Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs

SHANKARA PAILOOR*, Veridise, USA
YANJU CHEN*, Veridise, USA
FRANKLYN WANG, Harvard University/0xparc, USA
CLARA RODRÍGUEZ, Complutense University of Madrid, Spain
JACOB VAN GEFFEN, Veridise, USA
JASON MORTON, ZKonduit, USA
MICHAEL CHU, 0xparc, USA
BRIAN GU, 0xparc, USA
YU FENG, Veridise, USA
IŞIL DILLIG, Veridise, USA

As zero-knowledge proofs gain increasing adoption, the cryptography community has designed domain-specific languages (DSLs) that facilitate the construction of zero-knowledge proofs (ZKPs). Many of these DSLs, such as Circom, facilitate the construction of arithmetic circuits, which are essentially polynomial equations over a finite field. In particular, given a program in a zero-knowledge proof DSL, the compiler automatically produces the corresponding arithmetic circuit. However, a common and serious problem is that the generated circuit may be underconstrained, either due to a bug in the program or a bug in the compiler itself. Underconstrained circuits admit multiple witnesses for a given input, so a malicious party can generate bogus witnesses, thereby causing the verifier to accept a proof that it should not. Because of the increasing prevalence of such arithmetic circuits in blockchain applications, several million dollars worth of cryptocurrency have been stolen due to underconstrained arithmetic circuits.

Motivated by this problem, we propose a new technique for finding ZKP bugs caused by underconstrained polynomial equations over finite fields. Our method performs semantic reasoning over the finite field equations generated by the compiler to prove whether or not each signal is uniquely determined by the input. Our proposed approach combines SMT solving with lightweight uniqueness inference to effectively reason about underconstrained circuits. We have implemented our proposed approach in a tool called **QED^2** and evaluate it on 163 Circom circuits. Our evaluation shows that **QED^2** can successfully solve 70% of these benchmarks, meaning that it either verifies the uniqueness of the output signals or finds a pair of witnesses that demonstrate non-uniqueness of the circuit. Furthermore, **QED^2** has found 8 previously unknown vulnerabilities in widely-used circuits.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program analysis**; **Automated reasoning**; **Cryptographic protocols**.

*Both authors contributed equally to this research.

Authors' addresses: Shankara Pailoor, spailoor@cs.utexas.edu, Veridise, Austin, USA; Yanju Chen, yanju@cs.ucsb.edu, Veridise, Santa Barbara, USA; Franklyn Wang, Harvard University/0xparc, New York, USA, franklynw2000@gmail.com; Clara Rodríguez, Complutense University of Madrid, Madrid, Spain, clarrodr@ucm.es; Jacob Van Geffen, Veridise, Austin, USA, jsvg@cs.washington.edu; Jason Morton, ZKonduit, State College, USA, jason@zkonduit.com; Michael Chu, 0xparc, New York, USA, michael@0xparc.org; Brian Gu, 0xparc, New York, USA, brian@0xparc.org; Yu Feng, yufeng@cs.ucsb.edu, Veridise, Santa Barbara, USA; Işıl Dillig, isil@cs.utexas.edu, Veridise, Austin, USA.

2475-1421/2023/6-ART168
https://doi.org/10.1145/3591282

Additional Key Words and Phrases: zero-knowledge proofs, SNARKs, program verification

ACM Reference Format:
Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. 2023. Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs. *Proc. ACM Program. Lang.* 7, PLDI, Article 168 (June 2023), 23 pages. https://doi.org/10.1145/3591282

1 INTRODUCTION

Since their introduction, zero knowledge (zk) cryptographic proof systems [Goldwasser et al. 1985] have been used to build several security-sensitive applications such as verifiable computation [fiore and Tucker 2022], anonymous voting [Onur and Yurdakul 2022], and safe whistle-blowing [Jie 2019]. Moreover, in recent years, these systems have seen an explosion in tooling [Bellés-Muñoz et al. 2022; Eberhardt and Tai 2018] and usage [Ben Sasson et al. 2014; TornadoCash 2019a] in blockchain applications because they allow users to create private transactions and scale the blockchain with technologies like zkRollups.

At a high level, the goal of a zk proof system is to allow users to prove statements while using *but not revealing* some secret information. In more detail, these proof systems create two entities: a prover and a verifier. The goal of the prover is to generate a short proof that they know a witness W satisfying a relation $R(I, W)$ for an input I specified by the verifier. The verifier will verify the proof (with high probability) if and only if the prover actually knows a witness W satisfying the relation. We further say the proof system is zero knowledge if the verifier cannot learn anything about W other than the fact that $R(I, W)$ is satisfied.

In most of these proof systems, R is a set of polynomial equations over a finite field and is commonly referred to as a ZK Circuit. Thus, in order to make use of a zero knowledge proof system, users must be able to encode their computation as a set of polynomial equations. In particular, given some computation P that takes input x and outputs y, the developer must craft a set of polynomial equations $R(x, y)$ such that $R(x, y)$ is true if and only if $P(x) = y$. This is a highly non-trivial and error prone task even for domain experts. To simplify this process, the cryptography community has developed languages like Circom [Bellés-Muñoz et al. 2022], Zokrates [Eberhardt and Tai 2018], and Halo2 [Bowe et al. 2019] that allow users to express their intended computation in a somewhat natural way. Then, given a program in such a DSL, the compiler will generate most of the circuit automatically. Nevertheless, even with compiler support, developers still need to manually derive a large number of constraints, as automatically deriving such a constraint system for an arbitrary computation is an intractably hard problem.[1]

To gain some intuition, consider the example in Figure 1. The function `IsZero` (on the left), takes as input a number x and returns a boolean variable y which is true if and only if $x = 0$. Its corresponding encoding as a circuit is shown on the right. This transformation cannot be performed automatically by existing compilers, so the user needs to express this computation directly as polynomial equation. The transformation introduces an existentially quantified variable w and its correctness relies on the mathematical fact that a field element is non-zero if and only if it has a multiplicative inverse.

One particularly dangerous problem that can arise in this context is that the circuit is *underconstrained*, meaning that multiple distinct outputs satisfy the equation for the same input value. In other words, a circuit is underconstrained if the equations do not specify a function. Intuitively, such circuits are problematic because there exist inputs for which it is possible for a malicious user to generate bogus witnesses, thereby causing the verifier to accept a proof that it should not.

[1] In fact, there is no decision procedure for proving $R(x, y) \iff P(x) = y$ for arbitrary computation P.

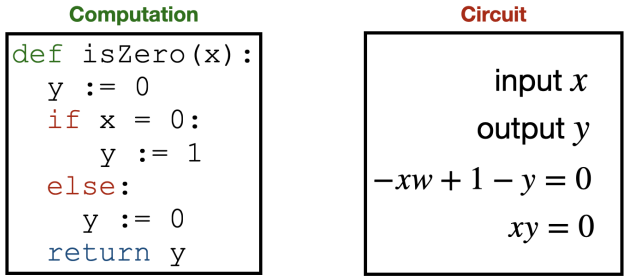

Fig. 1. Example conversion of computation (left) into constraints (right)

Recently, several blockchain hacks have been due to underconstrained circuits and resulted in several million dollars worth of cryptocurrency getting stolen [Aztec 2022; TornadoCash 2019b].

One way to detect underconstrained bugs is to encode the *underconstrained* property as a logical formula and check the satisfiability of the formula using an SMT solver. Given a set of polynomial constraints $P[i, o]$, we can express the underconstrained property using the following formula:

$$\exists i, o_1, o_2.\ P[i, o_1] \wedge P[i, o_2] \wedge o_1 \neq o_2$$

For any circuit represented by P, the circuit is underconstrained if and only if the corresponding logical formula is satisfiable. This solution, while straightforward, does not work well in practice because the resulting SMT encoding is challenging for state-of-the-art solvers, including those that incorporate built-in reasoning about polynomial equations over finite fields [Hader 2022; Ozdemir 2022].

A key observation underlying our technique is that if outputs can be expressed as functions (rather than relations) over input signals, then those outputs are unique. Moreover, we find, in practice, that many circuits contain intermediate and output variables that are easily expressible as functions over inputs. For example, consider the simple set of equations below with input variable *in*, output variable *out*, and intermediate variable *s*:

$$s = 3in^2 + 2$$
$$out = s^2 - 4$$

Here, we can determine that s is constrained by the input *in* since it is expressed as a function over *in*. Similarly, since *out* is expressed as a function of s, we can compose these functions to express *out* in terms of *in* and thus deduce that the circuit is properly constrained. This lightweight reasoning, which we refer to as *uniqueness constraint propagation (UCP)*, can establish key properties of the circuit without making expensive calls to an SMT solver. However, this approach alone cannot solve the underconstrainted bug-finding problem. Specifically, it cannot find pairs of witnesses to prove that a circuit is underconstrained.

In this paper, we pursue a new approach that combines the power of SMT-based semantic reasoning with lightweight UCP. Our technique iteratively invokes UCP analysis to augment the SMT encoding so that the resulting constraints are easier to solve. The workflow of our approach is shown in Figure 2. The input to the UCP engine is the ZK Circuit C and a set of variables K which the algorithm has proven to be fully determined by the inputs. At the start of the algorithm, $K = \emptyset$. The UCP phase then analyzes the equations with the knowledge that variables in K are fully determined and derives a new set of variables $K' \supseteq K$ that it can prove to be uniquely determined by the inputs.

