Scalable Verification of Zero-Knowledge Protocols

Miguel Isabel, Clara Rodríguez-Núñez, Albert Rubio
Complutense University of Madrid, Spain

***Abstract*—The application of Zero-Knowledge (ZK) proofs is rapidly growing in the industry and has become a key element to enable privacy and enhance scalability in public distributed ledgers. In most practical ZK systems, the statement to be proven is expressed by means of a set of polynomial equations in a prime field that describe an arithmetic circuit. Describing general statements using this kind of constraints is a complex and error-prone task. This can be partly mitigated by using high-level programming languages, but at the cost of losing control over the added constraints and, as a result, obtaining too large systems for complex statements. In this context, having tools to automatically verify properties of the constraint systems is of paramount importance to guarantee the security of the protocol. However, since non-linear polynomial reasoning over a finite field is needed for checking challenging properties, existing automatic tools either do not scale or cannot detect non-trivial bugs. In this paper, we present a new scalable modular technique based on the application of transformation and deduction rules that have proven to be very effective in verifying properties over the signals of a circuit given as a set of polynomial equations in a large prime field. Our technique has been implemented in a tool called CIVER and applied to verify safety properties for circuits implemented in circom, which is one of the most popular languages for defining ZK protocols. We have been able to analyze large industrial circuits and detect subtle vulnerabilities in circuits designed by expert programmers.**

1. Introduction

Zero-Knowledge (ZK) protocols allow one party to prove to another party that some statement is true without revealing any other information besides the validity of the statement [13], [23], [25]. Zero-Knowledge protocols have arisen in the context of distributed ledgers as one of the preferred solutions to provide security and privacy in a public ledger where transparency is considered a desirable feature [22], [24], [37], [38]. A ZK protocol must be sound (invalid proofs are not accepted except with negligible probability), complete (a valid proof is always accepted), and zero-knowledge (the verifier learns nothing beyond the fact that the statement is true). Relevant properties of ZK protocols in the context of the blockchain are being non-interactive (do not require the two parties to be available and interacting) and provide succinct proofs (which reduces the verification costs). This is the case of the so-called SNARKs (succinct non-interactive argument of knowledge), which do not need any interaction between the prover and the verifier and provide short proofs that are built from the secret information (witness) known by the prover [14]. In a SNARK, the statement to be proven is that, given an arithmetic circuit that operates in a large prime field, the public inputs and the outputs, we know the private inputs that together with the public inputs make the arithmetic circuit produce the outputs. The arithmetic circuit is expressed by means of polynomial equations in the finite field. Expressing a statement as polynomial equations is not an easy task. For this reason, high-level domain-specific languages like Zokrates [21] or Leo [15] have been designed. In these languages, the statement is described as a program in the given language and the compiler translates it into polynomial constraints. While this eases the work of the cryptographers, they lose control over the final generated constraint system and, in particular, it can end up being too large to be efficiently handled by the ZK protocol. For this reason, many cryptographers prefer to use low-level languages, like circom [11], to describe their statements as they provide full control over the constraints that are added. The programming language circom is currently used in a great variety of projects [18], [26], [37], mainly due to scalability reasons.

As it happens in standard programming languages, while the use of low-level languages can be very helpful to increase the efficiency of critical applications, the possibility of introducing bugs or simply not meeting the specifications grows. One of the main concerns in the context of defining arithmetic circuits for ZK-protocols is that the existence of bugs in the definition compromises the security of the protocol, which is the case when, for instance, the definition is *under-constrained*. Hence, it is important to verify the properties of the circuit that describe its intended behavior. Unfortunately, reasoning with non-linear polynomial equations over a large finite field is not an easy task, and thus existing automatic tools either do not scale or cannot detect non-trivial bugs.

In this paper, we introduce a new technique based on applying correct and complete transformation rules and correct deduction rules to the equations before applying existing SMT-solvers. Then, we take advantage of the natural hierarchical shape of the circuits to apply a refined modular reasoning where we only unfold the constraints associated with subcircuits when needed. With this approach we are able to verify or detect bugs in large circuits annotated with properties. These circuits come from industrial projects and sometimes include thousands or even millions of constraints.

A natural source of such annotated circuits is obtained from circuits written with a new feature of the circom pro-

gramming language called *tags*. With the aim of preventing security bugs, circom has recently included the use of tags for the input and output signals of the circuits. These tags are a simple form of types that have some associated semantics. As in circom, a circuit is built by connecting different subcircuits. The compiler checks that all these connections are correct with respect to the tags, that is, when an input is required to have a tag, the signal connected to this input should have the tag as well. However, while the compiler only introduces this syntactic check, the verification of the semantic information is left to the programmer.

Our first aim is to automatically verify this tag information. For this purpose, we have introduced a new construction in circom to formally specify the semantics of tags. Then, we can automatically show that, assuming that the inputs fulfill their specified tags, all other introduced tags are valid with respect to the specifications.

Our second aim is to generalize the automatic reasoning on tags to reason on a given formal specification of the circuit. This specification is given by means of pre- and post-conditions in a richer language than the language of polynomial equations. We allow the programmer to use quantifier-free general arithmetic formulas (including e.g., relational operators) over the signals. This way, she can verify that the introduced complex constraints satisfy the intended behavior expressed in a more natural language.

Finally, our third aim is to be able to show that the constraints given by the programmer define a deterministic circuit. That is, given the inputs, there is (at most) a single value for the outputs that satisfy the constraints. This property implies the so-called *weak-safety* property as defined in [11], and it is crucial to ensure the security of the ZK-protocol defined by the circom program.

Altogether, we want to provide the circom programmers with a tool that automatically verifies that the given constraints meet the intended specifications (given in an easier language), and that they define a deterministic circuit. This way, circom programmers can efficiently define circuits while notably reducing the risk of introducing bugs.

To show the applicability of our approach, we have implemented CIVER, a tool that can verify that the signals satisfy the semantics associated with the given tags, the pre-/post-specifications, and can prove weak-safety. CIVER is implemented as an extension of the circom compiler and can be used by activating some flags in the compiler call. We have extended the compiler to accept pre-/post- and tag specifications in a smooth way. We show CIVER is very effective on large circom circuits coming from real industrial projects and, as described in the experimental evaluation, has been able to automatically verify most of the properties as well as find several subtle critical bugs.

The paper provides the following results:

- A modular method for reasoning with non-linear constraints in a large finite field.
- Constructions to provide pre-/post- and tag specifications within the circom language.
- The application of the method to automatically verify weak-safety, and pre-/post- and tag specifications on large circom circuits.
- An implementation of the technique in a scalable tool called CIVER embedded in the circom compiler.
- An experimental evaluation on a wide variety of circuits, including those used in deployed industrial applications.

2. Efficiently Reasoning with Polynomial Equations over a Large Prime Field

In our context, we assume we are given a set of variables S, usually called *signals* (as they come from an arithmetic circuit), that take values in a prime finite field of cardinality p and a set of polynomial equations P in R1CS format describing the arithmetic circuit on S. In this format, every equation is of the form $A * B - C = 0$, where A, B, and C are linear expressions over the signals. This assumption is not necessary but it is a widely used format in ZK protocols and it simplifies the presentation of our transformation and deduction rules. In addition to the polynomial equations describing the circuit, we are also given a set of boolean formulas L over $\mathbb{F}_p$-arithmetic atoms (equations and inequations over arithmetic expressions) that provides some additional preconditions over the set of signals. We aim to prove that some target property T follows from P and L.

The following example illustrates the sets S, P, and L, and the property T to be proved. We represent the constraints using the notation $A * B - C = 0 \mod p$ to indicate that they hold modulo p.

Example 1. We consider the R1CS constraint system generated by a real template (called $ModSum(n)$) from a circuit written in circom, which is part of the `ECDSA` library [6], instantiated with $n = 10$. This constraint system models a 10-bit half adder: it receives two inputs a and b that can be expressed using just 10 bits (i.e., $0 \leq a, b \leq 1023$), and computes their addition returning the outputs sum and $carry$ expressing their sum and the overflow, respectively. It uses another subcomponent, called $n2b$, to add the constraints needed to transform $a + b$ to its binary representation. For example, if it receives the inputs 1000 and 50, then it returns $sum = 26$, $carry = 1$. The R1CS system P that defines this template is

$$
\begin{aligned}
& n2b\_in - (a + b) = 0 \mod p \\
& n2b\_out_0 * (n2b\_out_0 - 1) = 0 \mod p \\
& n2b\_out_1 * (n2b\_out_1 - 1) = 0 \mod p \\
& \ldots \\
& n2b\_out_{10} * (n2b\_out_{10} - 1) = 0 \mod p \\
& n2b\_in - n2b\_out_0 - 2 * n2b\_out_1 - \ldots \\
& \qquad -1024 * n2b\_out_{10} = 0 \mod p \\
& carry - n2b\_out_{10} = 0 \mod p \\
& sum - (a + b) + carry * 1024 = 0 \mod p
\end{aligned}
$$

where we have added $\mod p$ to all equations to denote that they define equalities modulo p.

Two target properties that the circuit has to verify are, for instance, that sum can be expressed using just 10 bits (that is, $0 \leq sum \leq 1023$), and that $carry$ can only take

0 and 1 as values. Hence, considering the assumptions and the properties to be checked we have:

- Preconditions $L : 0 \leq a \leq 1023 \wedge 0 \leq b \leq 1023$
- Target properties $T : 0 \leq sum \leq 1023 \wedge 0 \leq carry \leq 1$

In order to automatically prove $P \wedge L \models T$ we will show that there exists no solution for the existentially quantified problem $P \wedge L \wedge \neg T$: we add the negation of the target property and check if the formula is unsatisfiable. Thus, if it cannot be satisfied then it means that all values satisfying both P and L also satisfy T. Therefore, we have a satisfiability problem over non-linear arithmetic in a large finite field and thus, we have to reason modulo p. Unfortunately, existing tools can only solve small instances of such problems in a reasonable amount of time [12], [19], [20]. In this section, we present transformation and deduction rules that aim at 1) eliminating the need to work modulo a large prime number and 2) reducing the need to apply non-linear reasoning. The rules we provide here are the ones that have empirically shown to have a positive impact on the performance of the SMT solver.

2.1. Bound Analysis

Our method starts with a bound analysis on the signals to obtain the upper UB and lower bound LB for all signals s and all subexpressions occurring in P, L, and T. The purpose of this analysis is to avoid the reasoning modulo p.

Let us illustrate this with an example.

Example 2. Assume we have a constraint $s_1 - s_2 = 0$. Since s_1 and s_2 are signals in the finite field they can only take values in $\{0 \dots p-1\}$. Then, we have that the solutions of $s_1 - s_2 = 0 \mod p \wedge 0 \leq s_1 < p \wedge 0 \leq s_2 < p$ are the same as the solutions of $s_1 - s_2 = 0 \wedge 0 \leq s_2 < p$ since $-p < s_1 - s_2 < p$, and, in such interval, the only value that is equal to 0 modulo p is 0 itself. Thus, we can remove the modulo operator after adding the bounds for s_1 and s_2.

