CIRCOM: A Circuit Description Language for Building Zero-knowledge Applications

Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina

Abstract—A zero-knowledge (ZK) proof guarantees that the result of a computation is correct while keeping part of the computation details private. Some ZK proofs are tiny and can be verified in short time, which makes them one of the most promising technologies for solving two key aspects: the challenge of enabling privacy to public and transparent distributed ledgers and enhancing their scalability limitations. Most practical ZK systems require the computation to be expressed as an arithmetic circuit that is encoded as a set of equations called rank-1 constraint system (R1CS). In this paper, we present CIRCOM, a programming language and a compiler for designing arithmetic circuits that are compiled to R1CS. More precisely, with CIRCOM, programmers can design arithmetic circuits at a constraint level, and the compiler outputs a file with the R1CS description, and WebAssembly and C++ programs to efficiently compute all values of the circuit. We also provide an open-source library called CIRCOMLIB with multiple circuit templates. CIRCOM can be complemented with SNARKJS, a library for generating and validating ZK proofs from R1CS. Altogether, our software tools abstract the complexity of ZK proving mechanisms and provide a unique and friendly interface to model low-level descriptions of arithmetic circuits.

Index Terms—zero-knowledge proof, circuit, domain-specific language, compiler, blockchain.

1 INTRODUCTION

A *zero-knowledge* (ZK) protocol allows a *prover* to prove a statement to a *verifier* without revealing any knowledge beyond the fact that the statement is true [1]–[3]. ZK proofs were introduced 30 years ago as theoretical objects, but since then, there has been a lot of research into developing secure and efficient protocols that could be used in practice. In general, efficiency is measured considering three parameters: the computational cost of generating a proof, the size of the proof, and the time required to verify it. In the context of distributed ledgers, it is specially important to have small proof sizes and short verification times.

The most popular, efficient, and general-purpose ZK protocols are *ZK succinct arguments of knowledge* (ZK-SNARKs) [4]–[6]. A prominent application of ZK-SNARKs is Zcash [7], a public blockchain based on bitcoin that uses these proofs in its core protocol for verifying that private transactions (named *shielded* transactions) have been computed correctly while providing complete anonymity to the participants of the network. Efficient ZK protocols are also very convenient for distributed ledgers with the capability of executing *smart contracts*. A smart contract is a program that allows developers to define a logic for processing transactions. Adding the capability of processing transactions that include the verification of a ZK proof opens up a wide range of possibilities for smart contracts. For example, with ZK proofs, smart contracts can check logic statements while keeping certain details private, which is crucial for the adoption of public distributed ledger technology by enterprises and businesses. Examples of such privacy-preserving schemes can be found in [8], [9]. Recently, ZK protocols are also used in conjunction with smart contracts for enhancing the scalability of distributed ledgers, giving rise to what is known as *layer-2 solutions*. An example of a layer-2 solution is a *ZK-rollup*, a mechanism in which thousands of transactions are bundled into a single batch transaction [10]. This way, a smart contract can check the processing of large amounts of transactions by simply verifying a ZK proof of few bytes. The key of ZK-rollups is that ZK proofs are smaller and faster to verify than doing so with each individual transaction [11].

Generally, ZK-SNARK protocols are used to prove the correctness of a computation. In this context, the way of expressing a computation is by defining it as an *arithmetic circuit* [4], [12], [13]. An arithmetic circuit is a circuit built with addition and multiplication gates, and wires that carry values from a prime finite field $\mathbb{F}_p$, where p is typically a very large prime number. A prover uses a circuit to prove that he knows a valid assignment to all wires of the circuit, and if the proof is correct, the verifier is convinced that the computation expressed as a circuit is valid, but learns nothing about the wires' specific assignment. The common encoding of this type of circuits consists of a set of equations called *rank-1 constraint system* (R1CS), which is later used by the ZK-SNARK protocol to generate a proof. A valid proof shows, without revealing secret values, that the prover knows an assignment to all wires of the circuit that fulfill all constraints of the R1CS. An issue that appears when applying ZK protocols to complex computations, like a circuit describing the logic of a ZK-rollup, is that the amount of constraints to be verified is extremely large (up to hundreds of millions of constraints). In these cases, it is impractical to define circuits manually and we need tools for that. We can classify tools in the ZK ecosystem in two large categories: *frontends* (languages) and *backends* (libraries).

While frontends provide a way of specifying to-be-proved statements, backends are involved in the generation

- *M. Bellés-Muñoz is with the Information and Communication Technologies Engineering Department at Pompeu Fabra University, 08018 Barcelona, Spain. E-mail: marta.belles@upf.edu.*
- *M. Isabel and A. Rubio are with the Department of Computer Systems and Computing at Complutense University of Madrid, 28040 Madrid, Spain.*
- *J.L. Muñoz-Tapia is with the Department of Network Engineering at Polytechnic University of Catalonia, 08034 Barcelona, Spain.*
- *J. Baylina is with 0KIMS Association, 6300 Zug, Switzerland.*

and verification of the corresponding ZK proof. In this paper we present CIRCOM, a low-level language, also known as constraint-based language or hardware language [14], for specifying to-be-proved statements as circuits but aided by a *domain-specific language* (DSL) and its corresponding compiler [15], [16]. To be best of our knowledge and according to the available literature [14], CIRCOM is the only implemented constraint-based DSL.

Programmers can use the CIRCOM language to define arithmetic circuits and the compiler generates a file with the set of associated R1CS constraints together with a program (written either in C++ or WebAssembly) that can be run to efficiently compute a valid assignment to all wires of the circuit. One of the main particularities of CIRCOM is that it is designed as a modular language that allows the definition of parameterizable small circuits called *templates*, which can be instantiated to form larger circuits. CIRCOM users can create their own custom templates, but they can also use templates from CIRCOMLIB [17], a publicly available library with hundreds of circuits such as comparators, hash functions, digital signatures, binary and decimal converters, and many more. The architecture behind CIRCOM not only provides a simple interface to model arithmetic circuits and generate their corresponding constraints, but it also abstracts the complexity of the underlying ZK proving mechanism. In particular, the output files of CIRCOM can be used directly by SNARKJS [18], which is a library we developed to automatize the generation and verification of ZK-SNARK proofs.

