

8 Conclusion

We presented ZKAP, the first semantics-based tool for detecting common vulnerabilities in Circom, which is the most popular DSL for constructing ZKP circuits. The design of ZKAP was guided by a manual study of existing Circom vulnerabilities, which allowed us to categorize root causes of Circom bugs into three main classes. Based on the results of this study, we proposed the *circuit dependence graph (CDG)* abstraction of Circom programs for expressing and checking semantic vulnerability patterns. Our proposed framework allows implementing static checkers by writing *anti-patterns* in our Datalog-style vulnerability description language. Using this framework, we implemented a total of nine different detectors and evaluated them on 258 circuits from popular Circom projects on Github. Our evaluation shows that these detectors can identify vulnerabilities, including previously-unknown ones, with high precision and recall.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[2] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. Cryptology ePrint Archive, Paper 2013/316, 2013. https://eprint.iacr.org/2013/316.

[4] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631, 2022.

[5] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), apr 2015.

[6] aztec. Disclosure of recent vulnerabilities. https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities, 2022.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021.

[8] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. Cryptology ePrint Archive, Paper 2006/043, 2006. https://eprint.iacr.org/2006/043.

[9] Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. New privacy practices for blockchain software. *IEEE Software*, 39(3):43–49, 2022.

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, page 46, 2018.

[11] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[12] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association.

[13] David Bernhard and Bogdan Warinschi. Cryptographic voting—a gentle introduction. *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 167–211, 2014.

[14] Timothy M. Braje, Alice R. Lee, Andrew Wagner, Benjamin Kaiser, Daniel Park, Martine Kalke, Robert K. Cunningham, and Adam Chlipala. Adversary safety by construction in a language of cryptographic protocols. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 412–427, 2022.

[15] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009.

[16] Collin Chin. Leo: A programming language for formally verified, zero-knowledge applications. https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-leo.pdf, 2021.

[17] cloudflare. Introducing zero-knowledge proofs for private web attestation. https://www.cloudflare.com/, 2022.

[18] Fredrick Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 09 2022.

[19] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018.

[20] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 73–84, New York, NY, USA, 2013. Association for Computing Machinery.

[21] electriccoin. Zcash counterfeiting vulnerability successfully remediated. https://tinyurl.com/nbu6543n, 2019.

[22] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 576–587, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[24] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo - a turing-complete stark-friendly cpu architecture. *IACR Cryptol. ePrint Arch.*, 2021:1063, 2021.

[25] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V.N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534, 2015.

[26] Matter-Labs. Zinc. https://github.com/matter-labs/zinc, 2022.

[27] o1 Labs. Snarky: Write efficient, beautiful, safe zk-snark code. https://o1-labs.github.io/snarky/, 2022.

[28] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586, 2020. https://eprint.iacr.org/2020/1586.

[29] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto api rules from code changes. *SIGPLAN Not.*, 53(4):450–464, jun 2018.

[30] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1972–1989, 2021.

[31] Polygon. Scalable payments, with zero-knowledge rollups. https://polygon.technology/solutions/polygon-hermez, 2022.

[32] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery.

[33] sigma. Zero knowledge proofs with sigma protocols. https://tinyurl.com/mwupf447, 2022.

[34] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William W. Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 131–144. ACM, 2004.

[35] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363. ACM, 2005.

[36] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1105–1116, New York, NY, USA, 2014. Association for Computing Machinery.