The process starts with $LB(s) = 0$ and $UB(s) = p - 1$ for all signals $s \in S$, and $LB(e) = -\infty$ and $UB(e) = \infty$ for any other expression e and applies the rules in Figure 1 whenever a bound can be improved until we cannot improve anymore (i.e., a fix point is found). We only apply the rules to improve some LB if we obtain a bigger number or some UB if we obtain a smaller number.

The first two rules in Figure 1 allow us to deduce any lower or upper bound (respectively) from the linear arithmetic formulas in L and the current lower or upper bounds (respectively), and it is correct by construction. The third one extracts bounds on signals from polynomial equations of the form $(x - a) * (x - b) = 0$, since the only solutions for the signal x are a and b and assuming that $0 \leq a \leq b < p$, we have that $a \leq x \leq b$. The rest of the rules are trivially correct and extend the known bounds as much as possible.

Example 3. If we apply the bound inference rules to Example 1, we can obtain the following bounds:

$$0 \leq a, b \leq 1023$$
$$0 \leq n2b\_in \leq 2026$$
$$0 \leq n2b\_out_i \leq 1 \text{ for } i = 0 \dots 10$$
$$0 \leq carry \leq 1$$

2.2. Transformation and Deduction Rules

Once we have LB and UB, we first introduce two correct and complete transformation rules, and then one correct deduction rule that will be applied to P in order to simplify it before sending the problem to an SMT solver. We assume that all constraints in P are initially written as $A * B - C = 0 \mod p$.

2.2.1. Transformation rules. Let us show the correctness and completeness of the rules given in Figure 2:

- In the first rule, if $k_m * p < LB(Expr)$ and $UB(Expr) < k_M * p$. then we have that $k_m * p < Expr < k_M * p$. Therefore, we can substitute the constraint $Expr = 0 \mod p$ by the constraint $Expr = k * p$ with $k_m \leq k \leq k_M$ as the possible solutions to $Expr = 0 \mod p$ are the possible solutions to $Expr = k * p$. Note that there always exist such k_m and k_M, but the rule only works well in practice when the distance between them is small.
- In the second rule, we apply a well-known property of finite fields, which is that $\alpha * \beta = 0$ if and only if either $\alpha = 0$ or $\beta = 0$.

Example 4. If we apply the transformation rules in Figure 2 to Example 1, we obtain the following system

$$n2b\_in - (a + b) = 0$$
$$n2b\_out_0 = 0 \vee n2b\_out_0 = 1$$
$$n2b\_out_1 = 0 \vee n2b\_out_1 = 1$$
$$\dots$$
$$n2b\_out_{10} = 0 \vee n2b\_out_{10} = 1$$
$$n2b\_in - n2b\_out_0 - 2 * n2b\_out_1 - \dots - 1024 * n2b\_out_{10} = 0$$
$$carry - n2b\_out_{10} = 0$$
$$sum - (a + b) + carry * 1024 = k * p \wedge 0 \leq k \leq 1$$

which, in this case, has a single non linear expression and needs no reasoning modulo p.

2.2.2. Deduction rules. To extract more linear information from the equations in P, we have added the deduction rule shown in Figure 3. This rule is based on the same property as the (Quadratic binomial elimination) transformation rule, but here since C is not 0, we have to add a conditional linear clause, meaning that if C is equal to 0, then either A or B must be equal to 0 (in the finite field).

Let us illustrate the use of this rule in an example.

Example 5. We consider a circuit that receives an input in that represents an array of three signals in_1, in_2, and in_3, and checks if any of them is 0 or not. In case all signals

$$\frac{L \cup LB \models E \geq \alpha}{LB(E) = \alpha} \quad (Lower\ bound\ linear\ deduction)$$

$$\frac{L \cup \mathit{UB} \models E \leq \beta}{\mathit{UB}(E) = \beta} \quad (Upper\ bound\ linear\ deduction)$$

$$\frac{(s-\alpha)*(s-\beta) = 0 \quad \in P \quad \alpha \leq \beta}{LB(s) = \alpha \qquad \mathit{UB}(s) = \beta} \quad (Polynomial\ deduction)$$

$$\frac{E - s = 0 \in P \cup L}{LB(s) = LB(E) \qquad \mathit{UB}(s) = \mathit{UB}(E)} \quad (Equality)$$

$$\frac{true}{LB(E_1 + E_2) = LB(E_1) + LB(E_2) \quad \mathit{UB}(E_1 + E_2) = \mathit{UB}(E_1) + \mathit{UB}(E_2)} \quad (Addition)$$

$$\frac{true}{LB(E_1 * E_2) = LB(E_1) * LB(E_2) \quad \mathit{UB}(E_1 * E_2) = \mathit{UB}(E_1) * \mathit{UB}(E_2)} \quad (Multiplication)$$

$$\frac{true}{LB(E_1 - E_2) = LB(E_1) - \mathit{UB}(E_2) \quad \mathit{UB}(E_1 - E_2) = \mathit{UB}(E_1) - LB(E_2)} \quad (Subtraction)$$

Figure 1: Bound inference rules.

(Modulo elimination)
$Expr = 0 \mod p \Longrightarrow Expr = k*p \wedge k_m \leq k \leq k_M$
- for some fresh variable k,
- k_m is the minimal val. s.t. $LB(Expr) \leq k_m*p$,
- k_M is the maximal val. s.t. $k_M*p \leq UB(Expr)$

(Quadratic binomial elimination)
$A*B = 0 \mod p \Longrightarrow A = 0 \mod p \vee B = 0 \mod p$

Figure 2: Transformation rules.

(Conditional binomial deduction)
$A*B = C \mod p \vdash \neg(C = 0 \mod p) \vee A = 0 \mod p \vee B = 0 \mod p$

Figure 3: Deduction rules.

do not have the value 0, then the output of the circuit is 1. If any of them is 0, the output is 0. We consider the following constraint system modeling the circuit:

$$\begin{aligned}
&acul_0 = in_0 \mod p\\
&acul_1 = acul_0 * in_1 \mod p\\
&acul_2 = acul_1 * in_2 \mod p\\
&iszero_in = acul_2 \mod p\\
&iszero_out = -iszero_in * iszero_inv + 1 \mod p\\
&iszero_in * iszero_out = 0 \mod p\\
&out = 1 - iszero_out \mod p
\end{aligned}$$

Notice, $acul_2$ will be 0 if and only if one of the inputs is zero. We use a subcomponent $iszero$ (signals $iszero\_in$, $iszero\_inv$ and $iszero\_out$) to check if $acul_2$ is 0. The following target property indicating that in case all inputs are not zero then the output is 1:

$$(in_0 \neq 0 \wedge in_1 \neq 0 \wedge in_2 \neq 0) \implies out = 1$$

Even after applying the transformation rules described in the previous section, the obtained circuit contains too many complex non-linear operations and SMT solvers cannot handle it in a reasonable time. For instance, using the (Modulo elimination) rule, the second constraint is transformed into $p * k_2 = acul_0 * in_1 - acul_1$ with $0 \leq k_2 \leq p$, which is still too complex to be handled by SMT solvers. Instead, using the (Conditional binomial elimination) rule, we can add

$$\begin{aligned}
&acul_1 = 0 \mod p \implies\\
&\qquad (acul_0 = 0 \mod p \vee in_1 = 0 \mod p)\\
&acul_2 = 0 \mod p \implies\\
&\qquad (acul_1 = 0 \mod p \vee in_2 = 0 \mod p)
\end{aligned}$$

to the equations in P, which simplifies the problem and, after removing the modulo operations with the transformation rules, enhances the performance of SMT solvers.

(Quadratic equality deduction)
$A_1*B_1 = C_1 \mod p \ \wedge \ A_2*B_2 = C_2 \mod p \vdash$
- $\neg(B_1 = B_2 \mod p) \vee \neg(C_1 = C_2 \mod p) \vee B_1 = 0 \mod p \vee A_1 = A_2 \mod p$
- $\neg(A_1 = A_2 \mod p) \vee \neg(B_1 = B_2 \mod p) \vee C_1 = C_2 \mod p$

Figure 4: Other deduction rules.

We have not found other potentially useful transformation rules since the completeness property is a strong requirement. As deduction rules, we have considered the rule given in Figure 4 as well as instantiations of it replacing both A_1 and A_2 by A and/or C_1 and C_2 by C. However, our experimental evaluation consistently revealed a negative impact on the performance of our tool when these rules were applied to the whole constraint system. Since deduction adds new formulas to the SMT problem rather than replacing existing ones, in most cases, the significant increase in the size of the final problem tends to have an overall negative impact. For instance, the rule given in Figure 4 introduces a quadratic number (on the the number of non-linear constraints) of new formulas. However, we have detected that applying the rule only in specific cases has a great impact (see Section 3.4 for more details).

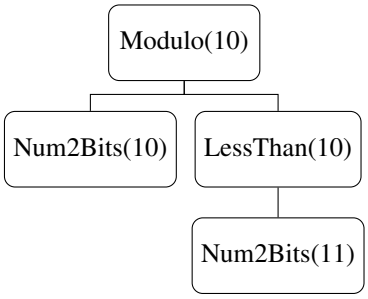
2.3. Modularity of the Technique

Arithmetic circuits are usually defined in a modular fashion, where each circuit component is built using other subcomponents. Most of these components have their own preconditions and target properties, and in many cases, the target properties (once they are proved) provide enough information to prove the properties of the parent components. Due to this, verification can take advantage of this hierarchical structure and attack the problem in a bottom-up modular way, reducing the number of constraints that need to be considered to verify the properties at each level of the circuit. For this purpose, we consider the topological order in which components are used and, following that order, we verify each component as follows:

- First, we only consider the constraints of the parent component and abstract its children using only their target properties.
- If the component is verified, we are done.
- Otherwise, we perform a second verification try adding the constraints of the children subcomponents, but abstracting, in this case, the information of the grandchildren.
- While the property is not proved we keep adding one more level of constraints but abstracting the next one.

With this approach, we are able to efficiently verify many properties considering just the constraints of the parent component, without losing the power of using constraints from the children subcomponents when needed. As shown in the experimental evaluation in Section 4, most of the properties can be verified only considering the abstractions of the children. The reason why this technique works well is that, in many cases, summarizing all the information provided by the subcomponents with their target properties is sufficient, since the properties of the subcomponents are treated as intermediate lemmas and proved sequentially. Let us present this approach with an example.

Example 6. The following R1CS system models a simplification of the template $Modulo(10)$ that is included in the `DarkForest` [18] library and contains instances of the subcomponents $Num2Bits$ and $LessThan$ from the circom library `circomlib` [27]. A full description of the circuit is given in Example 7.