The article is organized as follows. In Section 2, we provide background on ZK-SNARKs and arithmetic circuits. In Section 3, the present the main existing tools and compare them to CIRCOM. In Section 4, we introduce the characteristics of CIRCOM and give several examples of a correct use of the language. In the following Section 5, we present some practical applications that illustrate the power of the CIRCOM language. In Section 6, we evaluate the performance of CIRCOM in large circuits described by millions of constraints. In Section 7, we define the concepts of *correct* and *safe* CIRCOM programs, which can help programmers understand the philosophy of the language and help them with the writing of circuits. Conclusions are in Section 8.

2 Background

A ZK proof enables a *prover* to convince a *verifier* that a statement is true without revealing any information beyond the veracity of the statement [1], [3]. In this context, a statement is usually associated to an *instance*, a public input known to both prover and verifier, and a *witness*, a private input known only by the prover. In this section we provide some background on a particular type of ZK proof systems called ZK-SNARKs, and on arithmetic circuits.

2.1 ZK-SNARKs

ZK-SNARKs [4]–[6] belong to a group of ZK proofs known as *arguments of knowledge*. An argument of knowledge proves that, with very high probability, the prover does *know* a concrete valid witness. An argument of knowledge is considered a SNARK if it is non-interactive and, regardless of the size of the statement being proved, has succinct proof size (e.g. [6]-proofs are ≈ 200 bytes). Most ZK-SNARKs also guarantee short verification time [6], [19].

The main downside of these protocols is that they need an initial phase called *trusted setup*. This step requires the generation of some random values that need to be immediately destroyed. In fact, if the random values are ever exposed, the security of the whole scheme is compromised. To enhance the security of this setup phase, most implementations make use of a *multi-party computation* (MPC), which allows multiple independent parties to collaboratively construct the trusted setup parameters. In this process, it is enough that one single participant deletes its secret counterpart of the contribution to keep the whole scheme secure [20]. Our software SNARKJS has a framework for generating and verifying MPCs, providing a holistic architecture for building privacy-enabled applications.

Like most ZK systems, ZK-SNARKs operate in the model of arithmetic circuits, meaning that the language $\mathcal{L}$ is that of satisfiable arithmetic circuits. An assignment to the wires is *valid* if and only if for every gate, the value on the output wires matches that gate's operation and the values on its input wires [21].

2.2 Arithmetic Circuits

The most widely studied language in the context of ZK-SNARK proofs is the NP-complete language of *circuit satisfiability* [4], [12], [13]. Essentially, a *circuit* consists of a set of wires connected to gates that perform some operation. Circuit satisfiability is a classical problem of computability theory that consists of determining whether a given circuit has an assignment of its inputs that makes the output true. If that is the case, the circuit is called *satisfiable*. Otherwise, the circuit is called *unsatisfiable*.

In cryptographic implementations of this problem, we use a particular type of circuits called *arithmetic circuits* (often simply called "circuits"). The gates of an arithmetic circuit consist on additions and multiplications modulo p, where p is typically a large prime number of approximately 254 bits [22]. The wires of an arithmetic circuit are usually called *signals*, and they can carry any value from the prime finite field $\mathbb{F}_p$. As with electronic circuits, we can distinguish between *input*, *intermediate*, and *output signals*.

Usually, there is a set of public signals known both to prover and verifier, and the prover proves that, with that public information, he knows a valid assignment to the rest of signals that makes the circuit satisfiable. From now on, we extend the meaning of the word *witness* to an assignment to all signals a the circuit, both public and private.

Example 1. Circuit C from Fig. 1 is an arithmetic circuit defined over the prime finite field $\mathbb{F}_{11}$ that, given four private inputs s_1, s_2, s_3, s_4, it outputs the result of the operation

$$s_1 \times s_2 \times s_3 + s_4.$$

To perform the calculation, the circuit uses two multiplication gates and one addition gate, which requires two intermediate signals s_5, s_6, and an output signal s_7. Hence, C is a circuit defined by the set of signals

$$S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7\}.$$

An example of a witness for C is $w = \{2,\ 3,\ 3,\ 9,\ 6,\ 7,\ 5\}$.

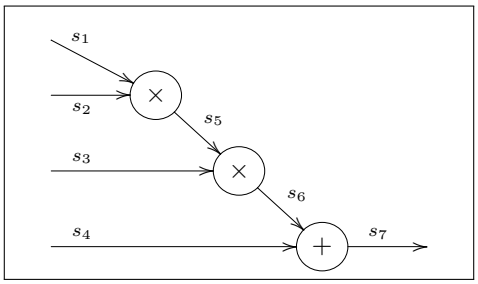

Figure 1: Representation of an arithmetic circuit C defined over the finite field $\mathbb{F}_{11}$ that outputs the result of the operation $s_1 \times s_2 \times s_3 + s_4 \mod 11$.

Recent years have seen a concentration of efforts towards different encodings of arithmetic circuits [23]. In the following, we define a classical form for encoding circuits in an algebraically useful way called *rank-1 constraint system* (R1CS). An R1CS encodes a program as a set of conditions over its variables, so that a correct execution of a circuit is equivalent to finding a satisfiable variable assignment. Due to the transformability of arithmetic circuits into R1CS, programs specified in R1CS are often referred to as *circuits*, and their variables as *signals*.