When the UCP phase cannot make further progress, it queries the semantic reasoning engine about the uniqueness of a *particular* variable q. The semantic reasoning engine incorporates the

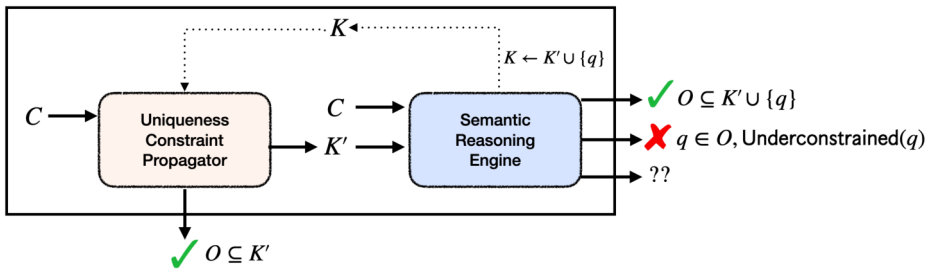

Fig. 2. Overall **QED²** Diagram. O denotes the set of output variables for the circuit C. The check mark means **QED²** proved the circuit was properly constrained. The cross mark indicates **QED²** proved the circuit was underconstrained. ?? means **QED²** was unable to prove the circuit was properly constrained or was underconstrained.

information discovered by the UCP phase as part of its logical encoding and issues a satisfiability query to an SMT solver. If the query can be discharged, then q is added to set K (i.e., variables proven to be unique). This loop continues until one of three conditions is satisfied:

(1) **Verified:** *All the output variables are a subset of K*. In this case, the circuit is proven to be properly constrained. Our approach may terminate with this outcome either after the UCP or the SMT solving phase.
(2) **Refuted:** *The SMT solver finds a counterexample to uniqueness of a query variable q which corresponds to an output of the circuit.* In this case, our approach terminates with a proof that the circuit is underconstrained. Note that our approach can only refute the property after the SMT solving phase.
(3) **Unknown:** *No new unique variables are inferred in either the UCP or SMT phase.* Our approach is not complete and may return unknown. When no new unique variables are inferred in either phase, the algorithm cannot make progress and returns unknown.

We have implemented this algorithm in a tool called **QED²** and evaluated it on ZkBench, a microbenchmark set we collected consisting of 163 real world circuits. Our evaluation shows that **QED²** can verify the uniqueness property for 70% of the benchmarks and found 8 vulnerabilities due to a circuit being underconstrained.

In summary, our paper makes the following contributions.

- We propose a new algorithm that automatically checks whether a given zero-knowledge proof circuit is underconstrained. Our algorithm combines lightweight inference for uniqueness with SMT-based reasoning to generate both proofs and counterexamples.
- We make available ZkBench, a open-source micro-benchmark suite for systematically evaluating the effectiveness of ZK circuits.
- We implement the approach in an end-to-end system called **QED²** and evaluate it on 163 arithmetic circuits from Circomlib. Our evaluation shows that **QED²** can successfully solve 70% of these benchmarks and detects 8 vulnerable templates that can be underconstrained.

2 BACKGROUND

In this section, we provide some background on Zero-Knowledge Proofs and the Circom programming language.

Zero-Knowledge Proofs. A zero-knowledge protocol allows one party, *the prover*, to prove some statement to another party, called *the verifier*, without revealing any secret knowledge. While

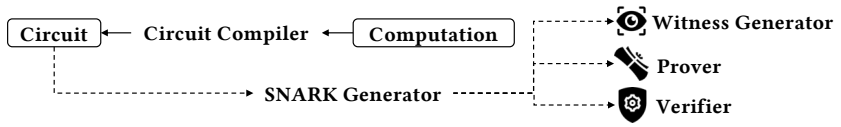

Fig. 3. Compiler workflow

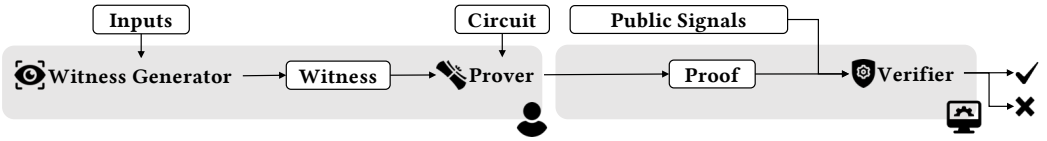

Fig. 4. Usage workflow of Circom. The left half shows the computations done by the user. The right half shows the computations performed by an untrusted party.

there are several different types of zero-knowledge protocols, *zk-snark* (*Zero-Knowledge Succinct Non-interactive ARgument of Knowledge*) protocols [Ben-Sasson et al. 2014] have recently gained popularity due to their succinct proof size and constant verification time. A key property of zk-snark protocols is that a suitable prover and verifier can be automatically generated from an *arithmetic circuit* representation of some computation. An arithmetic circuit takes some input signals which are values in the range $[0, p)$ and performs additions and multiplications modulo a prime number p. The output of every addition and multiplication produces a signal: an *intermediate* signal from intermediate operations, and an *output* signal from the final operation. In particular, given the arithmetic circuit, SNARK compilers construct the prover and verifier by translating the circuit into a set of polynomial equations. For example, compilers for SNARK proof systems like groth-16 [Groth 2016] first construct a set of RANK-1 constraints [Binello 2019], and then transform those constraints into Quadratic Arithmetic Program form [Buterin 2016]. Finally, the underlying cryptographic protocol generates the prover and verifier from these constraints. The details of how the prover and verifier are generated from the constraints are beyond the scope of the paper but we refer the interested reader to [Buterin 2016; Groth 2016; Parno et al. 2013] which explains the construction. In the rest of the paper, we refer to the polynomial equations generated by the compiler as the arithmetic circuit.

Circom Language. Circom is a popular domain-specific language that facilitates the construction of arithmetic circuits. The Circom DSL allows programmers to express computation using `<--` and `-->` for input and output signal assignment, `===` for constraint generation, and `<==`, `==>` to perform both simultaneously. Given some computation expressed in the Circom DSL, an end-user constructs the prover and verifier by first compiling the program to an arithmetic circuit and then using a SNARK Generator (e.g, SNARK-js [Iden3 2018]) to build the prover and verifier from the circuit. Additionally, the Circom compiler produces a so-called *witness generator*, which may be used by end-users to perform the computation and generate a witness that maps signals to values. This workflow of the Circom compiler is shown in Figure 3.

To provide further intuition, Figure 4 illustrates the typical usage pattern for the *output* of the compiler. First, the prover and the witness generator are executed by the user. The prover takes as input a witness, along with the arithmetic circuit, and generates a proof. Next, the verifier is executed by an untrusted party. The verifier takes the proof, in addition to the public inputs, and checks the validity of the proof. When the original Circom program is correct, the verifier approves the proof as intended. If an invalid witness is inserted by a malicious party, the verifier rejects the faulty proof.

```
pragma circom 2.0.0;
include "gates.circom";
include "comparators.circom";

template ValidateDecoding(w) {
    signal input x;
    signal input arr[w];
    signal output success;

    component multiAnd = MultiAND(w);
    component checkEq[w];
    component decoder = Decoder(w);
    decoder.inp <== x;
    for (var i=0; i<w; i++) {
        checkEq[i] = IsEqual();
        checkEq[i].in[0] <== arr[i];
        checkEq[i].in[1] <== decoder.out[i];
        multiAnd.in[i] <== checkEq[i].out;
    }
    success <== multiAnd.out;
}

component main = ValidateDecoding(2);
```

Fig. 5. A Circom program for validating decoder. <== performs signal assignment (<--) together with constraint generation (===)

Underconstrained Bugs. In this paper, we refer to a Circom program P as *underconstrained* if its corresponding circuit C is underconstrained, meaning that there exists an input x and two distinct outputs y, y' such that $C(x, y)$ and $C(x, y')$ are both true. Generally speaking, underconstrained programs are often problematic because they indicate a discrepancy between the computation expressed by P (witness generation) and the corresponding constraints. In particular, if the computation is deterministic, then given an input x, there is only one y such that $P(x) = y$. As such, the corresponding constraints should only evaluate to true for a unique y given an input x. Hence, even though there are rare cases where unconstrained circuits may not correspond to a bug (see Section 8), they often indicate a subtle problem in the underlying program. As shown by recent studies, underconstrained bugs in arithmetic circuits can allow an attacker to forge signatures [Connor 2021], steal user funds [Tornado.cash 2019], or mint counterfeit cryptocoins [min 2019]

To get a sense of the seriousness of underconstrained bugs, we briefly describe a real world attack due to an underconstrained circuit in the smart contract TornadoCash [TornadoCash 2019a] for the Ethereum blockchain. TornadoCash allows users to deposit and withdraw funds without people being able to link the specific deposits to the withdraws. However, because a circuit used in the TornadoCash implementation was underconstrained, attackers were able withdraw all the funds from TornadoCash including funds of other users.