The constraint system of the circuit is as follows, with the constraints grouped in their respective subcomponents:

$$
\begin{aligned}
&P_{Modulo(10)}: \\
&\quad dividend = (divisor * quotient + remainder) \mod p \\
&\quad n2b.in = remainder \mod p \\
&\quad lt.in_0 = remainder \mod p \\
&\quad lt.in_1 = divisor \mod p \\
&\quad lt.out = 1 \mod p \\
&P_{Num2Bits(10)}: \\
&\quad n2b.out_0 * (n2b.out_0 - 1) = 0 \mod p \\
&\quad \ldots \\
&\quad n2b.out_9 * (n2b.out_9 - 1) = 0 \mod p \\
&\quad n2b.in = n2b.out_0 + \cdots + 512 * n2b.out_9 \mod p \\
&P_{LessThan(10)}: \\
&\quad lt.n2b.in = lt.in_0 + 1024 - lt.in_1 \mod p \\
&\quad lt.out = lt.n2b.out_{10} \mod p \\
&P_{Num2Bits(11)}: \\
&\quad lt.n2b.out_0 * (lt.n2b.out_0 - 1) = 0 \mod p \\
&\quad \ldots \\
&\quad lt.n2b.out_{10} * (lt.n2b.out_{10} - 1) = 0 \mod p \\
&\quad lt.n2b.in = lt.n2b.out_0 + \ldots + 1024 * lt.n2b.out_{10} \mod p
\end{aligned}
$$

The first set of constraints corresponds to the constraints of the main component $Modulo(10)$, the second set to the constraints of its subcomponent $Num2Bits(10)$, and so on. In order to verify the circuit, we consider the following preconditions and target properties describing the behavior of the main component and its children:

- Main component; $Modulo(10)$:
  - Preconditions $L: 0 \leq divisor \leq 1023 \wedge 0 \leq dividend \leq 1023$
  - Target properties $T: 0 \leq remainder < divisor$
- $Num2Bits(10)$ component:
  - Preconditions L_{n2b}: $true$
  - Target properties $T_{n2b}: 0 \leq n2b.in \leq 1023$
- $LessThan(10)$ component:
  - Preconditions $L_{lt}: 0 \leq lt.in_0 \leq 1023 \wedge 0 \leq lt.in_1 \leq 1023$
  - Target properties $T_{lt}: (lt.in_0 < lt.in_1 \Rightarrow lt.out = 1) \wedge (lt.in_0 \geq lt.in_1 \Rightarrow tl.out = 0)$

Following our bottom-up approach, the problem that we consider to ensure the correctness of the main

component of the circuit (assuming that we have already verified the rest of the subcomponents) is the following:

$$L \wedge P \wedge (L_{n2b} \Rightarrow T_{n2b}) \wedge (L_{lt} \Rightarrow T_{lt}) \models T$$

In this case, to ensure that the target property (T) of the main component $Modulo(10)$ is satisfied, it is enough to consider the abstraction of its children $Num2Bits(10)$ and $LessThan(10)$, given by their pre/post specifications $L_{n2b} \Rightarrow T_{n2b}$ and $L_{lt} \Rightarrow T_{lt}$ (respectively), instead of considering all their constraints. However, when the target properties for the children are not strong enough, the abstraction loses too much information, and then, we fail. For example, in case we consider a target property for the $LessThan$ subcomponent only stating that its output is binary, i.e., $T_{lt} : 0 \leq lt.out \leq 1$, then our approach cannot verify T using the abstractions of the children and the process ends up adding the constraints of all the subcomponents to prove it.

As said, our experimental results show that, in general, the number of verification rounds that we have to perform to verify the correctness of a template is low. This way, our approach reduces the maximum number of constraints that we have to consider in a verification round: in some cases, we go from hundreds of thousands of constraints to just tens of constraints. Another advantage of the approach is that it gives extra information about the potential bugs of the circuit when the verification fails. For example, in case there is a bug in a subcomponent C but the rest of the circuit is safe, our approach is able to distinguish that the only erroneous behavior appears in C, instead of just flagging the complete circuit as buggy.

3. Verifying Circom Programs

In this section, we show how our method can be applied in the context of large industrial circuits written in circom [11], one of the most widely used programming languages in ZK projects. After a brief introduction of circom, we will show how to automatically verify properties associated with a new feature of the language, called *tags*. Then, we will generalize it to verify general pre/post specifications and conclude the section showing how our approach can also be applied to verify *weak-safety*, a key property of circom programs [11]. In Section 4, we provide a detailed evaluation of these three applications.

3.1. The Circom Language

In this section, we briefly describe the circom programming language. Circom is a DSL for designing arithmetic circuits to be used in ZK protocols. In contrast to other so-called program-based languages [11], [32], like Zokrates [21] or Leo [15], where the circuit is described by means of a program, and a compiler translates the program into a constraint system, in circom the programmer should state all constraints explicitly. For this reason, circom is called a constraint-based language [11]. Program-based languages abstract the programmer from the difficulties of expressing statements as polynomial equations, like standard programming languages do with respect to using assembly code. For standard programming languages modern compilers produce, in general, a very efficient and hard-to-improve assembly code. However, this level of optimization has not been achieved yet for the constraint systems generated by compilers for arithmetic circuit design programming languages [11]. One reason comes from the fact that optimizing and simplifying polynomial constraints is not an easy task, and there is still a lack of techniques to do it efficiently within a compiler [7]. As a result, these program-based languages can have scalability problems when considering large circuits in real industrial projects. Alternatively, since circom provides full control over the added constraints, the programmer can choose the best representation in terms of constraints in R1CS format for every particular situation, and hence achieve higher levels of scalability. Therefore, developers choose circom when they want to have such low-level control on the circuit definition, which is of paramount importance when building complex circuits, like for instance hash functions like SHA-256 or public-key cryptographic protocols like ECDSA [6]. This control requires a high level of expertise from the circom programmers and introduces potential security vulnerabilities due to the presence, for instance, of under-constrained signals. Note that, assuming correctness of the compiler, such cases cannot happen in a program-based circuit design language, since the generated constraints must be a precise definition of the deterministic behavior described by the program.

Given a circom program, the circom compiler generates 1) a set of constraints in R1CS (describing the circuit) and 2) a code in C++ or WebAssembly that allows us to efficiently compute a witness, i.e., the values of all the signals in the circuit, for given concrete inputs. Given the R1CS constraint system and a valid witness of a circuit, tools such as SNARKJS [28] can produce a valid ZK proof.

In circom, circuits are defined using *components* that are instances of generic circuits, called *templates*, that define the relation between input and output signals. The following example introduces these notions and presents the main operations used in the circom language.

Example 7.

The following circuit is a simplification of the template $Modulo(n)$ included in the `DarkForest` [18] library. The intended behavior of the template is the following: it receives two input signals, $dividend$ and $divisor$, that can be expressed using at most n bits; and performs the integer division ($dividend \backslash divisor$). It returns as outputs the $quotient$ and $remainder$ of such operation.

```
1 template Modulo(n) {
2   signal input {maxbit} dividend;
3   signal input {maxbit} divisor;
4
5   assert(dividend.maxbit <= n);
6   assert(divisor.maxbit <= n);
7
```

```
8   signal output remainder;
9   signal output quotient;
10
11  remainder <-- dividend % divisor;
12  quotient <-- dividend \ divisor;
13
14  dividend === divisor * quotient + remainder;
15
16  //signal bits_rem[n] <== Num2Bits(n)(remainder);
17
18  signal {maxbit} aux[2];
19  aux.maxbit = n;
20  aux[0] <== remainder;
21  aux[1] <== divisor;
22  signal result <== LessThan(n)(aux);
23  result === 1;
24
25 }
```

Lines 2–3, 8–9 and 18 declare the signals of the template. Signals can be inputs (lines 2–3), outputs (lines 8–9) or intermediates (18). In circom, signals, like in standard circuits, can take a single value which is immutable. Circom provides different operations to define the constraints of the R1CS system and the executable code used to efficiently compute a witness. First, the assignment instruction using the <== operation (e.g., in line 20) has a double effect: it introduces both (i) a constraint (in this case, $aux[0] - remainder = 0$) in the constraint system, and (ii) an assignment instruction ($aux[0] := remainder$) in the witness generation code produced by the compiler. The <-- and === operations used in lines 11 and 23 have a single effect as they only add an assignment instruction ($remainder := dividend \ \% \ divisor$;) and a constraint ($result - 1 = 0$), respectively. The <== operation is safe in the sense that it preserves the equivalence between the symbolic (R1CS constraint system) and the computational (witness generator) representations generated by circom. However, this operator cannot be used to assign non-quadratic expressions like $dividend \ \% \ divisor$, which must be assigned with <--. This way, it is responsability of the programmer to add the necessary R1CS constraints (via === operations) to preserve the equivalence between the computational and symbolic representations of the circuit. Note that Line 16 is not included in the original implementation of Modulo and so it is commented. We will present in the following sections why this line is needed as the circuit can present unexpected behaviors if it is not included.

As we mentioned before, this template is built using a hierarchical structure: it contains an instance of the template $LessThan(n)$ of the `circomlib` library, which checks if the value of $in[0]$ is smaller than $in[1]$.

```
27 template LessThan(n) {
28   assert(n <= 252);
29   signal input {maxbit} in[2];
30   signal output {binary} out;
31   assert(in.maxbit <= n);
32
33   component n2b = Num2Bits(n+1);
34   n2b.in <== in[0]+ (1<<n) – in[1];
35   out <== 1–n2b.out[n];
36 }
```

Finally, the template $LessThan(n)$ uses an instance of the template $Num2Bits(n)$, which returns the binary representation of the given input signal in using n bits.

```
37 template Num2Bits(n) {
38   signal input in;
39   signal output {binary} out[n];
40
41   var lc1=0;
42   var e2=1;
43   for (var i = 0; i<n; i++) {
44     out[i] <-- (in >> i) & 1;
45     out[i] * (out[i] –1 ) === 0;
46     lc1 += out[i] * e2;
47     e2 = e2+e2;
48   }
49   lc1 === in;
50 }
```

The $Num2Bits$ template utilizes *variables* (defined in lines 41, 42, 43) to perform repetitive constructions (loops) and build complex expressions. Variables in circom have two different semantics, (i) as symbolic expressions when generating constraints and (ii) as standard mutable variables that can change their value when generating the witness computation code.

Let us present in detail the first case showing how the constraints of the template $Num2Bits$ are generated when instantiated with $n = 3$. The symbolic execution of the template reaches the for loop with the state $\{lc1 = 0, e2 = 1, i = 0\}$. Then, in the first iteration, the constraint $out[0] * (out[0] - 1) = 0$, requiring $out[0]$ to be 0 or 1, is generated (line 45) and the state is changed to $\{lc1 = out[0], e2 = 2, i = 1\}$. In the second iteration, $out[1] * (out[1] - 1) = 0$ is generated and the state is changed to $\{lc1 = out[0] + out[1] * 2, e2 = 4, i = 2\}$. Finally, the third iteration adds the constraint $out[2] * (out[2] - 1) = 0$ and updates the variable states to $\{lc1 = out[0] + out[1] * 2 + out[2] * 4, e2 = 8, i = 3\}$. Then, the execution exits the for loop and the constraint $out[0] + out[1] * 2 + out[2] * 4 - in = 0$ is added in line 49.