Formally, a *quadratic constraint* over a set of signals $S = \{s_1, \ldots, s_n\}$ is an equation of the form

$$(a_1 s_1 + \cdots + a_n s_n) \times (b_1 s_1 + \cdots + b_n s_n) - (c_1 s_1 + \cdots + c_n s_n) = 0,$$

where $a_i, b_i, c_i \in \mathbb{F}_p$ for all $i \in \{1, \ldots, n\}$. In short, we write a constraint as $\mathbf{a} \times \mathbf{b} - \mathbf{c} = 0$, where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are linear combinations of $s_1, ..., s_n$. A *rank-1 constraint system* (R1CS) over a set of signals $S = \{s_1, \ldots, s_n\}$ is defined as a finite collection of quadratic constraints over S.

Example 2. We can represent the circuit C from Fig. 1 as the following R1CS over S:

$$\begin{cases} s_1 \times s_2 - s_5 = 0 & \mod 11 \\ s_5 \times s_3 - s_6 = 0 & \mod 11 \\ s_6 + s_4 - s_7 = 0 & \mod 11 \end{cases}$$

Note that all expressions of the system above are quadratic or linear. In fact, we could compact last two constraints into one, resulting in an equivalent R1CS defined over $S \backslash \{s_6\}$:

$$\begin{cases} s_1 \times s_2 - s_5 = 0 & \mod 11 \\ s_5 \times s_3 + s_4 - s_7 = 0 & \mod 11 \end{cases}$$

Compressing all constraints into a single one would not result in an R1CS, since we would end up with a non-quadratic equation:

$$s_1 \times s_2 \times s_3 + s_4 - s_7 = 0 \mod 11.$$

Hence, in this example, any R1CS arithmetic representation of C will always have at least two quadratic constraints.

Since arithmetic circuits are composed by additions and multiplications, the representation of arithmetic circuits as R1CS is a natural transformation. Moreover, a valid witness for an arithmetic circuit translates naturally into a solution of the R1CS representing the circuit. This way, we say that an arithmetic circuit is *satisfiable* if there exists a solution to the R1CS representing the circuit. Checking satisfiability in R1CS encoded form requires to check all gates of a circuit. Most ZK protocols use aggregation techniques, such as *quadratic arithmetic programs*, to check all gates at once [4].

Although there have been tremendous efforts into understanding, developing and improving ZK protocols and ZK-SNARKs, not much work has been done towards formalizing, standardizing, and optimizing the construction of arithmetic circuits. In fact, designing and implementing specific circuits is still a very handcraft procedure that can entail security flaws if done incorrectly, compromising any ZK machinery applied later on a circuit. For this reason, we developed the CIRCOM circuit programming language. With CIRCOM, programmers can implement arithmetic circuits, and the compiler takes care of the R1CS encoding and the rest of elements needed to compute ZK-SNARK proofs.

3 Related Work

The appealing properties of ZK-SNARKs set off the development of software tools that allow practical ways to define to-be-proved statements, and to generate and verify ZK proofs. In this section, we give an overview of the main tools that exist in the ZK-SNARK ecosystem. In Figure 2, we give a visual classification of the tools according to their functionality, and in Table 1, we summarize their features.

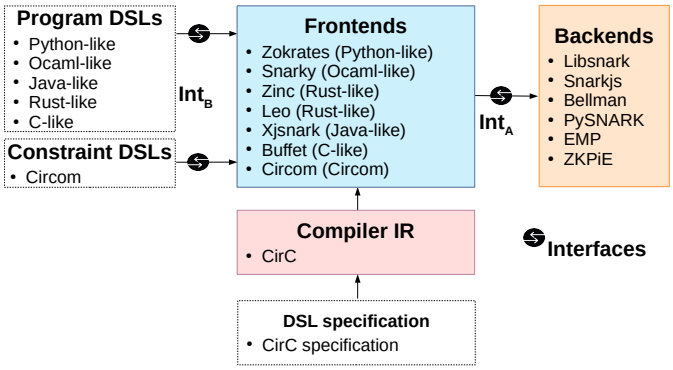

Figure 2: Classification of the main existing tools in the ZK-SNARK ecosystem.

3.1 Libraries, DSLs and Standardization Tools

The first type of tools that were developed were libraries, also known as *backends*. Libraries are written in some general purpose programming language and provide functions that help users describe statements, and also generate and verify ZK proofs. LIBSNARK [24] and BELLMAN [25] were the first libraries that came out. The former is written in C++ and is used as backend by many other tools. The later is a Rust crate for building ZK-SNARK circuits that provides circuit traits, primitive structures, and basic gadget implementations. The second most popular backend in GitHub after LIBSNARK is SNARKJS [18], a JavaScript library that we developed for generating and validating ZK proofs from a set of R1CS constraints. SNARKJS can run in desktops and servers with NodeJS, and since it is

written in JavaScript, it can also run seamlessly in browsers, bringing ZK proofs to the Web. A nice feature of this library is that it creates Solidity code to validate ZK-SNARK proofs within the Ethereum network. PYSNARK [26] is a library for writing zk-SNARKs in Python 3. The library can be used in combination with LIBSNARK and SNARKJS, and it also produces Solidity smart contracts automatically. There are also other backends that were designed for specific application scenarios. For instance, EMP [27] is a library that implements several interactive and communication-efficient ZK protocols [28]–[30] for proving statements in the context of neuronal networks. The protocols implemented by EMP are interactive and not based on R1CS. On the other hand, ZPiE [31], [32] is an implementation of ZK-SNARKs based on R1CS compatible with SNARKJS that is specifically designed for embedded systems. Since the application of these backends is narrowed, they are less popular and widespread than the aforementioned backends.