3 MOTIVATING EXAMPLE

In this section, we present an overview of our approach with the aid of the Circom program shown in Figure 5, which is intended to check whether `arr` is a one-hot decoding of `x`. Hence, this Circom program can be used to generate a zero-knowledge proof that `arr` is a valid decoding of `x` without revealing any information about the value of `x` or contents of `arr`. However, this Circom program actually contains a subtle bug that can lead to attacks. In the rest of this section, we explain why the code in Figure 5 is buggy and how our technique can be used to find such bugs.

Understanding the Circom Program. The Circom program from Figure 5 contains two (private) input signals `x` and `arr` and a public output signal called `success`. The output `success` should be

```
1  pragma circom 2.0.0;
2
3
4  template Decoder(w) {
5      signal input inp;
6      signal output out[w];
7      signal output success;
8      var lc=0;
9
10     for (var i=0; i<w; i++) {
11         out[i] <-- (inp == i) ? 1 : 0;
12         out[i] * (inp-i) === 0;
13         lc = lc + out[i];
14     }
15
16     lc ==> success;
17     success * (success -1) === 0;
18 }
```

(a) an underconstrained (buggy) decoder

```
1  pragma circom 2.0.0;
2  include "comparators.circom";
3
4  template Decoder(w) {
5      signal input inp;
6      signal output out[w];
7      signal output success;
8      var lc=0;
9
10     component checkZero[w];
11     for (var i=0; i<w; i++) {
12         checkZero[i] = IsZero();
13         checkZero[i].in <== inp - i;
14         checkZero[i].out ==> out[i];
15         lc = lc + out[i];
16     }
17     lc ==> success;
18 }
```

(b) a properly constrained (fixed) decoder

Fig. 6. Comparison between an underconstrained circuit (a) and its properly constrained (fixed) version (b).

1 when `arr` is a valid one-hot decoding of `x` and 0 otherwise. The program first calls the `Decoder` sub-circuit (lines 12-13) from the Circom standard library, presented in Figure 6(a). The `Decoder` circuit takes as input a value `in` to be decoded and returns `out`, a one-hot decoding of `in` of size `w`. If `in` is larger than `w`, then the `Decoder` should return an array of zeros. Next, it checks that all the elements of `arr` are equal to `out` (lines 14-20). If they are, then success is set to 1, otherwise it is set to 0 (line 20). To assist with value checking, it also calls the `MultiAND` and `IsEqual` sub-circuits from the Circom standard library, where `MultiAND` computes and returns conjunction of provided inputs from `in`, and `IsZero` checks whether given input is 0 or not in a safe way. Note that the correctness of `ValidateDecoding` depends on the correctness of the `Decoder`; however, the `Decoder` circuit has a subtle bug that makes `ValidateDecoding` vulnerable.

Bug in the Program. To understand the bug in `Decoder`, consider its implementation in Figure 6(a). The `Decoder` implementation computes the decoded array `out` and specifies corresponding polynomial constraints to be used in the circuit. For the implementation to be correct, the constraints should match the computation, i.e. any satisfying assignment of the constraints should correspond to a valid execution trace of `Decoder`. Otherwise a malicious prover can trick the verifier into validating a proof that does not correspond to a valid execution of the program.

`Decoder` computes the `out` array as we would expect: for each $i \in [0, w]$ (line 10) it sets `out`$[i]$ to be 1 if and only if `i` $=$ `in` and 0 otherwise (line 11). However, the way `Decoder` generates the corresponding constraints is more subtle, and does so in two phases. First, it adds the constraint that for every entry i in the array, if $i \neq x$ then `arr`$[i] = 0$ (line 12). Note that this assertion is written as the product of two expressions instead of a simple "if-then-else" since the compiler cannot easily translate the latter into polynomial equations. To enforce the constraint that `arr`$[i] = 1$ if and only if $i =$ `x`, Decoder computes the sum of the values of `out` using a local variable `lc` (line 13) and assigns it to the signal `success` (line 16). `lc ==> success` performs an assignment followed by a constraint that `lc` is equal to `success`. Finally, the constraint `success` $*$ `success` $-$ 1 `===` 0 (line 17) ensures that the sum of all the elements is either 1 or 0. Intuitively, if `out` is a valid decoding of `in`, then `success` should be 1 and 0 otherwise.

Underlying Circuit. The bug in this implementation is that the constraints generated do not match the computation. To see why, let us examine the constraints generated by the circom compiler for

the Decoder circuit when $w = 2$:

$$\begin{aligned} \mathsf{inp} \cdot \mathsf{out}[0] &= 0 \\ (\mathsf{inp} - 1) \cdot \mathsf{out}[1] &= 0 \\ \mathsf{out}[0] + \mathsf{out}[1] - \mathsf{success} &= 0 \\ (\mathsf{success} - 1) \cdot \mathsf{success} &= 0 \end{aligned}$$

Note that when inp = 1 then out[1] can be either 0 or 1 and satisfy the constraints. Hence, the following witness would satisfy the constraints:

$$\{\mathsf{inp} \mapsto 1, \mathsf{out}[0] \mapsto 0, \mathsf{out}[1] \mapsto 0, \mathsf{success} \mapsto 0\}$$

even though it does not correspond to a valid execution trace of Decoder. As a result, an attacker can always generate a verifiable proof that they ran the program by setting arr to be an array of zeros regardless of what gets passed in for x.

The Fix. The root cause of this bug is that the value of out[i] is *underconstrained* when i == in as it can be either 0 or 1. The fixed implementation of Decoder, shown in Figure 6(b), properly constrains out[i] by first calling the circuit IsZero (line 12) with inp − i as input (line 13). This is because the IsZero circuit will 1 when inp − $i = 0$ and 0 otherwise. Both finding and understanding this bug is non-trivial, as it requires understanding which equations are generated by the compiler and reasoning about whether the output is uniquely determined by the input.

Our Approach. The goal of the technique proposed in this paper is to automate this reasoning, thereby preventing subtle bugs in zero-knowledge proof protocols. Our proposed technique directly reasons about the polynomial equations generated by the compiler rather than Circom programs themselves. This design choice has several key advantages:

(1) Our technique is language-agnostic, as all zk-snark compilers generate the same mathematical objects.
(2) Our technique can catch bugs caused by the compiler, which are not uncommon [Aleo 2022; Noir 2022].
(3) Our technique avoids false positives that source-level pattern matching techniques would generate.

We have implemented our approach in a tool called **QED^2** to demonstrate these advantages. **QED^2** can *fully automatically* identify the bug in the Decoder circuit of Figure 6(a) in 15 seconds and verify the fixed circuit in Figure 6(b) in under 10 seconds.

4 PROBLEM STATEMENT

In this section, we provide background about arithmetic circuits and introduce the problem statement.

4.1 Background

A finite field $\mathbb{F}$ is a finite set equipped with two binary operators + and × that have identities (0 and 1), inverses (except 0 for ×) and satisfy associativity, commutativity, and distributivity. A *prime field* $\mathbb{F}_p$ is a finite field of size p where p is a prime number. We represent $\mathbb{F}_p$ as the set of integers $\{0, .., p-1\}$ and take + and × to be integer addition and multiplication modulo p respectively.

Given a set of variables X over $\mathbb{F}_p$ we write $\mathbb{F}_p[X]$ to denote the set of polynomials over X whose coefficients are in $\mathbb{F}_p$. We refer to any equation of the form $f = 0$ where $f \in \mathbb{F}_p[X]$ as a polynomial equation over $\mathbb{F}_p[X]$.

*Definition 4.1 (***Arithmetic Circuit)**. Let X be a set of variables over a prime field $\mathbb{F}_p$. An arithmetic circuit $C(X)$ is a set of polynomial equations $\{f_1 = 0, ..., f_n = 0\}$ over $\mathbb{F}_p[X]$.

Example 4.2. The equations on the right-hand side of Figure 1 is an example of an arithmetic circuit for any $\mathbb{F}_p$.

*Definition 4.3 (***Logical Encoding)**. Given an arithmetic circuit $C(X) = \{f_1 = 0, \ldots, f_n = 0\}$, we define its logical encoding $[\![C]\!]$ to be the formula $\bigwedge_{i=1}^{n} f_i = 0$.

In this work, we encode each polynomial equation as a formula in a specialized *theory of finite fields* used by a public fork of the CVC5 SMT solver [Ozdemir 2022]. Polynomial equations can also be encoded in Peano arithmetic by performing addition and multiplication modulo the prime in the field.

Since an arithmetic circuit $C(X)$ encodes some computation over input signals, it is common to partition X into three sets of variables I, W, and O where I, O are the input and output variables respectively, and W denotes the intermediate variables of the computation. In the rest of this paper, we write $C(I, W, O)$ to distinguish the input, intermediate, and output variables.

4.2 Underconstrained Circuits

Our goal in this paper is to demonstrate a *fully automated technique* for proving that arithmetic circuits are properly constrained, i.e. not underconstrained. This subsection introduces terminology to precisely define this problem.