The previous circuits use one of the new features included in the circom compiler v2.1 named *tags*. Tags are a simple form of types that are mainly added to input and output signals to introduce additional commitments when composing different components. This way, tags add an extra syntactic check as they force all these connections between components to be correct with respect to them. That is, when an input is required to have a tag, the signal connected to this input should have the tag as well. Thus, the tag $maxbit$ included in the input signal of the template $LessThan$ indicates that only signals with this tag can be connected to it. However, the compiler only introduces this syntactic check and does not check if the signals satisfy the intended semantic information associated with their tags. For instance, the circom compiler does not check that signal out of the $LessThan$ template is actually $binary$ ($out = 0 \vee out = 1$),

it only adds this information to the signal so it can be connected to inputs requiring this tag.

Circom considers two types of tags: the ones that have a value associated, like $maxbit$, and the ones that do not, like $binary$. When a tag of a signal has a value, it must be known at compilation time (that is, it can only depend on the parameters of the circuit, but not on the values of the signals). In circom, when a tag with value is added to an array of signals, all signals of the array are required to have the same tag value. For example, the instruction $aux.maxbit = n$ of Line 19 of the $Modulo(n)$ template associates the value n (that is known at compilation time) to the tag $maxbit$ of all signals of the array aux.

3.2. Tag specification and verification

Tags provide a cheap compilation mechanism to add some security to the designed circuits, since when a signal without a tag is passed to a component input that expects such a tag, the compiler raises an error. Then the programmer has to decide if the tag can be explicitly added because she knows that the property really holds or if extra checks are needed to ensure it. In any case, the compiler warns about a potential problem and the programmer has the responsibility to check what is needed. However, the fact that the programmer is responsible for checking the correctness of the added tags weakens the impact of this feature on the security of circuits.

In this section we will show that we can liberate the programmer from the obligation of checking that the domain of values of tagged signals can take corresponds to the semantics associated with their tags. To this end, before using the framework described in Section 2 to verify tags, we extend circom with a construction that allows the programmers to formally specify the semantics of the tags:

```
spec_tag tag_name: QF_Expr(x.tag_name,x)
```

where `QF_Expr(x.tag_name,x)` is a quantifier-free boolean formula over arithmetic atoms built over the signal `x` and its tag value `x.tag_name`.

For instance, we can specify the tags $binary$ and $maxbit$ used in previous examples as follows:

```
spec_tag binary: 0 <= x && x<= 1;
spec_tag maxbit: 0 <= x && x <= 2**x.maxbit – 1;
```

Note that $binary$ could also be specified as $x = 0 \vee x = 1$.

Once a circuit is compiled, all tags of the signals have known (constant) values. This way, we can always evaluate the expression describing each tag of each signal and obtain a formula that only depends on this signal. For instance, if the template $Modulo(n)$ of Example 7 is instantiated with $n = 10$, then the signal aux receives the tag *maxbit* with value 10. Hence, after the compilation of the circuit, its specification states that $0 \leq out \wedge out \leq 1023$.

We introduce the notion of *tag-specification* to describe functions that go from signals to the sets of values that they can take. Any specification given by the language presented above generates a tag-specification once the circuit is compiled. Given a circuit $C = (S, P)$, for a set of signals S and polynomial equations P, a tag-specification T_C is a function from the signals of the circuit S to $\mathcal{P}(\mathbb{F}_p)$ that associates each signal to the set of values that it can take.

Example 8. We consider the circuit $LessThan(n)$, described in Example 7, instantiated with $n = 10$, then the tag-specification of the circuit is given by:

$$T_C(in[i]) = \{0, \ldots, 1023\} \text{ for } i = 0, 1$$
$$T_C(n2b.out[i]) = \{0, 1\} \text{ for } i = 0, \ldots, 10$$
$$T_C(out) = \{0, 1\}$$

Given a circuit $C = (S, P)$, we assume that the tag specification T_C is given by means of quantifier-free arithmetic formulas over the tagged signals in S, and we will call L the set of all those formulas for input signals and T the set of formulas for the rest of signals. We consider that the circuit is correct with respect to the tags if $P \wedge L \models T$, that is, from the constraints of the circuit and the tag-specifications of the inputs we can ensure that the tag-specifications of the rest of the signals hold.

Example 8 (continued). We consider again the circuit $LessThan(10)$ and the tag-specification of the circuit that we introduced previously. In this case, the signals $in[0]$ and $in[1]$ are inputs, while the rest of the tagged signals are intermediates or outputs. Hence, we have to ensure that $P \wedge L \models T$ with P being the constraint system modeling the circuit, and L and T being $0 \leq in[0], in[1] \leq 1023$ and $0 \leq n2b.out[0] \ldots n2b.out[10], out \leq 1$, respectively.

As we discussed in Section 2, after applying the transformation and deduction rules, we can prove that $P \wedge L \models T$ by checking the unsatisfiability of the existentially quantified formula $P \wedge L \wedge \neg T$ using an SMT solver. If it is unsatisfiable we have proved that the circuit is tag-correct. Otherwise, if it is satisfiable, the model returned by the SMT solver is a counterexample showing that the circuit does not satisfy its tag-specification. Our approach combines the application of the transformation and deduction rules described in Section 2.2 and the modular reasoning described in Section 2.3 to efficiently check the tag-correctness of real-world circuits. Thus, in order to verify that a circuit satisfies its tag-specification, we consider each one of its components individually and try to verify its correctness considering the abstractions of its children subcomponents given by their tags specifications: if the input's tags of a subcomponent are specified by L_{sub} and the output's tags with T_{sub}, we abstract all constraints of the subcomponent by using $L_{sub} \Rightarrow T_{sub}$. As described, if the check fails for some component we add the constraints of its children and keep trying.

Example 8 (continued). Let us consider again the $LessThan(10)$ circuit, our modular reasoning allows us to consider the verification in two levels: first, we check if the component $Num2Bits(11)$ satisfies its tag-specification, and then we check if $LessThan(10)$ satisfies it, abstracting its child subcomponent $n2b$ using only its tag-specification. That is, instead of including the

constraints of the $Num2Bits(11)$ component, we include the formula $true \Rightarrow 0 \leq n2b.out[0] \ldots n2b.out[10] \leq 1$.

3.3. Verification of Postconditions in Circom

The kind of properties that we can express using tags is limited as they can only define properties in which a single signal is involved. For instance, we cannot express that the expected behavior of the signal out of the template $LessThan$ illustrated in Example 7 is $out = in[0] < in[1]$ using tags. In order to express more complex properties, we extend the circom language allowing users to define pre/post-conditions using quantifier-free formulas over the signals of the circuit. This way, programmers can specify the behaviors of their circuits in a more natural language than the language of polynomial equations. We introduce the following instructions to the circom language to allow programmers to define the pre- and post-conditions that a template has to satisfy, respectively.

```
spec_precondition QF_Expr;
spec_postcondition QF_Expr;
```

For instance, the specification of the $Modulo(10)$ template described in Examples 6 and 7 can be stated as follows:

```
spec_precondition 0<= divisor && divisor < 1024;
spec_precondition 0<= dividend && dividend < 1024;
spec_postcondition 0<= remainder && remainder <divisor;
```

Given the specification of a circuit, our goal is to check if the constraint system modeling the circuit ensures that its specification holds. That is, given a circom circuit defined by the constraint system P and its specification stated by preconditions and postconditions defined by the user (L and T respectively), we check if $P \wedge L \models T$, and we proceed like when verifying tag-specificatons.

3.4. Verification of Weak Safety on Circom Circuits

One of the main properties that ZK circuits have to satisfy is *weak-safety* [11]. Intuitively, we say that a circuit is weakly-safe when for any input of the circuit, the constraint system describing the circuit ensures that there is a unique possible assignment for the output signals for the given inputs. This determinism is essential for maintaining the security of the circuit, as a malicious prover could exploit vulnerabilities if alternative values for the outputs could be used to generate valid ZK proofs.

For example, the circuit $Module(10)$ (Example 7) does not ensure weak-safety if we do not uncomment Line 16. As, for instance, when the inputs are $dividend = 3$ and $divisor = 1$ besides the expected outputs $quotient = 3$ and $remainder = 0$, the outputs $quotient = 4$ and $remainder = -1$ also satisfy the constraints.

Intuitively, to check if a circuit ensures weak-safety we check that for any two solutions, w and w', of the constraint system P modeling the circuit, if the inputs are the same the outputs should be the same too. We model this using two copies of the constraint system P and P', one defined over the signals $s_1, \ldots, s_n$, and the other over $s'_1, \ldots, s'_n$. Then, for all the input signals in_i we add as the precondition L that $in_i = in'_i$ for all i and as target property the postcondition T stating that all outputs should be the same $out_1 = out'_1 \wedge \cdots \wedge out_n = out'_n$. As in the previous cases, after applying the transformation and deduction rules, we can prove that $P \wedge P' \wedge L \models T$ by checking the unsatisfiability of the existentially quantified formula $P \wedge P' \wedge L \wedge \neg T$ using an SMT solver.

Example 9. We encode the verification of the $Num2Bits(11)$ template as follows:

- We denote the constraints satisfied by the first solution as P:

$$\begin{aligned} &out_0 * (out_0 - 1) = 0 \mod p \\ &\ldots \\ &out_{10} * (out_{10} - 1) = 0 \mod p \\ &in = out_0 + \ldots + 1024 * out_{10} \mod p \end{aligned}$$

- We denote the constraints satisfied by the second solution as P' :

$$\begin{aligned} &out'_0 * (out'_0 - 1) = 0 \mod p \\ &\ldots \\ &out'_{10} * (out'_{10} - 1) = 0 \mod p \\ &in' = out'_0 + \ldots + 1024 * out'_{10} \mod p \end{aligned}$$

- We include the precondition L requiring the input signals to be equal:

$$in = in'$$

- Finally, we define the target property T stating that all outputs are also equal.

$$\begin{aligned} &out_0 = out'_0 \\ &\ldots \\ &out_{10} = out'_{10} \end{aligned}$$

This way, the circuit satisfies weak-safety in case

$$P \wedge P' \wedge L \models T$$

As mentioned at the end of Section 2.2.2, the *(Quadratic equality deduction)* rule in Figure 4 is only useful when applied selectively. In particular, this deduction rule has a positive impact when checking weak-safety if it is applied to every non-linear constraint $e \in P$ and its corresponding version $e' \in P'$. The reason is that, in most cases, it introduces linear formulas relating the signals in P with their primed versions in P', which notably helps the SMT solver to check weak-safety.

3.5. Extension to Other ZK DSLs and Non-Modular Circuits

Our transformation techniques readily extend to other languages like ZoKrates [21] or Noir [9] that produce constraints in R1CS format, and they can also be generalized and applied to scale tools like the one presented in [39]

that verifies programs written in Halo2 [42], that consider polynomial equations in a different format. However, the modular structure that we exploit in the circom programs cannot, in general, be directly detected from a plain R1CS. In order to do it, we may need to add a phase of constraint clustering to identify related constraints and create a hierarchical structure of clusters like the one we have for circom programs. Then, we can apply the modular approach described above to each one of these clusters. In addition, the user can specify lemmas or properties (like pre/post conditions described in Section 3.3 or facts in Section 4.2.3) that will be associated to clusters based on the signals that are involved in the specification, and use and prove them modularly as for circom programs.