Over time, DSLs appeared as a more natural way of expressing computational statements being proved in ZK. In these cases, statements are expressed in a higher level DSL and compiled by the corresponding DSL compiler to lower level functions provided by a library. The main advantage of a DSL over a library is that a DSL allows users to express statements in the idiom and at the level of abstraction of the problem domain [33]. Moreover, domain experts themselves (in our context, cryptographers and developers) can more easily understand, validate, modify, and develop DSL programs. Additionally, DSLs allow validation at the domain level, which is performed by a compiler. In the context of ZK, another advantage of a DSL is that the compiler can apply specific techniques to simplify the set of constraints. There are two categories of DSLs [14] for ZK (interface named Int_B in Figure 2): *program-based DSLs* and *constraint-based DSLs* (called *hardware DSLs* in [14]).

In a program-based DSL, a statement is specified as a "program" written with a subset of instructions of a regular programming language that can be converted into primitives provided by the backend. In this case, the frontend compiler transcompiles the program and converts it into a circuit definition consisting of a set of constraints that can be proved by the backend. One of the first practical program-based DSL was ZOKRATES [34], a Python-like DSL with a compiler written in Rust. This frontend was intended to help programmers use verifiable computation in their *decentralized application* (DApp) from the specification of a Python-like program for which proofs can be generated and finally verified by a Solidity smart contract. SNARKY [35] is an Ocaml-like program-based DSL with a backend based on LIBSNARK. To our knowledge, this is the only DSL built on top of a functional programming language. On the other hand, ZINC [36], [37] is a program-based DSL that borrows Rust's syntax and semantics with minor differences. In particular, ZINC does not allow recursion or variable loop indexes. Leo [38] is another Rust-like program-based DSL that abstracts the notions of native/non-native arithmetic and constraint types, which results in more expensive circuits in terms of constraints [38]. XJSNARK is a Java-like program-based DSLs for zk-SNARKs that uses as cryptographic backend a Java interface to LIBSNARK. Finally, the Pepper project is an academic research project that has developed some tools for practical verifiable computation. In particular, their Buffet's C-to-C compiler [39], [40] supports proof-specific optimizations that is used by Pequin [41], a toolchain to verifiably execute programs expressed in the C programming language.

On the other hand, in a constraint-based DSL, the statement being proved is specified directly as a circuit using arithmetic constraints [14]. In this case, the DSL simplifies the task of writing constraints by allowing the definition of small circuits that act as components that can be connected with each other to form larger circuits. The constraint-based DSL compiler ensures that constraints are correctly specified and, like in all DSLs, it can also apply simplification techniques over the set of constraints defining a circuit.

Finally, it is worth to mention that recently, there have been efforts towards creating more generic tools for building and prototyping compilers and also to standardize some of the interfaces from the ZK ecosystem. A notable example is CIRC [14], [42], which is a shared compiler infrastructure for creating frontends that compile to constraint representations. To construct a compiler with CIRC, the developer essentially writes an interpreter for the DSL using the CIRCIFY library. The advantage is that the implementation using the intermediate representation provided by the CIRCIFY library is much shorter and faster to develop than building a full compiler from scratch. CIRC is a very promising tool for quickly creating compilers and for prototyping. Actually, authors have created versions of ZOKRATES and CIRCOM with less lines of code that the original compilers. Regarding the efforts towards standardizing interfaces between frontends and backends, a remarkable initiative is ZKINTERFACE [43], [44], which specifies a protocol for communicating constraints, assignments (witness), and proving protocols. These data are specified using language-agnostic calling conventions and formats to enable interoperability between different authors, frameworks, and languages.

3.2 CIRCOM vs. Related Work

CIRCOM is a constraint-based description language for arithmetic circuits. To the best of our knowledge and according to the available literature [14], CIRCOM is the only implemented DSL of this type. CIRCOM is in a level of abstraction between a library and a program-based DSL.

On the one side, as with libraries, CIRCOM users can specify the constraints of the circuit that defines the to-be-proved statement. Moreover, libraries allow users can create or use already created gadgets (smaller circuits) and connect them with a program written in the language of the library. Similarly, CIRCOM users can make use of templates, which are small circuits that are parameterizable and can be instantiated to form larger circuits. CIRCOM also allows users to create their own custom templates or to use templates from CIRCOMLIB. Among other things, the CIRCOM compiler takes care that template definitions, parameters, interconnections, and the R1CS constraints are correctly defined. Compared to libraries, with CIRCOM, the circuit building process is checked by the compiler and possible errors are shown to users. This way, the CIRCOM language is a simple DSL for specifying templates and their interconnections.

On the other side, CIRCOM also supports splitting the circuit description into a pure proving part (constraints) and

Backend	**Application**		**Language**	**GitHub repository**
Libsnark [24]	Desktop and server		C++	`scipr-lab/libsnark` (1,564 ⋆)
snarkjs [18]	Browser, desktop, and server		JavaScript	`iden3/snarkjs` (1,153 ⋆)
Bellman [25]	Desktop and server		Rust	`zkcrypto/bellman` (675 ⋆)
PySNARK [26]	Python gadget library		Python	`meilof/pysnark` (113 ⋆)
EMP [27]	Interactive protocols		C++	`emp-toolkit/emp-zk` (43 ⋆)
ZPiE [31], [32]	Embedded systems		C	`xevisalle/zpie` (13 ⋆)
Frontend	**Type**	**DSL**	**Compiler**	**GitHub repository**
ZoKrates [34], [45]	Program	Python-like	Rust	`Zokrates/ZoKrates` (1,425 ⋆)
circom [15], [16]	Hardware	circom	Rust	`iden3/circom` (627 ⋆)
snarky [35]	Program	OCaml-like	OCaml	`o1-labs/snarky` (426 ⋆)
Leo [38], [46]	Program	Rust-like	Rust	`AleoHQ/leo` (341 ⋆)
Zinc [36], [37]	Program	Rust-like	Rust	`matter-labs/zinc` (306 ⋆)
xJsnark [47], [48]	Program	Java-like	Java	`akosba/xjsnark` (158 ⋆)
Buffet [39], [40]	Program	C-like	C, C++	`pepper-project/tinyram` (34 ⋆)
Tool	**Description**			**GitHub repository**
CirC [14], [42]	Tool for compilers			`circify/circ` (138 ⋆)
zkInterface [43], [44]	Standard tool for ZK interoperability			`QED-it/zkinterface` (109 ⋆)