*Definition 4.4 (***Constrained Variable)**. Given a circuit $C(I, W, O)$, let V denote $O \cup W$. We say that a variable $v \in V$ is *properly constrained* by C (or *constrained*, for short), denoted $C \Vdash v$, iff:

$$C \Vdash v \;\equiv\; \mathsf{UNSAT}([\![C]\!] \wedge [\![C]\!][V'/V] \wedge v \neq v')$$

Intuitively, a variable v is constrained if any two satisfying assignments that agree on the input variables also agree on v.

Example 4.5. Consider the following circuit where $I = \{i\}$, $W = \{w\}$ and $O = \{o\}$.

$$w * (w - 1) + i = 0$$
$$w + 1 - o = 0$$

Here o and w are underconstrained because there exist two models of the formula, namely $i \mapsto 0, w \mapsto 0, o \mapsto 1$ and $i \mapsto 0, w \mapsto 1, o \mapsto 2$, that both satisfy the constraints but differ on the values of o, w for the same value of i.

*Definition 4.6 (***Constrained Circuit)**. A circuit $C(I, W, O)$ is constrained if, for every $o \in O$, $C \Vdash o$. Conversely, a circuit is *underconstrained* iff it is not constrained.

5 VERIFICATION ALGORITHM

In this section, we present our verification algorithm for checking whether an arithmetic circuit is properly constrained. As shown in Algorithm 1, the Verify procedure takes as input a circuit C and returns ✓ if C can be proven to be constrained and ✗ if it is provably underconstrained. However, this algorithm is incomplete, so it can also return **?** to indicate unknown.

In Algorithm 1, K represents a set of variables that are provably constrained. Initially, this only includes the input variables I (line 4). Then, the verification algorithm enters a loop that terminates either when (1) all output variables are in K, in which case the circuit is verified (line 7), or (2) the semantic reasoning engine finds a counterexample (lines 16-17), or (3) the algorithm fails to prove any new variables as being constrained.

Algorithm 1 Algorithm for checking whether a circuit is under-constrained

```
1:  procedure Verify(C)
2:      Input: Circuit C(I, W, O)
3:      Output: ✓ if C is correct, ✗ if C is under-constrained, ? otherwise
4:      K ← I
5:      while true do
6:          (Δ, K′) ← UCP(C, K)        ▷ Determine range information and new set of constrained variables
7:          if O ⊆ K′ then return ✓     ▷ Return when all outputs are constrained
8:          V ← (O ∪ W)\K′
9:          while V ≠ ∅ do
10:             v ← ChooseVar(C, V)
11:             V ← V \ {v}
12:             res ← Query(Δ, C, K′, v)     ▷ Query the uniqueness of chosen variable with SMT
13:             if res = ✓ then              ▷ If v is constrained, add to K′ and continue
14:                 K′ ← K′ ∪ {v}
15:                 break
16:             else if v ∈ O ∧ res = ✗ then     ▷ If v is an output and proven unconstrained, return ✗
17:                 return ✗
18:         if K′ = K then                   ▷ Return ? when no progress is made
19:             return ?
20:         K ← K′
```

In each iteration of the outer while loop, there are two possible ways to grow set K: either through a call to the UCP procedure at line 6 or through the loop in lines 9–17. As discussed earlier, the UCP procedure essentially performs lightweight "static analysis" of the circuit to identify as many constrained variables as possible. However, because the UCP procedure is based on a set of incomplete inference rules, it may fail to prove the correctness of the circuit even though it is actually properly constrained. Conversely, it also cannot definitively conclude that C is under-constrained. Thus, if $O \not\subseteq K'$ at line 7, the algorithm enters the inner while loop in which it tries to grow the set of constrained variables by invoking an SMT solver.

In each iteration of this inner loop, the algorithm chooses a variable v at line 10 and attempts to prove that v is constrained with an SMT solver query (the call to Query at line 12). If the SMT solver proves v to be constrained, the algorithm breaks out of the inner loop (line 15) and repeats the process with another chosen variable. On the other hand, if v is an output variable and the SMT solver produces a counterexample, then the algorithm returns ✗ at line 17. Note that unconstrained intermediate variables do not imply that the overall circuit is underconstrained, so the algorithm only returns ✗ if v corresponds to an output variable.

In the remainder of this section, we explain the UCP and Query procedures in more detail. We discuss the heuristic used for query variable selection in Section 6.

5.1 Uniqueness Constraint Propagation

Our uniqueness constraint propagation (UCP) method is presented in Algorithm 2. Given a set of variables K proven to be constrained, the UCP procedure returns a new set of constrained variables $Q \supseteq K$ as well as a mapping Δ that maps each expression in the circuit to a set of constants that it may be equal to. This value information is used both internally by the UCP procedure as well as later in the SMT encoding.

Algorithm 2 first invokes the InferValues procedure (presented in Algorithm 3) to compute a mapping Δ (line 4) which maps each expression e in the circuit to a set of constants $\Omega \subset \mathbb{F}_p$

Algorithm 2 Uniqueness Constraint Propagation

```
1: procedure UCP(C, K)
2:     Input: Circuit C, variables proven to be constrained K
3:     Output: Constrained variables Q, range information Δ
4:     Δ ← InferValues(C)
5:     Q ← K
6:     while true do
7:         K' ← {v ∈ Vars(C) | Δ, C, Q ⊨ v}      ▷ Apply inference rules to find constrained variables
8:         if K' = ∅ then return (Δ, Q)           ▷ If no progress is made, return
9:         Q ← Q ∪ K'
```

$$\frac{x \in K}{K \Vdash x}\ (\textsc{Var}) \qquad \frac{\text{Constant}(c)}{K \Vdash c}\ (\textsc{Const}) \qquad \frac{K \Vdash e_1 \qquad K \Vdash e_2 \qquad \odot \in \{+, \times\}}{K \Vdash e_1 \odot e_2}\ (\textsc{Op})$$

Fig. 7. Expression rules for uniqueness propagation

such that Ω *over-approximates* the values of e that can satisfy the constraints in the circuit. The algorithm then enters a loop in which it repeatedly infers new constrained variables using the set of inference rules shown in Figure 8 (explained later). Since the inference of new variables can trigger the application of more inference rules, these rules are applied until a fixpoint is reached. In what follows, we first explain the rules for inferring new constrained variables and then turn our attention to value inference.

Inference of Constrained Variables. The inference rules used in the UCP procedure are presented in Figure 8 and derive judgments of the following form:

$$\Delta, C, K \Vdash x$$

The meaning of this judgment is that, under the assumption that all variables in K are constrained and the range information in Δ is correct, then variable x is constrained in circuit C (i.e., $C \Vdash x$). These rules make use of the (simpler) auxiliary judgment shown in Figure 7, so we start by explaining them first.

According to the Var rule in Figure 7, any variable in K is constrained. The second rule, called Const, states that all constants are also constrained. The last rule, labeled Op, infers whether a more complex expression is constrained. In particular, given an expression e of the form $e_1 \odot e_2$, e is constrained if both e_1 and e_2 are constrained. Note that this rule can be recursively applied to determine whether an arbitrary expression is constrained.

Next, the inference rules in Figure 8 correspond to common, representative *patterns* of equations in arithmetic circuits that enable us to propagate uniqueness constraints to new variables. We describe each rule in more detail below:

(1) (Assign) If the circuit contains an equation of the form $cx - e = 0$, and we have inferred e is constrained and c is non zero, then we can infer x is constrained since the equation can be rewritten to $x = c^{-1} \times e$. Such equations appear frequently in circuits since zk compilers generate such constraints for nearly every assignment.

(2) (Base-Conv) At a high level, this rule states that if x is constrained, then so is its base-c encoding. The premise $\forall i \in [0, n].\ \Delta(y_i) \subseteq [0, c-1]$ ensures that $[y_0, \ldots, y_n]$ is a valid base-c encoding of x, and the premise $y_n < p/c^n - 1$ ensures that $\sum_{i=0}^{n} c^i \times y_i$, when interpreted as an integer, cannot be larger than p. To see why the latter is important, suppose $p = 5$,

$$\frac{\Delta, C, K \models e \qquad \text{Constant}(c) \qquad c \neq 0 \qquad c \times x - e = 0 \in C}{\Delta, C, K \models x}\ (\textsc{Assign}) \qquad \frac{A\vec{x} - \vec{b} = 0 \subseteq C \qquad \Delta, C, K \models \vec{b} \qquad A \in \mathbb{F}_p^{n \times n} \wedge \det(A) \neq 0}{\Delta, C, K \models \vec{x}}\ (\textsc{BigInt-Mul})$$

$$\frac{\sum_{i=0}^{n} c^i \times y_i - x = 0 \in C \qquad \Delta, C, K \models x \qquad y_n < \frac{p}{c^n} - 1 \qquad c > 1 \qquad \forall i \in [0, n].\ \Delta(y_i) \subseteq [0, c-1]}{\Delta, C, K \models y_i \quad \text{for } i \in [0, n]}\ (\textsc{Base-Conv})$$

$$\frac{\sum_{i=0}^{n} y_i = e \qquad \Delta, C, K \models e \qquad \Delta, C, K \models x \qquad \forall i \in [0, n].\ y_i \times (x - i) = 0 \in C}{\Delta, C, K \models y_i \quad \text{for } i \in [0, n]}\ (\textsc{All-But-One-0})$$

Fig. 8. Equation rules for uniqueness propagation. We use the notation $\Delta, C, K \models \vec{b}$ as shorthand for $\Delta, C, K \models b_1, \ldots, \Delta, C, K \models b_n$ where b_i is an element of $\vec{b}$.