4. Experimental evaluation

In this section, we present how the proposed technique has been implemented. After that, we provide and discuss the results obtained when applied to a variety of circuits. Finally, we analyze the most important bugs reported by our tool.

4.1. Implementation of CIVER

We have implemented CIVER [17] as an extension of the circom compiler [11]. Both the circom compiler and CIVER are written in Rust. The SMT solver chosen to perform the experimental evaluation is Z3 [19] due to its good performance and its easy integration with Rust through the Z3 library.

CIVER is activated when compiling a circuit using the flag `--CIVER`. We have implemented three use modes: `--check_tags`, `--check_postconditions`, and `--check_safety` to verify tag-specification, pre-/post-specifications, and weak safety, respectively. Their uses can be combined, although it could affect CIVER's performance. After choosing one or more of the modes, CIVER starts the verification process template by template. It gathers all the specifications related to the chosen mode and uses the modular approach presented in Section 2.3 to prove the correctness of the circuit. For each component, it applies the transformation rules presented in Section 2.2 and then, relies on Z3 to check the satisfiability of the problem. As explained in Section 2.3, the SMT solver can be called multiples times during template verification, while the template is not yet fully verified and there are pending constraints from its subcomponents to be included. We can use the option `--verification_timeout` to define the timeout considered by the Z3 solver. Its default value is 15 seconds for each call. It is important to highlight that if the verification of a circuit timeouts, CIVER points out the subcomponents that could not be verified. Hence, programmers can focus on these parts (applying manual inspection or other verification techniques) instead of considering the whole circuit.

Finally, CIVER incorporates the option `--print_smt` that produces the SMT problem in smtlib2 format to facilitate its integration with other SMT solvers like yices [20], barcelogic [12], mathsat [16], or cvc5 [10].

4.2. Results in Real-World Benchmarks

We have conducted our experimental evaluation using, as benchmarks, real-world circuits that are currently on deployment. Our first source of experiments is the circom library, called `circomlib` [27]. This library is widely used for cryptographic purposes and contains tens of templates such as comparators, hash functions, digital signatures, binary and decimal converters, and many more. The `circomlib` has been previously audited by cryptographers and security engineers. We have also used two projects as benchmarks: `DarkForest` [18], a decentralized space-conquest game based on Ethereum, whose cryptographic logic is built with circom and is currently on deployment since 2020; and the `ECDSA` library [6], which provides proof-of-concept implementations of the ECDSA protocol in circom. We have applied the three use modes previously described to every circuit test defined in these three sets of benchmarks. Some results are not included in this section due to space limitations, but the complete results can be found in the Appendix. Experimental results have been obtained using an AMD Ryzen Threadripper PRO 3995WX 64-Cores Processor with 512GB of RAM (Linux Kernel Debian 5.10.70-1) and a timeout of 15 seconds per call to Z3 solver. The main results are reported in the next sections.

4.2.1. Experimental evaluation of tag verification. The `circomlib` has been fully tagged with mainly two kinds of tags: $binary$, and $maxbit$. Similarly, we have tagged all the circuits from the `ECDSA` library implementing big-int operations using $binary$ and $maxbit$ tags. Regarding the `DarkForest` project, we have tagged all the circuits using $binary$, $maxbit$, max and max_abs tags. Tags max and max_abs mean, respectively, that the value of the tagged signal x cannot be greater than $x.max$ and greater than $x.max\_abs$ in absolute value.

Results are presented in Table 1. Columns show 1) the circuit name, 2) the number of template instances contained in each circuit, 3) the number of tagged templates to be verified (V?), and the main results: 4) the number of verified templates (V), 5) the number of buggy templates (F), and 6) the number of timeouts (T). To show the size of the analyzed circuits, we also provide: 7) the number of non-linear, and 8) linear constraints used during the tag verification process, 9) the total number of signals contained in the circuits, 10) the number of intermediate and output signals whose tags must be verified by CIVER, 11) the maximum number of child levels needed to consider verifying a template, and 12) the time needed to verify them. For instance, the $SHA256$ circuit is built with 105 different components, and all of them have been verified. This circuit is quite large, since it is defined by 62416 non-linear constraints and 439664 linear constraints. It has 501457 signals, of which 197840 have been tagged as binary. CIVER is able to prove that the circuit satisfies its tags-specification in 11.62s. Notice that the

TABLE 1: Verification of tag specifications

Circuit	Inst.	V?	V	F	T	NL-C.	L-C	Sigs	Tags	Level	Time
eddsamimicsponge	29	10	10	0	0	8889	12606	21493	787	0	0.65s
sha256	105	105	105	0	0	62416	439664	501457	197840	0	11.62s
escalarmulfix	12	2	2	0	0	3948	6167	10116	254	0	0.06s
pedersen	133	3	3	0	0	5540	7570	13111	502	0	0.11s
mux4_1	5	1	1	0	0	23	62	86	4	0	0.01s
pointbits_loopback	10	6	6	0	0	2337	3350	5673	1835	0	0.42s
babypbk	12	2	2	0	0	3948	6167	10116	254	0	0.05s
pedersen2	12	2	2	0	0	3381	4747	8129	254	0	0.05s
escalarmul_min	70	4	4	0	0	2816	4871	7688	769	0	0.14s
check_gates	13	13	13	0	0	46	194	241	77	0	0.11s
eddsa	36	13	13	0	0	17825	33371	51167	5181	0	1.01s
check_bitify	13	12	12	0	0	832	2190	3019	1278	0	0.39s
escalarmulany	13	2	2	0	0	2554	5577	8134	254	0	0.04s
binsub	4	4	4	0	0	49	55	104	50	0	0.04s
smtverifier10	159	11	11	0	0	4110	7990	12106	813	0	0.32s
eddsaposeidon	101	10	10	0	0	7395	13609	21002	787	0	0.60s
sum	4	4	4	0	0	97	103	200	99	0	0.06s
aliascheck	6	4	4	0	0	516	1529	2044	644	0	0.22s
check_comparators	14	13	13	0	0	132	67	197	151	1	0.13s
decoder	1	1	1	0	0	7	1	9	7	0	0.01s
bigmod_32	22	22	21	0	1	124	209	313	195	1	15.74s
bigsubmodp_32	17	17	17	0	0	40	104	141	83	2	0.24s
bigmult_21	8	8	8	0	0	12	31	42	25	1	0.10s
bigmult_23	8	8	7	0	1	54	99	148	95	1	15.36s
bigadd_23	10	10	10	0	0	23	76	97	56	1	0.13s
bigadd_2030	10	10	10	0	0	1859	751	2581	2162	1	0.17s
bigsub_23	13	13	13	0	0	23	95	116	68	2	0.17s
biglessthan_23	12	12	12	0	0	33	101	132	62	1	0.12s
bigmodsubthree	7	6	5	1	0	14	25	39	23	2	0.09s
move	79	61	57	0	4	12860	8507	21225	8900	6	378.64s
reveal	77	59	55	0	4	11270	7989	19123	8612	6	317.38s

maximum level for the SMT solver is 0, which means that all tags have been proven using just the abstraction of their subcomponents given by the tags of their inputs and outputs. However, other circuits, such as $check\_comparators$, cannot be verified considering just this abstraction and require multiple verification levels.

CIVER can verify all the circuits from the `circomlib` contained in the test set. Most of them, in less than a second. Regarding the `ECDSA` library, CIVER is able to verify six out of nine test cases in half a second. The template $BigModSubThree()$ contains a bug which is explained in detail in Section 4.3.1. The verification of the remaining test cases timeouts, the reason is that the verification of the template $BigMultNoCarry()$ is too complex. This template receives two $maxbit$-tagged arrays as inputs, which are the coefficients of two polynomials, and returns a $maxbit$-tagged array as output, which is the coefficients of the product of these two polynomials. Although the behavior of the template is simple, the constraint system used to describe it is not, since it codifies the behavior of the circuit by applying the Lagrange Theorem to guarantee polynomial uniqueness. CIVER timeouts when trying to verify this template when the degree of the polynomials (the size of the arrays) is greater than 3. Finally, in the `DarkForest` project, 112 out of 120 template instances are verified correctly. The remaining correspond to different instances of the $Modulo$ template. As explained in Section 4.3.2, the template $Modulo$ contains a bug that compromises its security. A counterexample showing this buggy behavior can be found using CIVER when the template is instantiated using smaller parameters.

4.2.2. Experimental evaluation of postconditions verification. We have specified using pre- and post-condition many of the circuits in the three sets of benchmarks under consideration. We have selected circuits that either have a clear and easy-to-follow implementation or are well documented. The results provided by the CIVER verification are presented in Table 2. Columns show 1) the circuit name, 2) the number of templates containing postconditions to be verified (V?), 3) the number of verified templates (V), 4) the number of buggy templates (F), and 5) the number of timeouts (T). To show the size of the analyzed circuits, we also provide: 6) the total number of postconditions to be verified, 7) the maximum number of child levels, and 8) the time needed to verify them.

The majority of the circuits in the `circomlib` are verified. However, CIVER finds a counterexample for two circuits containing the template instance $Bits2Num(254)$. Our tool discovers that this template instance admits two possible solutions and does not satisfy its postcondition. It is important to highlight that these circuits are not buggy since they also contain another template, called $AliasCheck()$, to

prohibit the second solution, avoiding the possible bug. However, template *Random*(), which appears in multiple circuits of the DarkForest project, also contains an instance of *Bits2Num*(*254*), but in this case, it is not protected by *AliasCheck*(), then the bug remains unsolved. The fix in [4] adds the missing *AliasCheck*() to prevent the second solution. Apart from this bug, this project contains another critical issue: the *Module*() template is under-constrained and thus it does not satisfy its specification. This issue is explained in detail in Section 4.3.2. Finally, the ECDSA library satisfies all the specifications of their templates but four, which cannot be proven before CIVER timeouts. The templates causing the timeouts are *BigMultNoCarry*() and *BigMult*(), which uses the previous template to perform a big-int multiplication.