Table 1: Open-source zero-knowledge tools and their properties. GitHub stars as for November 2022.

a pure witness computation part. This splitting is necessary when the witness computation requires operations that cannot be expressed as quadratic constraints. With this feature of the language, the compiler can automatically generate a program to efficiently compute a valid assignment to all wires of the circuit (the witness). That is, when we compile a circuit with CIRCOM, the compiler outputs the set of associated R1CS constraints and, if asked, it also can output programs for computing the witness efficiently. In particular, the compiler can output programs written in C++ (to be executed in a desktop/server) and in WebAssembly (to be executed in a browser). In comparison, program-based DSLs provide a higher level of abstraction by allowing users to specify the to-be-proved statement as a program and the compiler transforms the program into a circuit description. To do so, the compiler has to explore all paths through the program, unrolling all loops, considering all branches, while guarding all state modifications by the condition under which the corresponding path is taken [14].

Although a program-based approach might be a right level of abstraction for many programmers, it is worth noting that when writing these programs, programmers cannot completely abstract from the details of the underlying system. To illustrate this, we will make use of an example from the ZOKRATES official documentation [49, Section 3.4]: a circuit that given an input signal x, if $x \neq 0$ then the output is its inverse x^{-1}, and if $x = 0$, then the output is 0. The natural way of writing the circuit in ZOKRATES would be the following one:

```
1 def main(field x) -> field {
2   return if x == 0 {
3     0
4   } else {
5     1 / x
6   };
7 }
```

However, as the official documentation states, the caveat with the previous ZOKRATES code, is that it leads to an execution failing because line 5 is executed even when $x = 0$. The reason for this type of caveats is that, at the end, the program is compiled down to an arithmetic circuit, and hence, jumping on a branch condition does not work as with traditional architectures. For this reason, programmers should still take into consideration the limitations of this type of circuits. By contrast, in CIRCOM, the program can be written as:

```
1  template Inverse() {
2    signal input in;
3    signal output out;
4    signal inv;
5    signal iszero;
6
7    inv <-- in!=0 ? 1/in : 0;
8    iszero <== -in * inv + 1;
9    in * iszero === 0;
10   out <== (1 - iszero) * inv;
11 }
12
13 component main = Inverse();
```

In the previous CIRCOM code, the first two constraints (lines 8 and 9) enforce that the signal named `iszero` is 1 if the input signal `in` is 0, and 0 otherwise (for a detailed explanation of the `iszero` constraints see Section 4.10). The last constraint (line 10), sets the output signal `out` to 0 if the input signal `in` is 0, and with any other number, `out` is set as the inverse of the given number. The differential factor between the CIRCOM code and the ZOKRATES code is that in CIRCOM, the user can explicitly specify how exactly the `inv` intermediate signal is computed in the template (line 7), which avoids the division by zero issue of the ZOKRATES code and allows the witness computation part to run without issues. It is worth saying that the authors of ZOKRATES are currently working on an experimental feature that only activates constraints that are in a logically executed branch. However, this feature comes with a significant overhead of constraints [49].

On the other hand, the expressiveness and flexibility of constraint-based languages may also be a better option for those developers that, in order to create highly optimized circuits, wish to have greater control about the set of signals and constraints that define the computation. In this sense, the spirit of CIRCOM is to provide an unopinionated tool

in which users can use the CIRCOM language to implement their optimizations at the template level. Moreover, as we show in Section 6, the CIRCOM compiler can apply several rounds of simplification of linear constraints, an option that can be activated or deactivated by the user.

Many projects in the Ethereum network are using the low-level approach provided by CIRCOM including payment mixers like Tornado cash [50], anonymous multi-asset pools like Zeropool [51], ZK signaling gadgets like Semaphore [52], public-key cryptographic protocols like ECDSA [53], decentralized ZK-RTS games like Dark Forest [54], and ZK-rollups like Hermez [55], that use circuits described by hundreds of millions of constraints. Moreover, projects like zkREPL [56], which provide an online development environment for zk-SNARKS, are built on top of CIRCOM. Remark that authors of CIRC have developed an alternative compiler implementation for the CIRCOM language. Obviously, their implementation is considerably smaller that our Rust implementation of the compiler. They say that they achieve roughly the same performance as our compiler. In fact, they are referring to our previous version of the CIRCOM compiler (version 0.5) which was a prototype written in Javascript. Indeed, our current version written in Rust is in average 5 times faster than the JavaScript/CIRC versions for small/medium size circuits and can increase up to 10 times faster for large circuits.

4 CIRCOM

CIRCOM [16] is a constraint-based DSL that allows programmers to design and create their own arithmetic circuits for ZK purposes. The CIRCOM compiler is mostly written in Rust and it is open source[1]. It is designed as a low-level circuit language, close to the design of electronic circuits.

4.1 Introduction

CIRCOM allows programmers to define the constraints of an arithmetic circuit in a low-level but friendly way. Recall that arithmetic circuits consist of operations in a finite field $\mathbb{F}_p$ that can be expressed as constraints of the form $\mathbf{a} \times \mathbf{b} - \mathbf{c} = 0$, where **a**, **b** and **c** are linear combinations over a set of signals $\{s_1, ..., s_n\}$. From a CIRCOM circuit description, the CIRCOM compiler outputs the corresponding set of constraints and a program that, given a set of input values, can compute an assignment to the rest of circuit signals. By default, CIRCOM takes p as the order of BN-128 elliptic curve, but the compiler also accepts the large prime dividing the order of BLS12-381, and the Goldilocks-like prime $2^{64} - 2^{32} + 1$. The user can select the prime to be used with a compiler's command-line option.