$n = c = x = 2$ and $x = y_0 + 2y_1 + 4y_2$. Then the distinct assignments $[y_0 \mapsto 0, y_1 \mapsto 1, y_2 \mapsto 0]$ and $[y_1 \mapsto 1, y_1 \mapsto 1, y_2 \mapsto 1]$ both satisfy the equation. We note that zk compilers (or programmers) frequently add such constraints to encode converting a field element into a base c representation; this conversion allows one to encode comparison operators like inequalities over fixed-width integers into polynomial equations over prime fields.

(3) (BigInt-Mul) If our circuit has a set of equations matching the pattern $A\vec{x} - \vec{b} = 0$ where A is a square, invertible matrix (i.e. $\det A \neq 0$), and $\vec{b}$ is constrained, then we can conclude $\vec{x}$ is constrained. To see why, if A is invertible, then each x_i in $\vec{x}$ is equal to $A_i^{-1} \cdot b$ where A_i^{-1} denotes the i^{th} row of A. Since a linear combination of constrained variables is constrained we can conclude each x_i is constrained. Constraints matching the pattern $A\vec{x} - \vec{b} = 0$ are most commonly generated to encode performing multiplication over big integers (larger than p) in base-c for some $c > 1$. Such computation is especially common in cryptographic schemes such as ECDSA [Johnson et al. 2001] or BLS [Boneh et al. 2004].

(4) (All-But-One-0) This rule formalizes the intuition that variables y_i are set to 0 precisely when some $x \neq i$. This rule is useful for cases like the `Multiplexer` circuit, where output variables can be expressed as piece-wise functions over constrained variables. To understand why this rule is sound, consider the piece-wise function over x that expresses each y_i: `if` $i = x$ `then` e `else` 0. Since we can express each y_i this way, then if x is constrained, so is each y_i.

Soundness proofs of all of these rules are provided in the appendix of the extended version of this paper [Pailoor et al. 2023].

Inference of Variable Values. Recall that the Base-Conv rule from Figure 8 can only be applied if certain variables are within a range. Thus, in order to effectively propagate uniqueness constraints, our method also needs to infer possible values that each variable can take; this is done via the InferValues procedure presented in Algorithm 3. Given an arithmetic circuit C, InferValues infers a set of possible values for every arithmetic expression among C's equations. The algorithm initializes Δ to map every expression in C to the set of all field elements (lines 4-5). It then enters a loop (line 6) and applies the rules in Figure 10 to infer more precise sets of values (line 7). Since

Algorithm 3 Value Inference

1: **procedure** InferValues(C)
2: **Input:** Circuit $C(X)$
3: **Output:** Value Mapping Δ
4: $E \leftarrow \text{Exprs}(C)$
5: $\Delta \leftarrow \{e \mapsto \mathbb{F}_p \mid e \in E\}$ ▷ Initialize ranges
6: **while** true **do**
7: $\Delta' \leftarrow \{e \mapsto \bigcap \Omega \mid \Delta, C \vdash e : \Omega,\ e \in E\}$ ▷ Apply value inference rules to narrow ranges
8: **if** $\Delta = \Delta'$ **then return** Δ ▷ Return when no progress is made
9: $\Delta \leftarrow \Delta'$

$$\frac{}{\Delta \vdash x : \Delta(x)}\ (\textsc{Var}) \qquad \frac{\text{Constant}(c)}{\Delta \vdash c : \{c\}}\ (\textsc{Const}) \qquad \frac{\Delta \vdash e_1 : \Omega_1 \qquad \Delta \vdash e_2 : \Omega_2}{\Delta \vdash e_1 \odot e_2 : \{v_1 \odot v_2 \mid (v_1, v_2) \in \Omega_1 \times \Omega_2\}}\ (\textsc{Op})$$

Fig. 9. Rules for value inference over expressions. $\odot \in \{+, \times\}$

$$\frac{\Delta \vdash e : \Omega \qquad c \neq 0 \qquad c \times x - e = 0 \in C}{\Delta, C \vdash x : \{v \times c^{-1} \mid v \in \Omega\}}\ (\textsc{Assign})$$

$$\frac{\prod_{i=1}^{n}(x - c_i) = 0 \in C}{\Delta, C \vdash x : \{c_1, \ldots, c_n\}}\ (\textsc{Root}) \qquad \frac{\sum_{i=0}^{n} c^i \times y_i - x = 0 \in C \qquad c > 1 \qquad \forall i \in [0, n].\ \Delta(y_i) \subseteq [0, c-1]}{\Delta, C \vdash x : \{v \in \mathbb{F}_p \mid 0 \leq v < c^{n+1}\}}\ (\textsc{Base-Conv})$$

Fig. 10. Rules for value inference over equations.

inferring the values of one variable may allow us to more precisely constrain the values of another, we apply these rules in a loop until reaching a fixed point (line 8).

We now describe the rules in Figure 10 which derive judgements of the form

$$\Delta, C, \vdash e : \Omega$$

This judgement means that, given a mapping Δ from expressions in C to a set of values, expression e can only take on values in Ω. Similar to our UCP propagation rules, the rules in Figure 10 make use of helper rules presented in Figure 9 so we describe them in more detail first.

The Var and Const rules state that the values of a variable x and constant c given Δ must lie in $\Delta(x)$ and $\{c\}$ respectively. The last rule, labeled Op, infers values for more complex expression of the form $e_1 \odot e_2$. In particular, the set of values obtained is the result of applying $v_1 \odot v_2$ for every possible value v_1 for e_1 and v_2 for e_2.

Finally, the inference rules in Figure 10 correspond to equations that match certain common syntactic patterns and allow us to infer more precise values for variables other than $\mathbb{F}_p$. Given an equation $c \times x - e = 0$ where $c \neq 0$, the Assign rule tells us that the values of x can only be of the form $v \times c^{-1}$ where v is a possible value of e and c^{-1} denotes the multiplicative inverse of c mod p. The next rule, Root, states that if we have an equation of the form $\prod_{i=1}^{n}(x - c_i) = 0$ in our circuit, then x can only be values among $\{c_1, \ldots, c_n\}$. Lastly, Base-Conv allows us to infer a more precise

upper bound on the value of a variable x if it has a valid base-c encoding. In particular, if x has a valid base-c encoding using n digits, then we can infer that x can be at most $c^{n+1} - 1$.

Soundness proofs of these rules are also provided in the appendix of the extended version of this paper [Pailoor et al. 2023].

5.2 SMT-Based Reasoning

Algorithm 4 QUERY Procedure

1: **procedure** QUERY(Δ, C, K, v)
2: **Input:** Value information Δ, circuit $C(I, V = O \cup W)$, constrained variables K, query variable v
3: **Output:** ✓ if v is correct, ✗ if v is under-constrained, **?** otherwise
4: $\phi \leftarrow \bigwedge_{u \in K}(u = u')$
5: $\psi \leftarrow \bigwedge_{w \in Dom(\Delta)} \bigvee_{(l,u) \in \mathcal{P}(\Delta(u))} l \leq w \leq u$
6: $res \leftarrow \text{VALID}((\phi \wedge \psi \wedge [\![C]\!] \wedge [\![C]\!][V'/V]) \implies v = v')$
7: **return** res

Recall that, when the UCP procedure cannot infer any new constrained variables, our verification algorithm queries the SMT solver. The procedure for querying a variable v is presented in Algorithm 4. The basic idea is to encode two copies of the circuit, one over variables I, V and one over variables I, V' as $\Phi \equiv [\![C]\!]$ and $\Phi' \equiv [\![C]\!][V'/V]$ respectively and then check whether $\Phi \wedge \Phi'$ implies that $v = v'$ for the given query variable v. If this is the case, the query variable is indeed properly constrained. However, since the variables in K have already been proven to be constrained, the SMT query strengthens the antecedent of the implication with the following formula:

$$\phi \equiv \bigwedge_{u \in K} u = u'$$

Finally, the query to the SMT solver also utilizes the value information obtained through the lightweight analysis presented in Figure 9. In particular, for each variable w in the domain of Δ, we first partition the possible values of w into a set of intervals of the form (l, u) — that is,

$$(l, u) \in \mathcal{P}(\Delta(w)) \iff (\{l, l+1, \ldots, u\} \subseteq \Delta(w) \wedge l - 1 \notin \Delta(w) \wedge u + 1 \notin \Delta(w))$$

Each of these intervals $(l, u) \in \mathcal{P}(\Delta(w))$ is then encoded as the constraint $l \leq w \leq u$ (simplifying to $w = l$ if $l = u$), and the value of each w is a disjunction over all intervals in $\mathcal{P}(\Delta(w))$, as shown in line 5 of Algorithm 4. This constraint ψ is also added to the antecedent of the implication when querying whether $v = v'$ is implied by the circuit encoding.