TABLE 2: Verification of pre-/post-condition specifications

Circuit	V?	V	F	T	Conds	L	Time
eddsamimicsponge	10	7	0	3	1374	1	75.41s
sha256	95	95	0	0	151408	0	1.48s
escalarmulfix	2	1	0	1	1361	0	15.04s
pedersen	2	1	0	1	8066	0	15.07s
mux4_1	3	2	0	1	33	0	15.04s
pointbits_loopback	6	3	1	2	20	1	60.59s
babypbk	2	1	0	1	1361	0	15.05s
pedersen2	3	1	0	2	1153	1	30.13s
escalarmul_min	3	2	0	1	4608	0	15.09s
check_gates	12	12	0	0	48	1	0.18s
eddsa	13	7	1	5	5770	1	121.19s
eddsamimc	10	7	0	3	1374	1	75.41s
check_bitify	12	10	0	2	558	1	45.32s
escalarmulany	2	2	0	0	2	0	0.03s
binsub	3	3	0	0	4	0	0.03s
smtverifier10	14	13	0	1	146	1	33.04s
eddsaposeidon	10	7	0	3	1374	1	75.41s
sum	3	3	0	0	4	0	0.04s
aliascheck	5	4	0	1	511	1	30.19s
check_comparators	13	13	0	0	26	2	0.15s
decoder	1	0	1	0	13	0	0.01s
bigmod_32	20	18	0	2	60	2	46.00s
bigsubmodp_32	15	15	0	0	33	2	0.21s
bigmult_21	7	7	0	0	12	1	0.08s
bigmult_23	7	5	0	2	32	2	45.71s
bigadd_23	9	9	0	0	25	0	0.07s
bigadd_2030	9	9	0	0	241	0	0.12s
bigsub_23	12	12	0	0	32	2	0.19s
biglessthan_23	11	11	0	0	43	1	0.28s
bigmodsubthree	6	5	1	0	10	2	0.09s
move	44	38	1	5	700	6	362.71s
reveal	41	35	1	5	682	6	326.58s

4.2.3. Experimental evaluation of weak-safety verification. We can check weak safety for any of the circuits in the benchmarks, even for the circuits that are not well documented. For this reason, there are new circuits in Table 3. This table provides the experimental evaluation of the weak-safety verification, and its columns are similar to those in Table 2. Most of the circuits in the circomlib are weakly-safe. CIVER is able to find a counterexample for 14 template instances, most of them related to cryptographic functions like *Eddsaposeidon*(), *Babypbk*(), or *Pedersen*(). After studying these circuits, we believe that these templates admit several solutions for values of inputs that do not make sense in the algebra of the cryptographic curve. Consequently, they cannot be considered as bugs. In fact, some of these counterexamples could be avoided by adding known facts encoding the information about the mathematical structures behind the functions. This is done in CIVER using the construction

```
spec_fact QF_Expr;
```

which is handled as a fact to be included in the problem in order to solve it. For instance, the *BabyAdd*() template can be verified by adding a fact stating that for any two points of the curve (x_1, y_1) and (x_2, y_2), the value of

$$\lambda = 168700 * x_1 * x_2 * y_1 * y_2$$

satisfies $\lambda \neq \pm 1 \mod p$. This is a known property of this curve that is necessary to ensure that the addition of the points is safe, as its value is given by

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + x_2 y_1}{1 + \lambda}, \frac{y_1 y_2 - x_1 x_2}{1 - \lambda} \right)$$

We also highlight that the template *Decoder*() contains a bug which is explained in Section 4.3.3. Regarding the DarkForest project, CIVER proves 71 out of 79 template instances to be weakly-safe and fails to prove it for 8. Again, the template causing most of the timeouts is the template *Modulo*(). CIVER verified all the circuits from the ECDSA library but three: two instances of template *BigMultNoCarry*(); and *BigMod*(), which performs the big-int modulo operation.

We have finally compared the performance of CIVER with QEP^2 [35], using the available instances this tool has in [5], which are part of the circomlib benchmarks and the benchmarks in ECDSA and DarkForest. CIVER performed better in all benchmarks but a subset of the circomlib that uses Baby-JubJub circuits for which QEP^2 includes a built-in pattern that allows solving them without any SMT reasoning. We have not included in CIVER this kind of ad-hoc solutions for particular circuits. Apart from these cases, in all the rest of benchmarks when QEP^2 does not time out, CIVER is faster (more than 10 times, in general), and in the four circuits of DarkForest (only two of them are shown in the table) QEP^2 times out after three hours, while CIVER finds the bugs (and verify the rest of the subcomponents) in around 10 minutes.

4.3. Real-World Bugs Detected by CIVER

In this section, we present the bugs that we have found using CIVER in the real-world projects circomlib, ECDSA and DarkForest. All these projects are widely used and have been previously audited. Consequently, it is even more relevant that some of these bugs have not been previously reported by related tools (see Section 5). For these bugs, CIVER has been able to find a counterexample proving that the circuit is not correct. These counterexamples can help programmers to understand the bug and fix it. It is important

TABLE 3: Verification of weak-safety

Circuit	V?	V	F	T	L	Time
eddsamimicsponge	29	21	3	5	3	137.96s
sha256	105	104	0	1	4	203.47s
escalarmulfix	12	6	3	3	0	47.72s
pedersen	133	131	0	2	0	32.36s
mux4_1	5	5	0	0	0	0.07s
pointbits_loopback	10	7	0	3	3	74.74s
babypbk	12	6	3	3	0	47.71s
pedersen2	12	6	3	3	0	47.57s
escalarmul_min	70	69	0	1	0	16.38s
check_gates	13	13	0	0	0	0.13s
eddsa	36	28	3	5	3	109.93s
eddsamimc	29	21	3	5	3	132.68s
check_bitify	13	11	0	2	3	85.85s
escalarmulany	13	7	3	3	0	47.86s
binsub	4	4	0	0	0	1.56s
smtverifier10	159	157	0	2	3	86.04s
eddsaposeidon	101	93	3	5	3	131.91s
sum	4	4	0	0	0	5.17s
aliascheck	6	5	0	1	0	15.40s
check_comparators	14	14	0	0	0	0.22s
decoder	1	0	1	0	0	0.01s
babyadd	1	0	0	1	0	15.02s
mimc_sponge_hash	2	2	0	0	0	1.83s
montgomerydouble	1	0	0	1	0	15.02s
edwards2montgomery	1	0	1	0	0	0.02s
montgomery2edwards	1	0	1	0	0	0.02s
poseidonex	97	97	0	0	0	2.78s
escalarmul	69	67	0	2	0	31.19s
bigmod_32	22	21	0	1	3	55.91s
bigsubmodp_32	17	17	0	0	0	0.19s
bigmult_21	8	8	0	0	0	0.10s
bigmult_23	8	7	0	1	0	15.55s
bigadd_23	10	10	0	0	0	0.10s
bigadd_2030	10	10	0	0	0	0.27s
bigsub_23	13	13	0	0	0	0.13s
biglessthan_23	12	12	0	0	0	0.13s
bigmodsubthree	7	7	0	0	0	0.07s
move	79	71	0	8	6	393.76s
reveal	77	69	0	8	6	401.37s

to highlight that these bugs have been reported through PRs [1]–[3] in their corresponding GitHub repositories. A fourth bug, reported in [4] (already commented in section 4.2.2), was detected because CIVER could not verify the circuit.

4.3.1. Bugs detected when checking tags. We have discovered a bug [2] in the template $ModSubThree(n)$ of the `ECDSA` library when checking if the tagged signals satisfy their specification. This template receives three inputs a, b and c that represent numbers that can be expressed using n bits (tagged as $maxbit$ with value n) and performs the operation $a - b - c$. It returns the result of the operation using n bits (signal out, tagged as $maxbit$ with value n), along with a signal indicating if there has been an underflow ($borrow$, tagged as $binary$). The specification of the tag $maxbit$ introduced in Section 3.2 states that the output signal out satisfies $0 \leq out \leq 2^n - 1$. However, CIVER has been able to find counterexamples of witnesses accepted by the template that do not satisfy this property. For example, if we consider the instantiation of the template with $n = 3$ and the inputs $a = 0$, $b = 7$ and $c = 7$, the generated R1CS constraint system is satisfied by the assignment $out = -6$, $borrow = 1$, which should not be valid according to the tag-specification of the circuit. We have implemented a fix for this issue [2] which adds the check between lines 62 and 67 and used CIVER to verify the fix.

```
56  template ModSubThree(n) {
57    signal input {maxbit} a;
58    signal input {maxbit} b;
59    signal input {maxbit} c;
60    signal output {maxbit} out;
61    signal output {binary} borrow;
62    /* Needed check, not included
63       in the current version
64    component lt_aux = LessThan(n + 1);
65    lt_aux.in[0] <== b + c;
66    lt_aux.in[1] <== a + 2 ** n;
67    lt_aux.out === 1; */
68    component lt = LessThan(n + 1);
69    lt.in[0] <== a;
70    lt.in[1] <== b + c;
71    borrow <== lt.out;
72    out.maxbit = n;
73    out <== borrow * (1 << n) + a - b - c;
74  }
```

We have also used CIVER to study the tags verification of the ongoing project PIL-STARK [36]. This project is part of the ZK-EVM developed by Polygon-Hermez, and defines a circuit that contains 11 million of constraints, (5 million of them are non-linear). We have detected a bug similar to the one presented in this section, where a signal was supposed to be less than a fixed number, but it was not guaranteed by the constraint system.

4.3.2. Bugs detected when checking postconditions. We have discovered a critical bug [3] in one of the most widely-used templates of the `DarkForest` project: the $Module(n)$ template. A simplification of this template has been presented in Example 7. We specify the circuit using the following postcondition: $0 \leq remainder \leq divisor - 1$. However, CIVER fails when trying to prove this property and returns counterexamples showing unexpected behaviors. For instance, it returns a witness with input signals $dividend = 0$ and $divisor = 1$, and outputs $quotient = 11$ and $remainder = -11$, which does not satisfy the postcondition described above. We can fix the bug [3] by adding the line 16 (commented in the example). This way, we add a $Num2Bits(n)$ component to the circuit that ensures that the signal $remainder$ can be expressed using just n bits, which is crucial to guarantee that $remainder$ is positive. Let the reader notice that if $remainder$ was negative, it would require more than n bits to be expressed.

4.3.3. Bugs detected when checking weak-safety. Finally, the weak-safety analysis performed by CIVER has also helped us to find bugs in real-world projects.

We have found that the template $Decoder$, included in the `circomlib`, has a weak-safety bug [1]. The intended behavior of the circuit is the following: it receives an input inp and returns an array out of w signals such that $out[i] = 1$ if $i == inp$, otherwise $out[i] = 0$. However, the template is buggy, because it admits multiple outputs beyond

the expected one, leading to a weak-safety bug. CIVER reports a counterexample breaking the weak-safety property. For instance, in case $w = 5$ and $inp = 3$, the template accepts both the expected output $out = [0, 0, 0, 1, 0]$ and the buggy one $out = [0, 0, 0, 0, 0]$. We can fix the bug [1] by substituting lines 81-82 by line 83 (commented in the example) which adds the missing constraints.

```
75 template Decoder(w) {
76   signal input inp;
77   signal output {binary} out[w];
78   signal output {binary} success;
79   var lc=0;
80   for (var i=0; i<w; i++) {
81     out[i] <-- (inp == i) ? 1 : 0;
82     out[i] * (inp-i) === 0;
83     // out[i] <== IsZero()(inp - i);
84     lc = lc + out[i];
85   }
86   lc ==> success;
87   success * (success -1) === 0;
88 }
```

4.4. Main Conclusions of the Experimental Evaluation

Our experimental results show that our approach is able to verify real circuits from some of the most relevant Zero-Knowledge projects on deployment. Moreover, we have been able to detect critical bugs in these projects that could not been found by existing tools. We consider that the modular reasoning is a key element to increase the scalability of the approach. As shown in the previous sections, for most of the templates, CIVER only needs the consider the local information of the templates and the abstractions of their subcomponents to verify them. Moreover, the experimental results show that the transformation and deduction rules presented in Section 2 enhance the behavior of SMT solvers when considering polynomial equations in R1CS. The elimination of any of the rules described in such a section causes CIVER to fail in several circuits of the projects presented above.