Example 3. Before going into details, we first illustrate how CIRCOM works with a circuit that will allow us to prove that the product of two secret input signals are equal to a certain public output.

1. The CIRCOM compiler can be installed following the instructions from `https://docs.circom.io/getting-started/installation`. The source code has more than 150K lines of Rust, WebAssembly, and C++, and is available at `https://github.com/iden3/circom.git`.

```
1  pragma circom 2.0.0;
2
3  template Multiplier () {
4     // declaration of signals
5     signal input a;
6     signal input b;
7     signal output c;
8     // constraints
9     c <== a * b;
10 }
```

The first line of this code is a `pragma` instruction that specifies the version of the CIRCOM compiler that is used to ensure that the circuit is compatible with the compiler version indicated after the `pragma` instruction. If it is incompatible, the compiler throws a warning. All files with the `.circom` extension should start with such `pragma` instruction, otherwise, it is assumed that the code is compatible with the latest compiler's version.

In line 3, we use the reserved keyword `template` to define the configuration of a circuit, in this case called `Multiplier`. Inside the template definition, we start by defining the signals that comprise it. Signals can be named with an identifier, in our example, these are identifiers `a`, `b` and `c`. In this case, we have two input signals `a` and `b`, and an output signal `c`.

After declaring the signals, we write the constraints that define the circuit. In this example, we used the operator `<==`. The functionality of this operator is twofold: on the one hand, it sets a constraint that expresses that the value of `c` must be the result of multiplying `a` by `b`; and on the other hand, the operator instructs the compiler in how to generate the program that computes the assignment of the circuit signals. The compiler also accepts the left-to-right operator `==>` with the same semantics, but for simplicity, from now on, we will always use the right-to-left operator `<==`.

4.2 Creating a Circuit

Templates are parametrizable general descriptions of a circuit that have some input and output signals and describe, sometimes using other subcircuits, the relation between the inputs and the outputs. In the previous snippet of CIRCOM code, we created the template called `Multiplier`, but to actually build a circuit, we have to *instantiate* it. The template `Multiplier` does not depend on any parameter, but as we show in next examples, it is possible to create generic parametrizable templates that are later instantiated using specific parameters to construct the circuit. In CIRCOM, the instantiation of a template is called *component*, and it is created as follows (line 10):

```
1  pragma circom 2.0.0;
2
3  template Multiplier () {
4      signal input a;
5      signal input b;
6      signal output c;
7      c <== a * b;
8   }
9
10 component main = Multiplier();
```

By means of the declaration of components and templates, CIRCOM allows programmers to work in a modular fashion: defining small pieces and combining them to create large circuits that can entail millions of operations.

4.3 Compiling a Circuit

As we said in Section 4.1, the use of the operator <== in the template `Multiplier` has a double functionality: it captures the arithmetic relation between the signals, but it also provides a way of computing `c` from `a` and `b`. In general, the description of a CIRCOM circuit also keeps this double functionality. This way, the compiler can easily generate the R1CS describing a circuit but also the instructions to compute the intermediate and output values of a circuit. More specifically, given a circuit with the `.circom` extension the compiler can return four files. For example, we can compile `multiplier.circom` with the next options:

```
1 circom multiplier.circom --r1cs --c --wasm --sym
```

With the previous options, we are telling the compiler to generate a file with the R1CS constraints (*symbolic task*) and the programs for computing the values of the circuit wires in C++ and WebAssembly (*computational task*). The last option tells the compiler to generate a file of symbols for debugging and printing the constraint system in an annotated way.

After compiling a circuit, we can calculate all the signals that match the set of constraints of the circuit using the C++ or WebAssembly programs generated by the compiler. To do so, we simply need to provide a file with a set of valid input values, and the program will calculate a set of values for the rest of signals of the circuit. Recall that a valid set of input, intermediate and output values is called *witness*.

Regarding the prime number being used, it is included in the header of the R1CS file, so that the backend can appropriately build the proof. The prime specification is also used by the compiler to generate the witness-calculator programs, which are linked to the correct modular arithmetic libraries to efficiently deal with the selected prime modular operations. Currently, we provide support and libraries for the three primes mentioned in Section 4.1.

4.4 Generating a ZK Proof

Imagine we want to show that we know two numbers a and b such that $a \times b = 33$, while keeping a and b private. For that, we could use the previous template `Multiplier` by setting the inputs `a` and `b` as private signals of the circuit, and the output `c` as a public signal. However, by default, the inputs of a CIRCOM circuit are all considered private signals, whereas outputs are always public signals. Hence, we can use the template `Multiplier` already as it is.

In Figure 3 we show the complete process of generating and validating a ZK proof with our architecture. As we can see, we should first create a file containing the inputs written in the standard JSON format: `{"a": 3, "b": 11}`. Next, we pass the file with the inputs to the C++ or WebAssembly program generated by the compiler, which will generate a file containing the witness in binary format. After compiling the circuit and running the witness calculator with an appropriate input, we will have a file with extension `.wtns` that contains all the computed signals and, a file with the `.r1cs` extension that contains the constraints describing the circuit.