6 OPTIMIZATION AND IMPLEMENTATION

We have implemented our proposed approach in a tool called **QED**^{2} written in Racket [Flatt and PLT 2010]. **QED**^{2} incorporates a public fork of cvc5 [Barbosa et al. 2022] with a custom decision procedure for solving polynomial equations over finite fields as its backend SMT solver.

Query Variable Selection. In Algorithm 1, we use a procedure called CHOOSEVAR to select the next variable to query. Our implementation uses the following heuristic for choosing variables:

$$\text{CHOOSEVAR}(C, V) = \arg\max\{u \mapsto \sum \text{count}(f, u) \mid f \in C, u \in V\},$$

CHOOSEVAR selects the variable $u \in V$ that has the most number of terms in C that are linear in u. To do so, it uses a helper function, $\text{count}(f, u)$ that computes the number of terms f that are linear in u. The intuition behind this heuristic is twofold: variables that appear often in the circuit are more highly restricted, and the SMT solver can more easily reason about linear terms.

Counterexample Construction. The finite field solver used by **QED²** uses a semi-decision procedure for proving unsatisfiability. When the solver cannot prove that the formula is unsatisfiable, it attempts to explicitly construct a model by performing stochastic search over the prime field. This model construction can be expensive as its expected runtime is exponential in the number of variables in the formula. Hence, when **QED²** observes that the model construction phase is taking too long, it uses a custom strategy to construct counterexamples.

QED²'s counterexample generation strategy starts by decomposing the circuit into sub circuits $C_1, ..., C_n$ using debug information from the compiler. This is feasible because circuits are often built using other circuits (as we see from Figure 5). **QED²** then builds a counterexample M incrementally by iterating over each C_i and querying the solver whether C_i is constrained (using the counterexample M constructed so far). If the solver says C_i is constrained, **QED²** moves on to C_{i+1}. Otherwise, if C_i is underconstrained, the solver returns a model m and **QED²** updates M to $M \cup m$ before proceeding to the next circuit. If the solver returns unknown for any C_i, **QED²** also continues to the next circuit. At the end of this procedure, if all C_i's are proven to be constrained, **QED²** returns ✓. On the other hand, if M is a complete model, **QED²** returns ✗ and **?** otherwise. Because each query to the solver is often exponentially smaller than the query for the full circuit, we found this compositional counterexample construction strategy to be helpful in a few cases.

7 EVALUATION

In this section, we describe the results for the experimental evaluation, which are designed to answer the following key research questions:

- **RQ1**: How effective is **QED²** in verifying real-world circuits?
- **RQ2**: Is **QED²** useful for detecting unknown vulnerabilities in real-world circuits?
- **RQ3**: What is the relative importance of uniqueness constraint propagation (UCP), and how important is it to use an SMT solver?

Benchmarks. We evaluated **QED²** on circuits built from Circom programs as Circom is one of the most popular languages for writing ZKPs and powers applications that manage millions of dollars on the blockchain. In particular, we gathered circuits from `circomlib`[2], the standard library for Circom. Circomlib is a set of circuit *templates* like the `Decoder` example in Figure 6(a). The templates themselves are not circuits but can become circuits by setting their template parameters to a constant. For example, line 23 in Figure 5 builds a circuit from the template `ValidateDecoding` by initializing w to 2. It is critically important that the templates from Circomlib are properly constrained as they are used in nearly every Circom application.

With this in mind, we collected two representative benchmark sets from Circomlib:

- **The `circomlib-utils` benchmarks**. These benchmarks consist of circuits built from 59 *utility* templates. These templates help developers perform fixed-width integer computation like range checks or integer arithmetic. It also contains some commonly used blockchain primitives like Merkle-tree verification and ZKP friendly hash functions like the Poseidon hash [Grassi et al. 2021]. Since these utility functions are instantiated in many different ways by applications, we constructed the circuits by randomly selecting parameters for the templates.
- **The `circomlib-core` benchmarks**. This contains 104 circuits collected from `circomlib`, but with a focus of a more in-depth coverage of different instantiations of the 50 most security-critical templates in the library. We generated the circuits from these templates by instantiating them with the most widely used values. For example, in this benchmark set, we instantiated the

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

Table 1. Key statistics of the benchmark sets: number of circuits in each benchmark set, average size of each circuit, and average number of output variables per circuit. The size of a circuit is the number of equality constraints that describes the circuit.

Benchmark Set	# circuits	Avg. # constraints	Avg. # output signals
circomlib-utils	59	352	10
circomlib-core	104	6,690	32
All	163	4,396	24

Table 2. Main experimental results. For each benchmark set, we categorize circuits based on size: small (< 100 constraints), medium ($[100, 1000)$), and large (≥ 1000). The top half of the table describes information about the types of variables and constraints in each sub-group of benchmarks. The bottom half of the table describes performance metrics per sub-group. Overall, **QED^2** successfully solves 70% (114 / 163) of all benchmarks.

Benchmark		circomlib-utils				circomlib-core				overall
Size		small	medium	large	overall	small	medium	large	overall	
Avg. Variables (#)	in	5	76	103	21	27	30	167	55	43
	out	2	2	103	10	11	81	41	32	24
	witness	14	318	3,465	342	11	391	34,102	6,651	4,368
	total	20	396	3,671	374	49	502	34,310	6,738	4,435
Avg. Constraints (#)	linear	7	170	2,159	209	5	198	28,002	5,432	3,541
	non-linear	7	149	1,413	143	12	274	6,189	1,258	854
	total	15	319	3,571	352	17	472	34,190	6,690	4,396
Total (#)		47	7	5	59	61	23	20	104	163
Avg. Time (s)		9s	10s	9s	9s	8s	13s	18s	10s	9s
✓ (#)		36	4	3	43	44	10	4	58	101
✗ (#)		6	0	0	6	7	0	0	7	13
Solved (%)		89%	57%	60%	83%	84%	43%	20%	63%	**70%**

template to perform the Pedersen hash[Pedersen 1991] with values that would be used in most smart contracts.

Table 1 shows key statistics of the collected benchmarks. Overall, the average number of constraints in `circomlib-utils` benchmark is 352, with 10 output signals on average. The `circomlib-core` benchmarks are more challenging: they contain 6,690 constraints and 32 output signals on average.

Experimental Setup. All experiments are conducted on an Amazon EC2 t3a.xlarge instance. The time limit for each benchmark is 10 minutes and the memory limit is 32GB.

Evaluation Metrics. We use the following two key metrics to evaluate our tool:

- **Solved benchmarks:** Recall that **QED^2** has three possible outcomes, namely, verified, refuted, or unknown. In particular, **QED^2** can return unknown either because it exhausts the allocated resource limit or reaches a fix point without finding a proof or counterexample. Hence, one of our key evaluation metrics is the percentage of benchmarks that **QED^2** can solve successfully, meaning that it returns an answer other than unknknown.
- **Solving time:** In addition to the percentage of solved benchmarks, we also measure the time it takes **QED^2** to return an answer for the benchmarks that it can successfully solve.

7.1 Main Results

Table 2 summarizes the main results of our evaluation on the `circomlib-utils` and `circomlib-core` benchmarks. This table classifies circuits into three categories as either small, medium, or large based on the number of constraints (denoted by C):

- **Small:** $C < 100$.

- **Medium:** $100 \leq C < 1000$.
- **Large:** $1000 \leq C$.

As we can see from Table 2, **QED²** can successfully solve 70% of the benchmarks, meaning that it finds either a proof or counterexample. As expected, benchmarks in the large category are much harder to solve, which explains why the overall success rate for `circomlib-core` is lower than that of `circomlib-utils`. Also as expected, **QED²**'s success rate is higher for circuits in the small category for both benchmark sets. Finally, we note that among the benchmarks that **QED²** can solve, its runtime is fairly fast, including the large circuits in `circomlib-core` which take an average of 18 seconds to solve. This is because, on those benchmarks, our UCP engine is able to quickly detect that most of the output signals are unique, and so our semantic reasoning engine only has to check at most a handful of signals.

Among the circuits that **QED²** can solve, **QED²** returns verified for the vast majority (89%). This result is expected since many of circuits that are part of `circomlib-utils` and `circomlib-core` are written by cryptographers who are also Circom experts. However, there are 13 circuits for which **QED²** produces counterexamples, meaning that these circuits are provably underconstrained. Looking at the results closer we found that these circuits were generated from 8 distinct templates. Since these circuits belong to widely-used libraries that are used by other clients, this evaluation shows that **QED²** can find critical vulnerabilities in real-world circuits. For example, we found an underconstrained bug in the EdwardsToMontgomery circuit, which converts points on an Edwards curve [Boudabra and Nitaj 2019] to their corresponding point on a Montogmery curve [Costello and Smith 2017], which is used frequently in the Circomlib implementation of the Pedersen Hash function [Pedersen 1991].