Finally, let us remark that the good results we have obtained when verifying the circuits in the ECDSA benchmarks and the cicuit in the PIL-STARK project, show that CIVER can work well on vector representations of numbers, which opens the door to incorporate in CIVER reasoning mechanisms for *extensions fields*, whose standard representation is by means of similar vectors.

5. Related Work

There has been recently a great effort to develop tools and techniques that allow programmers to detect bugs and verify ZK circuits. As we discussed in the introduction, the full control of the added R1CS constraints, which languages like circom promote, increases the risk of introducing bugs when describing a circuit.

Let us discuss the main differences between CIVER and other tools for ZK circuit verification. First, we consider tools such as `circomspect` [31], `ZKAP` [41], and the –inspect option included in the circom compiler. These tools mostly work at a syntactic level, looking for patterns that indicate the presence of potential vulnerabilities. For example, these tools report a warning when they find a signal that does not appear in any constraint of the system (which usually is a sign of a bug as the signal can take any value). While these tools are able to detect the most common bugs and help developers to perform a first sanity check of their circuits, they cannot detect more complex and subtle vulnerabilities as the ones studied in this work. For instance, these tools cannot detect the bugs detected by CIVER described in the previous section.

Secondly, we consider the tools `ecne` [40] and QEP^2 [35] whose goal is the verification of ZK circuits against under-constrained bugs. Intuitively, `ecne` relies on a set of propagation rules that are applied at the level of the constraint system to propagate the uniqueness of the signals of the circuit. Initially, `ecne` considers that only the inputs of the circuit have unique values and then applies propagation rules iteratively until reaching a fixed point. For instance, if the R1CS constraint system modeling the circuit includes a constraint of the form $s = Expr$ with all signals that occur in $Expr$ being unique, `ecne` deduces that the value of s must be unique too. `ecne` is able to detect under-constrained bugs in real-world circuits (for instance, it detects the bug in the circuit $Decoder()$ presented in the previous section) but flags as dangerous, many valid solutions when they do not match the propagation rules.

QEP^2 is a tool that tries to overcome this limitation by combining the use of propagation rules and the use of SMT solvers to verify circuits against under-constrained bugs. Intuitively, QEP^2 applies propagation rules to extend as much as possible the information about signal uniqueness of the circuit, and then relies on SMT solvers to detect if the uniqueness of a new signal can be deduced from the constraint system. This way, QEP^2 is able to reduce the number of false positives raised by `ecne`. However, QEP^2 performance decreases when it needs to consider real-world circuits with thousands of constraints as the efficiency of SMT solvers when considering non-linear constraint systems is limited as we discussed in this paper. For instance, if we apply QEP^2 to the `Darkforest`'s circuits with a timeout of 3 hours, we do not obtain any results, whereas CIVER detected the described bug after 6 minutes.

We consider that our work and these tools are complementary, as some of the transformation and deduction rules described in this paper could help these tools to reduce the number of non-linear constraints considered (via modular reasoning or module elimination), and hence reduce the false positives while making them applicable in real large code. Similarly, advances in the development of SMT solvers that can check satisfiability of polynomial equations over finite fields [30], [33] can also help improve the performance of these tools. Another difference between CIVER and these tools is that CIVER is not only focused on finding under-constrained bugs, but can consider different target properties describing the behavior of the circuit. For

example, the circuit $ModSubThree()$ previously described is not under-constrained but does not satisfy its specification.

Recently, a preliminary work on verifying Halo2 circuits [42] has been published [39]. Similar to ours, this work also uses an SMT solver to verify weak-safety and more general properties are satisfied by the PLONKish constraints systems. However, authors claim in their discussion that their approach do not scale and it cannot be applied to real-world benchmarks. Their work could benefit from our transformation rules and be extended to take advantage of the particularities of the PLONKish format.

In this work, we have assumed the correctness of the ZKP compiler. This assumption is sometimes too strong since compilers sometimes contain small bugs as it has been previously studied in other works like [34], where a bug was found in the CirC compiler. This work presents a verified finite-field blaster for CirC. A field blaster is a set of encoding rules that transform a statement containing bit-vectors, finite-field elements, and booleans into a set of R1CS equations over a finite field. They also give verification conditions for these rules and use SMT-based finite-field reasoning to prove that the transformations are correct. Notice our work is different from [34] since they assume that the original statement is correct and then the SMT solver is used to check that the statement and the final R1CS equations produced by the field blaster are equivalent, but if the original statement contains a bug, then the R1CS equation will also have it.

Similar to CirC, ZKCrypt [8] is a verifying compiler which takes a high-level specification of a cryptographic protocol via pre/post-conditions and automatically synthesizes an efficient implementation of a prover and a verifier of the protocol from the specification. This approach is a synthesis problem which differs from ours since it is a verification problem, where CIVER verifies if a given implementation of the protocol via constraints satisfies the specification.

Finally, a functional language called CODA [29] has emerged. CODA allows building and specifying circuits at the same time and, thus, certifies the correctness of their circuits semiautomatically. Our approach enables the specification of complex correctness properties directly using the circom language, making it more accessible to cryptographers (usually, non-experts in verification techniques).

Acknowledgments

This work was funded partially by the Spanish MCI, AEI and FEDER (EU) project PID2021-122830OB-C41 and by the CM project S2018/TCS-4314 co-funded by the EU.

References

[1] Pull-request to fix the bug in the circomlib detected by civer when checking weak-safety. https://github.com/iden3/circomlib/pull/81.

[2] Pull-request to fix the bug in the ecdsa library detected by civer when checking tags. https://github.com/0xPARC/circom-ecdsa/pull/34.

[3] Pull-request to fix the bug in the template modulo() of the darkforest project detected by civer when checking postconditions. https://github.com/darkforest-eth/circuits/pull/5.

[4] Pull-request to fix the bug in the template random() of the darkforest project detected by civer when checking postconditions. https://github.com/darkforest-eth/circuits/pull/4.

[5] QEP^2. Available at https://github.com/Veridise/Picus (accessed on 11 december 2023).

[6] 0xPARC. zk–ECDSA. https://0xparc.org/blog/zk-ecdsa-1.

[7] Elvira Albert, Marta Bellés-Muñoz, Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Distilling constraints in zero-knowledge protocols. In *Computer Aided Verification*, pages 430–443, 2022.

[8] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. Cryptology ePrint Archive, Paper 2012/258, 2012.

[9] Aztec. Language noir. https://github.com/noir-lang/noir.

[10] 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. cvc5: A versatile and industrial-strength smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2022.

[11] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2022.

[12] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic smt solver. In *Computer Aided Verification*, pages 294–298, 2008.

[13] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, 2017/1066, 2017.

[14] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks. 2022.

[15] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Paper 2021/651, 2021.

[16] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2013.

[17] CIVER. https://github.com/costa-group/circom_civer.

[18] darkforest eth. Darkforest: decentralized and persistent game built on ethereum. Github repository: https://github.com/darkforest-eth (accessed on 7 June 2023).

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software*, TACAS, 2008.

[20] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer International Publishing, 2014.

[21] Jacob Eberhardt and Stefan Tai. ZoKrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things*, pages 1084–1091, 2018.

[22] EthHub. ZK-Rollups. EthHub. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/.

[23] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, 1985.

[24] Vincent Gramoli, Len Bass, Alan D. Fekete, and Daniel W. Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2711–2724, 2016.

[25] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, 2016.

[26] Hermez Network. Hermez docs. https://hermez.io.

[27] Iden3. Circomlib: Library of circom templates. https://github.com/iden3/circomlib.

[28] Iden3. SNARKJS: JavaScript implementation of zk-SNARKs. https://github.com/iden3/snarkjs.

[29] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. Cryptology ePrint Archive, Paper 2023/547, 2023.

[30] Leonardo Alt" Lucas Clemente Vella. On satisfiability of polynomial equations over large prime fields. In *SMT 2022*.

[31] Trail of Bits. circomspect: Static analyzer of zk circuits. Github: https://github.com/trailofbits/circomspect (accessed on 13 June 2023).

[32] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *43rd IEEE Symposium on Security and Privacy*. IEEE, 2022.

[33] Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark Barrett. Satisfiability modulo finite fields. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 163–186, Cham, 2023.

[34] Alex Ozdemir, Riad Wahby, Fraser Brown Brown, and Clark Barrett. Bounded verification for finite-field-blasting (in a compiler for zero knowledge proofs). In *Computer Aided Verification*, 2023.

[35] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proc. ACM Program. Lang.*, 7(PLDI), 2023.

[36] Polygon. PIL STARK. https://github.com/0xPolygonHermez/pil-stark.

[37] Polygon. Polygon zk-evm. Available documentation online: https://zkevm.polygon.technology/ (accessed on 7 June 2023).

[38] Scroll. Scroll zk-evm. Available documentation online: https://scroll.io/blog/architecture (accessed on 7 June 2023).

[39] Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. Automated analysis of halo2 circuits. Cryptology ePrint Archive, Paper 2023/1051, 2023. https://eprint.iacr.org/2023/1051.

[40] Franklyn Wang. Ecne: Automated verification of zk circuits. Blog entry: https://0xparc.org/blog/ecne (accessed on 13 June 2023).

[41] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. Cryptology ePrint Archive, Paper 2023/190, 2023. https://eprint.iacr.org/2023/190.

[42] zcash. The Halo2 book. https://zcash.github.io/halo2/index.html.

Appendix A.

In this appendix, we present the results of the complete experimental evaluation on the three sets of benchmarks.

TABLE 4: Verification of weak safety (continued)

Circuit	V?	V	F	T	Level	Time
move	79	71	0	8	6	393.76s
init	79	71	0	8	6	353.08s
whitelist	3	3	0	0	0	1.86s
reveal	77	69	0	8	6	401.37s
biomebase	77	69	0	8	6	360.80s