With the witness and the R1CS files, we can compute and verify ZK proofs using SNARKJS. All ZK protocols implemented in SNARKJS require a trusted setup. In some cases, it is possible to reuse a trusted setup, like in [19],

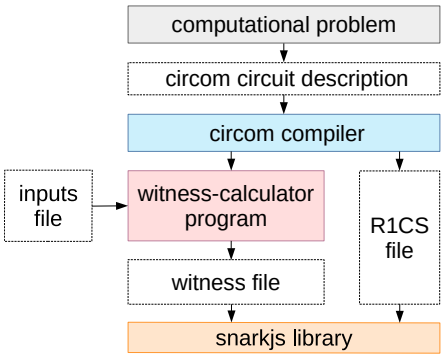

Figure 3: Our architecture for generating and verifying ZK-SNARK proofs using CIRCOM and SNARKJS software tools.

whereas in others, it is necessary to generate a new trusted setup per circuit, as in [4] and [6]. For this reason, SNARKJS already provides the necessary commands to create MPC ceremonies for generating the trusted setup and also verifying that an existing trusted setup has been computed correctly. From the R1CS and the MPC, SNARKJS produces a generation and a verification key for the circuit. Finally, with the generation key and the witness, the prover can generate a ZK-SNARK proof and send it to a verifier, who uses the verification key and a file with the public signals of the circuit to check if the prover's proof is valid[2].

Note that we could have started the process choosing different input values. For example, we could have used `{"a": 1, "b": 33}` as input and generated a valid proof for our circuit. Hence, a proof for the circuit `Multiplier` would not really show that we know how to factor 33. In Section 4.10, we will use a template that checks if a signal is zero to modify the template `Multiplier` to only accept inputs that are not 1.

4.5 The Main Component

The CIRCOM compiler needs a specific component as entry point. This initial component is called `main` and, as we did in Example 3, it needs to be instantiated with some template.

Unlike other intermediate components that we will introduce later, the `main` component defines the global input and output signals of a circuit. As mentioned in Section 4.4, by default, the global inputs are considered private signals while the global outputs are considered public. However, the `main` component has a special attribute to set a list of global inputs as public signals. The general syntax to specify the `main` component is the following:

```
1 component main {public [s1,..,sn]} = templateID(v1,..,vn);
```

The `{public [s1,..,sn]}` part is an optional argument that specifies the list of public signals of the circuit. Any other input signal not included in this list is considered a private signal.

Example 4. Let us illustrate the use of public signals following the previous example. For simplicity, we will no longer start out code with the `pragma` instruction.

2. For further information about the creation of a trusted setup and the generation and verification of ZK-SNARK proofs with SNARKJS, we refer the reader to https://github.com/iden3/snarkjs.

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

component main {public [a]} = Multiplier();
```

In this code snippet, we declare the `main` component with the global input `a` as a public input signal, whereas `b` remains as a global private input signal of `Multiplier`.

Recall that the prover needs all signals (private and public) to generate a ZK proof, while the verifier only needs the public signals to verify a proof, which in this case are signals `a` and `c`.

4.6 Connecting Templates

CIRCOM is a modular language that allows the definition of small circuits called *templates*. Typically, templates are then instantiated to form larger circuits. The idea of building large and complex circuits from smaller parts makes it easier to test, review, and audit large CIRCOM circuits.

Example 5. Let us illustrate how to connect templates by continuing our previous example. In Example 3, we created a template for a multiplier of two signals. In this case, we will extend this idea by connecting two of these 2-input multipliers to get a multiplier for three signals.

```
1  include "multiplier.circom";
2
3  template Multiplier3() {
4    signal input in1;
5    signal input in2;
6    signal input in3;
7    signal output out;
8
9    component multiplierA = Multiplier();
10   component multiplierB = Multiplier();
11
12   multiplierA.a <== in1;
13   multiplierA.b <== in2;
14   multiplierB.a <== multiplierA.c; // in1 * in2
15   multiplierB.b <== in3;
16   out <== multiplierB.c; // (in1 * in2) * in3
17 }
18
19 component main {public [in1, in2]} = Multiplier3();
```

In line 3, we create a template called `Multiplier3` that has three inputs called `in1`, `in2` and `in3`, and one output called `out`. Notice that in the instantiation of the template (line 19), we specify that `in1` and `in2` are public, and `in3` is private. To build the multiplication of the three input signals, we create two subcomponents that are 2-input multipliers (lines 9 and 10). To do so, we have to import the definition of the 2-input multiplier template from a separate file using the keyword `include` (line 1). Then, we connect the inputs `in1` and `in2` to the input wires of the first subcomponent `multiplierA` using the dot (`.`) operator. Next, we use the second subcomponent to do the other multiplication by connecting the output of the previous 2-input multiplier (line 14) and `in3` (line 15). Finally, to provide the multiplication of the three inputs, we assign the output of the second 2-input multiplier to the output of `Multiplier3` (line 16).

Remark. From a template, we can only access the inputs and outputs of its direct subcomponents.

4.7 Debugging

The CIRCOM language provides a small logging function that is called with $\texttt{log(arg}_1\texttt{,...,arg}_n\texttt{)}$ that can greatly help users debug their circuits. This function can be called with strings, values of signals, or expressions. This way, when the witness computation program is executed, the console prints the logged values.

Example 6. Following Example 5, we can use the logging function to show a string followed by the value of the signal `multiplierA.c`.

```
log("The result is ", multiplierA.c);
```

4.8 Building More Complex Circuits

In our previous example, we created a template composed of different subcomponents. The capability of building large circuits from smaller pieces is far more powerful in CIRCOM. For instance, we can create parametrized templates using flow control structures like `for` loops and `if` statements, include variables for using them, and even define *arrays of signals* and *arrays of subcomponents*.

Example 7. Let us illustrate how to build more complex circuits by generalizing Example 5 to an n-multiplier. That is, we will create a parametrized template that will allow the instantiation of circuits that will verify the multiplication of n input values.