Failure Analysis. For the 49 circuits that **QED²** could not solve, we manually analyzed the root causes of the failures. Among the 5 circuits that **QED²** could not solve in the small category in circomlib-utils, all failures were due to timeouts caused by two complex sub-circuits. One of them is the BabyAdd circuit which implements elliptic curve addition [Baylina 2021]. In order to show that this circuit is properly constrained, the semantic reasoning engine must essentially prove Theorem 3.3 from [Bernstein and Lange 2007], which is an extremely difficult task for an automated tool. However if we were to add this theorem as an axiom into our tool, **QED²** can quickly prove the 5 additional circuits in the circomlib-utils (small) category as well as one additional circuit in the medium category of circomlib-utils. A similar problem arises in several circomlib-core circuits due to a shared sub-circuit called Num2BitsNeg. This circuit encodes a prime field element as a bitvector and takes the bitwise negation of the bitvector. In this case, the bitvector conversion generates constraints where the polynomials have very large degrees as well as coefficients ($> 10^5$). The underlying SMT solver uses a Groebner Basis engine to check if the queries are unsatisfiable, but the runtime of Groebner basis computation is very sensitive to the degree and coefficients of the polynomials, so the solver times out. In particular, among the remaining medium and large circuits that **QED²** times out on, we observed that it is also due to the Groebner basis computation.

7.2 Ablation Study

Since there is no prior published research on finding under-constrained zk circuits, we are not able to compare **QED²** against existing baselines. Instead, we present the results of an ablation study to assess the relative importance of SMT-based reasoning as well as our proposed uniqueness constraint propagation technique. In particular, we compare **QED²** against the following two ablations:

- **QED²**-Ucp is a variant of **QED²** that *only* performs uniqueness constraint propagation using Algorithm 2. However, it does not invoke the SMT solver either for verification or refutation.

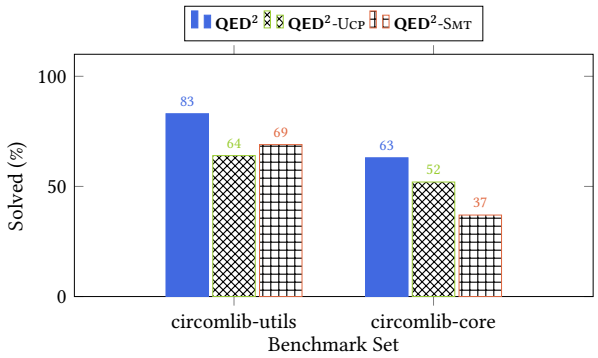

Fig. 11. Comparison between **QED²** and its ablations. **QED²**-UCP performs uniqueness constraint propagation but does not utilize the SMT solver. On the other hand, **QED²**-SMT only uses the SMT solver but does not perform lightwight analysis to infer constrained variables.

- **QED²**-SMT is a variant of **QED²** that only performs symbolic reasoning through an off-the-shelf SMT solver. As mentioned in Section 6, we leverage the state-of-the-art finite field solver from CVC5 [Ozdemir 2022]. This ablation does not leverage uniqueness constraint propagation to perform lightweight inference of constrained variables.

The results of this ablation study are presented in Figure 11. As we see from this bar chart, **QED²** can solve more benchmarks on than the other two ablations on both the `circomlib-utils` and `circomlib-core` benchmarks. For the `circomlib-utils` benchmarks, **QED²**-SMT performs slightly better than **QED²**-UCP, but both are considerably worse than the full version of **QED²** that incorporates both the UCP engine as well as the SMT solver. For the larger `circomlib-core` benchmarks, **QED²**-UCP outperforms **QED²**-SMT by a fairly large margin. This is because many of the circuits in the `circomlib-core` benchmark suite can be verified using lightweight uniqueness inference, but, because these circuits are fairly large, the SMT solver is unable to successfully decide satisfiability of the corresponding constraints.

8 DISCUSSION

Intentionally Underconstrained Circuits. As we mentioned in Section 1, an underconstrained circuit could allow a malicious user to verify a proof that the circuit programmer did not intend to get verified. However, there are cases where the programmer intends for a circuit to be underconstrained. For example, suppose the programmer constructs a circuit which states that the output must be the square root of the input (if it exists). They may express this circuit with a single equation: $\text{out} = \text{in}^2$. This circuit is underconstrained as there may be two roots for a single input; however, the developer may intend for this behavior to give users flexibility in the choice of root. Such cases of intentional nondeterminism occasionally occur when a circuit is designed to be used by other circuits and very rarely occur for top-level circuits. Thus, while underconstrained circuits do not necessarily *always* indicate a bug, we believe it is important for developers to know whether or not their circuits are properly constrained.

Bugs in ZK Circuits. This work focuses on detecting a *specific* class of bugs in ZK circuits; however, there are other ways (beyond being underconstrained) that ZK circuits could be problematic. For instance, a ZK circuit could also be *overconstrained*, meaning that the constraint system is inconsistent for a given input. Intuitively, overconstrained circuits can result in denial of service attacks, as

the verifier can reject proofs that should actually be accepted. In practice, however, overconstrained circuits are much less common and considered less security critical than underconstrained circuits.

More generally, both underconstrained and overconstrained bugs can be viewed as symptoms of equivalence violations, where the witness generation semantics do not agree with the constraints. In particular for a given program P that takes input x and produces y, its constraints C should be such that $C(x, y)$ is true iff $P(x) = y$. We believe that proving and finding violations of equivalence bugs in ZK circuits is an important and challenging problem, and we see the techniques proposed in this paper being a first step in achieving this goal. Finally, just like in traditional programs, the computation P expressed in a ZK program could have functional correctness bugs as the implementation may not match the developer's intentions. While finding such functional correctness bugs is also very important, this paper only focuses on a specific (but common) class of problems caused by underconstrained circuits.

9 RELATED WORK

ZK Programming Languages & Compilers. Due to the increasing importance of zero-knowledge proofs in many application domains, there have been several proposals for new programming languages and compilers that target this domain. Similar to Circom, Zokrates [Eberhardt and Tai 2018], Zinc [Matter-Labs 2022], Snarky [o1 Labs 2022], and Leo [Chin 2021] are other domain-specific language that facilitate the construction of zk-snarks. Because our approach operates over finite field equations generated by the compiler, our proposed technique can be used to reason about bugs in *all* of these languages, including those introduced by the compiler. CirC [Ozdemir et al. 2020] is a recent effort that aims to provide a unified compiler infrastructure for all DSLs that compile down to arithmetic circuits. In this way, CirC is somewhat akin to LLVM but targeted towards ZKP DSLs that produce existentially quantified circuits (EQCs). Our proposed technique can also be used to analyze circuits that are generated by a CirC-based compiler. All of the aforementioned programming languages produce mathematical objects that fall in the class of Rank 1 Constraint System (R1CS). There are also languages such as Halo2 [Bowe et al. 2019] that produce a more general class of polynomial equations over finite fields. Under-constrained output variables are also problematic in that setting, and our technique can be applied here. However, because our uniqueness constraint propagation rules are primarily targeted for R1CS constraints, our UCP algorithm may not work as well for languages like Halo2.

Another popular programming language for the verifiable computing domain is Cairo [Goldberg et al. 2021], which is a Turing complete language that allows general computation. Unlike SNARK-based languages like Circom, Cairo programs do not get compiled to polynomial constraints over finite fields. They are instead based on a different protocol called STARKs (Scalable, Transparent, Arguments of Knowledge) with a different type of prover and verifier. In particular, Cairo consists of a single prover and verifier for all Cairo programs. The prover takes as input an execution trace of a Cairo program and generates a proof asserting that the trace is a valid execution of a Cairo program. The trace consists of (1) the program input, (2) a memory function mapping memory cells to concrete values (including the program's bytecode), (3) a value N indicating the number of instructions executed, (4) a sequence of N+1 state transitions. Intuitively, the memory function can be viewed as a mapping from signals (program variables) to values where the last memory cell written is the output signal of the program. The prover encodes the trace as a sequence of polynomial equations and then uses the STARK protocol to transform the equations into a short proof. We believe **QED^2** can be used to find underconstrained bugs after the prover constructs the polynomial equations. In particular, **QED^2** can check if for the same input, bytecode, and sequence of states, whether a different memory function (with a distinct output and fixed bytecode layout) can satisfy the polynomial constraints.

Formal Methods for Cryptography. There is a rich body of work on applying formal verification techniques to cryptographic protocols. For instance, Corin et al. [Corin and den Hartog 2005] leverage a variant of probabilistic Hoare logic to prove the security of ElGamal; Gagne et al. [Gagné et al. 2013] use similar methods to prove the security of the front-end of many CBC-based MACs, PMAC and HMAC. Tiwari et.al [Tiwari et al. 2015] leverages component-based program synthesis to automatically generate padding-based encryption schemes, and block cipher modes of operations. EasyCrypt [Barthe et al. 2013] is a toolset that allows user to specify and prove the correctness of cryptographic protocols.