TABLE 5: Verification of weak safety

Circuit	V?	V	F	T	Level	Time
poseidon3	71	71	0	0	0	0.83s
iszero	1	1	0	0	0	0.01s
escalarmulw4table	1	1	0	0	0	0.02s
eddsamimicsponge	29	21	3	5	3	137.96s
sha256	105	104	0	1	4	203.47s
escalarmulfix	12	6	3	3	0	47.72s
poseidonex	97	97	0	0	0	2.78s
greatereqthan	8	8	0	0	0	0.19s
pedersen	133	131	0	2	0	32.36s
poseidon6	74	74	0	0	0	1.09s
mux4_1	5	5	0	0	0	0.07s
pointbits_loopback	10	7	0	3	3	74.74s
sha256_2	105	103	0	2	4	209.09s
mimc_sponge	1	1	0	0	0	2.06s
montgomerydouble	1	0	0	1	0	15.02s
constants	2	2	0	0	0	0.02s
check_switcher	3	3	0	0	0	0.03s
mux3_1	5	5	0	0	0	0.06s
mux2_1	5	5	0	0	0	0.06s
babypbk	12	6	3	3	0	47.71s
pedersen2	12	6	3	3	0	47.57s
multiplexer	4	4	0	0	0	0.05s
escalarmul_min	70	69	0	1	0	16.38s
edwards2montgomery	1	0	1	0	0	0.02s
sha256	105	104	0	1	4	187.13s
check_gates	13	13	0	0	0	0.13s
eddsa	36	28	3	5	3	109.93s
greaterthan	7	7	0	0	0	0.18s
babyadd	1	0	0	1	0	15.02s
escalarmulw4table	1	1	0	0	0	0.03s
montgomery2edwards	1	0	1	0	0	0.02s
escalarmulw4table	1	1	0	0	0	0.01s
eddsamimc	29	21	3	5	3	132.68s
mimc	1	1	0	0	0	0.34s
check_bitify	13	11	0	2	3	85.85s
escalarmulany	13	7	3	3	0	47.86s
escalarmul	69	67	0	2	0	31.19s
binsub	4	4	0	0	0	1.56s
lessthan	6	6	0	0	0	0.18s
isequal	2	2	0	0	0	0.02s
smtverifier10	159	157	0	2	3	86.04s
montgomeryadd	1	0	1	0	0	2.02s
lesseqthan	9	9	0	0	0	0.21s
babycheck	1	1	0	0	0	0.01s
eddsaposeidon	101	93	3	5	3	131.91s
sum	4	4	0	0	0	5.17s
mimc_sponge_hash	2	2	0	0	0	1.83s
mux1_1	5	5	0	0	0	0.04s
aliascheck	6	5	0	1	0	15.40s
sign	6	5	0	1	0	15.45s
check_comparators	14	14	0	0	0	0.22s
decoder	1	0	1	0	0	0.01s
test_bigmod_22	22	20	0	2	3	47.47s
test_bigmod_32	22	21	0	1	3	55.91s
test_bigsubmodp_32	17	17	0	0	0	0.19s
test_bigmult_21	8	8	0	0	0	0.10s
test_bigmult_22	8	7	0	1	0	15.17s
test_bigmult_23	8	7	0	1	0	15.55s
test_bigadd_15	8	8	0	0	0	0.08s
test_bigadd_23	10	10	0	0	0	0.10s
test_bigadd_2030	10	10	0	0	0	0.27s
test_bigsub_15	13	13	0	0	0	0.13s
test_bigsub_23	13	13	0	0	0	0.13s
test_biglessthan_23	12	12	0	0	0	0.13s
test_bigsubthree_33	7	7	0	0	0	0.07s

TABLE 6: Verification of tag specifications

Circuit	Inst.	V?	V	F	T	NL-C.	L-C	Sigs	Tags	Level	Time
iszero	1	1	1	0	0	2	0	4	1	0	0.03s
eddsamimicsponge	29	10	10	0	0	8889	12606	21493	787	0	0.65s
sha256	105	105	105	0	0	62416	439664	501457	197840	0	11.62s
escalarmulfix	12	2	2	0	0	3948	6167	10116	254	0	0.06s
greatereqthan	8	8	8	0	0	91	23	114	99	1	0.08s
pedersen	133	3	3	0	0	5540	7570	13111	502	0	0.11s
mux4_1	5	1	1	0	0	23	62	86	4	0	0.01s
pointbits_loopback	10	6	6	0	0	2337	3350	5673	1835	0	0.42s
sha256_2	105	105	105	0	0	31384	218139	249212	98205	0	10.93s
constants	2	1	1	0	0	0	33	34	32	0	0.01s
check_switcher	3	1	1	0	0	2	9	14	1	0	0.01s
mux3_1	5	1	1	0	0	11	36	48	3	0	0.01s
mux2_1	5	1	1	0	0	6	22	29	2	0	0.01s
babypbk	12	2	2	0	0	3948	6167	10116	254	0	0.05s
pedersen2	12	2	2	0	0	3381	4747	8129	254	0	0.05s
multiplexer	4	2	2	0	0	24	43	80	13	1	0.03s
escalarmul_min	70	4	4	0	0	2816	4871	7688	769	0	0.14s
sha256	105	105	105	0	0	62352	439344	501073	197712	0	11.31s
check_gates	13	13	13	0	0	46	194	241	77	0	0.11s
eddsa	36	13	13	0	0	17825	33371	51167	5181	0	1.01s
greaterthan	7	7	7	0	0	91	21	112	98	1	0.07s
eddsamimc	29	10	10	0	0	8889	12606	21493	787	0	0.60s
check_bitify	13	12	12	0	0	832	2190	3019	1278	0	0.39s
escalarmulany	13	2	2	0	0	2554	5577	8134	254	0	0.04s
escalarmul	69	2	2	0	0	2813	3590	6404	254	0	0.06s
binsub	4	4	4	0	0	49	55	104	50	0	0.04s
lessthan	6	6	6	0	0	91	18	109	97	1	0.07s
isequal	2	2	2	0	0	2	2	7	2	0	0.02s
smtverifier10	159	11	11	0	0	4110	7990	12106	813	0	0.32s
lesseqthan	9	9	9	0	0	91	26	117	100	1	0.10s
eddsaposeidon	101	10	10	0	0	7395	13609	21002	787	0	0.60s
sum	4	4	4	0	0	97	103	200	99	0	0.06s
mux1_1	5	1	1	0	0	2	12	15	1	0	0.01s
aliascheck	6	4	4	0	0	516	1529	2044	644	0	0.22s
sign	6	6	6	0	0	516	1530	2046	646	0	0.28s
check_comparators	14	13	13	0	0	132	67	197	151	1	0.13s
decoder	1	1	1	0	0	7	1	9	7	0	0.01s
test_bigmod_22	22	22	21	0	1	96	209	285	167	1	15.56s
test_bigmod_32	22	22	21	0	1	124	209	313	195	1	15.74s
test_bigmult_21	8	8	8	0	0	12	31	42	25	1	0.10s
test_biglessthan_23	12	12	12	0	0	33	101	132	62	1	0.12s
test_bigadd_15	8	8	8	0	0	20	126	142	81	1	0.09s
test_bigadd_23	10	10	10	0	0	23	76	97	56	1	0.13s
test_bigmult_23	8	8	7	0	1	54	99	148	95	1	15.36s
test_bigsub_23	13	13	13	0	0	23	95	116	68	2	0.17s
test_bigsub_15	13	13	13	0	0	24	157	177	97	2	0.16s
test_bigadd_2030	10	10	10	0	0	1859	751	2581	2162	1	0.17s
test_bigsubmodp_32	17	17	17	0	0	40	104	141	83	2	0.24s
test_bigmult_22	8	8	8	0	0	31	65	93	58	1	0.18s
test_bigmodsubthree_3	7	6	5	1	0	14	25	39	23	2	0.09s
move	79	61	57	0	4	12860	8507	21225	8900	6	378.64s
init	79	61	57	0	4	11403	8005	19269	8748	6	385.80s
reveal	77	59	55	0	4	11270	7989	19123	8612	6	317.38s
biomebase	77	59	55	0	4	11270	7989	19123	8612	6	343.85s

TABLE 7: Verification of pre-/post-condition specifications

Circuit	V?	V	F	T	Conds	L	Time
iszero	1	1	0	0	1	0	0.01s
eddsamimicsponge	10	7	0	3	1374	1	75.41s
sha256	95	95	0	0	151408	0	1.48s
escalarmulfix	2	1	0	1	1361	0	15.04s
greatereqthan	7	7	0	0	10	1	0.09s
pedersen	2	1	0	1	8066	0	15.07s
mux4_1	3	2	0	1	33	0	15.04s
pointbits_loopback	6	3	1	2	20	1	60.59s
sha256_2	95	95	0	0	75323	0	1.49s
constants	1	1	0	0	32	0	0.01s
check_switcher	2	2	0	0	3	0	0.02s
mux3_1	5	4	0	1	34	1	15.07s
mux2_1	5	5	0	0	18	1	0.54s
babypbk	2	1	0	1	1361	0	15.05s
pedersen2	3	1	0	2	1153	1	30.13s
multiplexer	4	4	0	0	34	1	0.18s
escalarmul_min	3	2	0	1	4608	0	15.09s
sha256	95	95	0	0	151280	0	1.56s
check_gates	12	12	0	0	48	1	0.18s
eddsa	13	7	1	5	5770	1	121.19s
greaterthan	6	6	0	0	9	1	0.07s
eddsamimc	10	7	0	3	1374	1	75.41s
check_bitify	12	10	0	2	558	1	45.32s
escalarmulany	2	2	0	0	2	0	0.03s
escalarmul	2	1	0	1	4097	0	15.05s
binsub	3	3	0	0	4	0	0.03s
lessthan	5	5	0	0	8	1	0.07s
isequal	2	2	0	0	2	0	0.03s
smtverifier10	14	13	0	1	146	1	33.04s
lesseqthan	8	8	0	0	11	2	0.12s
eddsaposeidon	10	7	0	3	1374	1	75.41s
sum	3	3	0	0	4	0	0.04s
mux1_1	5	5	0	0	10	1	0.06s
aliascheck	5	4	0	1	511	1	30.19s
sign	5	4	0	1	511	1	30.16s
check_comparators	13	13	0	0	26	2	0.15s
test_bigmod_22	20	18	0	2	60	2	45.75s
test_bigmod_32	20	18	0	2	60	2	46.00s
test_bigmult_21	7	7	0	0	12	1	0.08s
test_biglessthan_23	11	11	0	0	43	1	0.28s
test_bigadd_15	7	7	0	0	41	0	0.06s
test_bigadd_23	9	9	0	0	25	0	0.07s
test_bigmult_23	7	5	0	2	32	2	45.71s
test_bigsub_23	12	12	0	0	32	2	0.19s
test_bigsub_15	12	12	0	0	52	2	0.30s
test_bigadd_2030	9	9	0	0	241	0	0.12s
test_bigsubmodp_32	15	15	0	0	33	2	0.21s
test_bigmult_22	7	7	0	0	22	1	5.12s
test_bigmodsubthree_3	6	5	1	0	10	2	0.09s
move	44	38	1	5	700	6	362.71s
init	44	38	1	5	688	6	345.22s
reveal	41	35	1	5	682	6	326.58s
biomebase	41	35	1	5	682	6	328.45s

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents CIVER, an SMT-based tool for verifying zkSNARK circuits written the circom language. CIVER uses a simple set of transformations to eliminate non-linear constraints and takes a modular verification approach, resulting in a much more scalable tool than prior work. The authors show how to verify the semantic consistency of circom tags as well as other important properties of zkSNARK circuits, including weak-safety (that circuits are not underconstrained).

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) Clever use of constraint transformations and bounds checks to eliminate nonlinear constraints, greatly helping scalability.
2) Demonstrates efficiency of CIVER through benchmarks.
3) Demonstrates effectiveness of CIVER by finding bugs that other tools missed.

B.4. Noteworthy Concerns

It is unclear how difficult it would be to generalize the results beyond the circom language.

Appendix C.
Response to the Meta-Review

Regarding the concern raised by the reviewers on the applicability of the results to other ZK DSLs beyond circom, Section 3.5 provides insights in this direction and, in particular, it addresses the challenge of recovering the modular structure of the circuit in those languages where we can only have access to the full generated constraint system. Its implementation and experimental evaluation for languages like Halo2 or ZoKrates remains as future work.