Despite the rich literature at the intersection of cryptography and formal methods, there is little work on applying formal methods to reason about the correctness of zero-knowledge proofs. Almeida1 et al. [Almeida et al. 2010] developed a certifying compiler for Σ−protocols, a broad class of zero knowledge protocols which includes zk-SNARKs [Ben-Sasson et al. 2014]. Given a high level description of the protocol, the compiler generates an executable implementation that is provably correct using the Isabelle/HOL [Nipkow et al. 2002] theorem prover. Sidorenco et al.[Sidorenco et al. 2021] produced the first machine checked proofs of ZK protocols based on the Multi-Party-Computation-In-The-Head paradigm using EasyCrypt. More recent work has focused on building *specialized solvers* for polynomial equations over finite fields. While it is theoretically possible to encode finite field arithmetic in the theory of integers or bitvectors, the resulting constraints are very difficult to solve using off-the-shelf solvers. Hader et al. [Hader 2022] developed a decision procedure for solving polynomial equations over finite fields using a combination of a custom quantifier elimination procedure and by computing Groebner bases. Since this solver is not open source, we use a public fork of CVC5 [Ozdemir 2022] which implements a custom decision procedure for polynomial equations over a finite field.

Bug Finders for Zero-Knowledge Programs. Writing correct yet efficient zk programs requires specialized domain expertise. To the best of our knowledge, there are very few tools apart from **QED^2** that automatically find bugs [aztec 2022; electriccoin 2019; TornadoCash 2019b; trailofbits 2022] in zk programs. The most related work to **QED^2** is an open source static analyzer called Circomspect [Dahlgren 2022] designed to find bugs in Circom programs. Circomspect looks for simple syntactic patterns such as using the <-- operator when <== can be used. Such a syntactic pattern-matching approach generates many false positives and can also miss real bugs, such as the motivating example from Section 2. In particular, we emphasize that, while our motivating example does use the <− operator, it cannot be replaced with <== . Furthermore, none of the bugs detect by **QED^2** conform to this pattern, so Circomspect would not be useful for identifying any of the 8 bugs that **QED^2** detected. Finally, since Circomspect only operates on the Circom AST, it is limited to Circom programs, whereas **QED^2**, can, in principle, analyze any polynomial equations over a finite field.

10 CONCLUSION

We have presented a technique for detecting zero-knowledge proof bugs that are caused by under-constrained arithmetic circuits. Our method uses lightweight reasoning based on inference rules to propagate uniqueness constraints and switches to SMT-based reasoning when it can no longer make progress. The process terminates either when the SMT solver finds a proof or counterexample, the inference engine proves all output variables to be constrained, or no further inference is possible. Because our approach reasons directly about arithmetic circuits, it is not tied to a particular DSL and can be applied to a wide range of DSLs that support zk-snarks. We have implemented our approach in a tool called **QED^2** and evaluated it on 163 Circom circuits. Our approach was able to successfully verify or refute 70% of these benchmarks and found 8 serious vulnerabilities.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the helpful feedback. This material is based upon work partially supported by a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding source.

11 DATA AVAILABILITY STATEMENT

The data and artifacts that support the findings in this paper is openly available on Zenodo [Chen et al. 2023]

REFERENCES

2019. Tornado.cash got hacked. by Us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8

Aleo. 2022. Leo code translates to invalid Aleo instruction code. https://github.com/AleoHQ/leo/issues/2042.

José Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. 2010. A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Sigma-Protocols., Vol. 6345. 151–167. https://doi.org/10.1007/978-3-642-15497-3_10

Aztec. 2022. Disclosure of recent vulnerabilities. https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities.

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

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *FOSAD*.

Jordi Baylina. 2021. Circomlib/babyjub.circom at CFF5AB6288B55EF23602221694A6A38A0239DCC0 · Iden3/circomlib. https://github.com/iden3/circomlib/blob/cff5ab6288b55ef23602221694a6a38a0239dcc0/circuits/babyjub.circom#L45

Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. 2022. New Privacy Practices for Blockchain Software. *IEEE Software* 39, 3 (2022), 43–49. https://doi.org/10.1109/MS.2021.3086718

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 459–474. https://doi.org/10.1109/SP.2014.36

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

Daniel J. Bernstein and Tanja Lange. 2007. Faster addition and doubling on elliptic curves. Cryptology ePrint Archive, Paper 2007/286. https://eprint.iacr.org/2007/286 https://eprint.iacr.org/2007/286.

Maurizio Binello. 2019. R1CS. https://www.zeroknowledgeblog.com/index.php/the-pinocchio-protocol/r1cs

Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short Signatures from the Weil Pairing. *J. Cryptol.* 17, 4 (sep 2004), 297–319. https://doi.org/10.1007/s00145-004-0314-9

Maher Boudabra and Abderrahmane Nitaj. 2019. A New Public Key Cryptosystem Based on Edwards Curves. *Journal of Applied Mathematics and Computing* 61 (04 2019), 1–20. https://doi.org/10.1007/s12190-019-01257-y

Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Halo: Recursive Proof Composition without a Trusted Setup. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1021.

Vitalik Buterin. 2016. Quadratic arithmetic programs: From zero to hero. https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649

Yanju Chen, Shankara Pailoor, Clara Rodriguez, Franklyn Wang, Jacob Van Gaffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. 2023. Automated Detection of Underconstrained Circuits in Zero-Knowledge Proofs. Zenodo. https://doi.org/10.5281/zenodo.7776035

Collin Chin. 2021. LEO: A Programming Language for Formally Verified, Zero-Knowledge Applications. https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-leo.pdf.

Michael Connor. 2021. Disclosure of recent vulnerabilities. https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities

Ricardo Corin and Jerry den Hartog. 2005. A Probabilistic Hoare-style logic for Game-based Cryptographic Proofs (Extended Version). http://eprint.iacr.org/2005/467 To appear in ICALP 2006 Track C corin@cs.utwente.nl 13264 received 23 Dec 2005, last revised 26 Apr 2006.

Craig Costello and Benjamin Smith. 2017. Montgomery curves and their arithmetic: The case of large characteristic fields. Cryptology ePrint Archive, Paper 2017/212. https://eprint.iacr.org/2017/212 https://eprint.iacr.org/2017/212.

Fredrick Dahlgren. 2022. It pays to be Circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/.

Jacob Eberhardt and Stefan Tai. 2018. 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)*. 1084–1091. https://doi.org/10.1109/Cybermatics_2018.2018.00199

electriccoin. 2019. Zcash Counterfeiting Vulnerability Successfully Remediated. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated.

Dario fiore and Ida Tucker. 2022. Efficient Zero-Knowledge Proofs on Signed Data with Applications to Verifiable Computation on Data Streams. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1067–1080. https://doi.org/10.1145/3548606.3560630

Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Martin Gagné, Pascal Lafourcade, and Yassine Lakhnech. 2013. Automated Security Proofs for Almost-Universal Hash for MAC verification. Cryptology ePrint Archive, Paper 2013/407. https://eprint.iacr.org/2013/407 https://eprint.iacr.org/2013/407.

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

S Goldwasser, S Micali, and C Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 519–535. https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

Jens Groth. 2016. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Paper 2016/260. https://eprint.iacr.org/2016/260 https://eprint.iacr.org/2016/260.

Thomas Hader. 2022. Non-linear SMT-reasoning over finite fields.

Iden3. 2018. SnarkJS. https://github.com/iden3/snarkjs.

Wei Koh Jie. 2019. Private voting and whistleblowing on Ethereum using Semaphore. https://weijiek.medium.com/private-voting-and-whistleblowing-in-ethereum-using-semaphore-449b376808e.

Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* 1, 1 (aug 2001), 36–63. https://doi.org/10.1007/s102070100002

Matter-Labs. 2022. Zinc. https://github.com/matter-labs/zinc.

Tobias Nipkow, Markus Wenzel, and Lawrence Charles Paulson. 2002. Isabelle/HOL: A Proof Assistant for Higher-Order Logic.

Noir. 2022. Proof verification fails with a simple example. https://github.com/noir-lang/noir/issues/358.

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

Ceyhun Onur and Arda Yurdakul. 2022. ElectAnon: A Blockchain-Based, Anonymous, Robust and Scalable Ranked-Choice Voting Protocol.

Alex Ozdemir. 2022. CVC5-ff. https://github.com/alex-ozdemir/CVC4/tree/ff.

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

Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Gaffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. 2023. Automated Detection of Underconstrained Circuits for Zero-Knowledge Proofs. Cryptology ePrint Archive, Paper 2023/512. https://doi.org/10.1145/3591282 https://eprint.iacr.org/2023/512.

Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. Cryptology ePrint Archive, Paper 2013/279. https://eprint.iacr.org/2013/279 https://eprint.iacr.org/2013/279.

Torben P. Pedersen. 1991. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '91)*. Springer-Verlag, Berlin, Heidelberg, 129–140.

Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. 2021. Formal security analysis of MPC-in-the-head zero-knowledge protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–14. https://doi.org/10.1109/CSF51468.2021.00050

Ashish Tiwari, Adria Gascon, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (LNCS, Vol. 9195)*. 482–497. https://doi.org/10.1007/978-3-319-21401-6_33

TornadoCash. 2019a. Introducing Private Transactions On Ethereum NOW! https://tornado-cash.medium.com/introducing-private-transactions-on-ethereum-now-42ee915babe0.
TornadoCash. 2019b. Tornado.cash got hacked. By us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8.
Tornado.cash. 2019. Tornado.cash got hacked. by Us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8
trailofbits. 2022. The Frozen Heart vulnerability in Bulletproofs. https://blog.trailofbits.com/2022/04/15/the-frozen-heart-vulnerability-in-bulletproof.

Received 2022-11-10; accepted 2023-03-31