```
1  include "multiplier.circom";
2
3  template MultiplierN(n) {
4    signal input in[n];
5    signal output out;
6
7    component multiplier[n-1];
8
9    multiplier[0] = Multiplier();
10   multiplier[0].a <== in[0];
11   multiplier[0].b <== in[1];
12
13   for(var i=1; i<(n-1); i++){
14     multiplier[i] = Multiplier();
15     multiplier[i].a <== in[i+1];
16     multiplier[i].b <== multiplier[i-1].c;
17   }
18
19   out <== multiplier[n-2].c;
20 }
21
22 component main = MultiplierN(4);
```

In the previous code snippet, we create a template called `MultiplierN` which depends on a parameter `n`. The template uses an array called `in` of `n` elements to describe the template inputs (line 4). Then, we create `n-1` `Multiplier` subcomponents (line 7), which are referenced with an `n-1`-dimensional array called `multiplier`. Then, we appropriately initialize the first subcomponent (lines 9–11). Next, we use a `for` loop with a control variable called `i`, which is created using the keyword *var*. Notice how inside the `for` loop we create subcomponents and wire the connections between them.

Remark. It is useful to think of building CIRCOM circuits as a similar process of building electronic circuits. With CIRCOM circuits, the compiler must know all the required parameters of the circuit. As a result, in loops that involve constraints (*symbolic* part), CIRCOM only allows to define the loop condition based on the template parameters. In our previous example, the loop condition used `n`, which was perfectly valid. For further information, see Section 4.13.

4.9 Splitting Between Computation and Constraints

In this section, we explain what happens when the calculation of a signal does not come from a quadratic formula. To give some intuition, we start with an example of a template that performs a division.

Example 8. A division $c = a/b$ is an operation that cannot be computed using a quadratic formula but it can be checked using the quadratic expression $a = b \cdot c$.

```
1 template Divider() {
2   signal input a;
3   signal input b;
4   signal output c;
5   c <-- a/b;
6   a === b * c;
7 }
```

In this case, we have to split the *computational task*, which instructs the compiler in how to compute signals, from the *symbolic task*, which instructs the compiler in how to create constraints that verify a computation. As we can see in the code, the computational task is expressed using the individual operator `<--` (line 5). The language also accepts the left-to-right operator `-->` with the same semantics. On the other side, the symbolic task is expressed separately using the individual operator `===`, which adds a constraint that captures the quadratic relation between signals (line 6).

As an implementation detail, just mention that the `===` operator also adds an *assert* to the program that computes the witness. As expected, if after computing a witness there is an *assert* instruction that is not satisfied, the program stops and returns an error. Therefore, the `===` operator also plays a small role in the computational task.

At this point, it should be clear that the following templates `A` and `B` are equivalent:

```
1 template A() {
2   signal input in;
3   signal output out;
4   out <-- in;
5   out === in;
6 }
```

```
1 template B() {
2   signal input in;
3   signal output out;
4
5   out <== in;
6 }
```

These two templates are equivalent because their compilations will produce the same R1CS and the code of the witness computation program will be the same except for the fact that the code from template `A` will have an extra *assert* instruction with respect to the code generated from template `B`. Anyway, in this particular case the *assert* will always be fulfilled, so the witness computation programs are effectively equivalent.

In general, the *dual operator* `<==` is preferred whenever possible, because it always guarantees the equivalence between the computed witness and the constraints that check the computation. Notice that, if not handled with care, the use of the *individual operators* `<--` and `===` might produce a situation in which the witness does not fulfil the constraints or in which the constraints are unconnected to the witness.

Example 9. Let us look at the following template, which given two inputs `a` and `b`, it outputs `c = a+b`.

```
1 template Incorrect() {
2   signal input a;
3   signal input b;
4   signal output c;
5   c <-- a+b;
6   c === a * b;
7 }
```

In this template, the computational program will output `c = a+b`, but the R1CS describing the template will consist of the constraint `c = a*b`. Therefore, given two inputs, the witness computed by the witness computation program will not be correct in general. In this case, only inputs such that `a+b = a*b` will be valid inputs for the circuit. Circuits in which a computation is not reflected as an equivalent constraint, are considered *incorrect* circuits.

To avoid these cases, individual operators must only be used in cases in which the dual operator cannot express a computation like it happened in Example 8. In Section 7, we analyse these situations in greater detail.

Remark. Neither the operator `===` nor `<==` can be used with signal expressions that are not quadratic.

4.10 Checking If Zero

Now that we know the basic syntax of the CIRCOM language, we present the template `IsZero`, which has some subtleties. `IsZero` checks if a certain signal in a circuit is zero or not. In this case, the output signal `out` is 1 if the input signal `in` is zero, and `out` is 0, otherwise. The implementation is based on a trick from [4].

```
1 template IsZero() {
2    signal input in;
3    signal output out;
4    signal inv;
5    inv <-- in!=0 ? 1/in : 0;
6    out <== -in * inv +1;
7    in * out === 0;
8 }
9
10 component main = IsZero();
```

First, we use an intermediate signal `inv` to compute the inverse of the input signal `in`. Since signals of CIRCOM circuits are elements of a prime field $\mathbb{F}_p$, the only element that has no inverse is 0. Hence, if `in` is not 0, we can assign to `inv` the inverse of `in`. In the other case, where such inverse does not exist because `in` is zero, we assign 0 to `inv`. Note that the value of the signal `inv` depends on a conditional expression and hence, we cannot use the operator `<==` and we use the individual computational operator `<--` instead.

After that, we assign the value `-in*inv + 1` to the signal `out` (line 6), which will be `1` if `in = 0` and `0`, otherwise. Since we do the assignment using the dual operator `<==`, the constraint `out = -in*inv + 1` is also added to the R1CS.

Observe that the previous constraint ensures that `out` is 1 if `in` is zero, but if `in` is not zero, the value of `inv` is not captured in any constraint, since its assignment is done only with the individual computational operator. Hence, `inv` could be manipulated to take any value. For this reason, if we want to enforce that `out` is really 0 when `in` is not zero, we add a new constraint `in*out === 0` (line 7).

Note that when `in` is 0, we decided to assign 0 to `inv`, but in fact, we could have chosen any other value. Indeed, when `in` is zero, both constraints (lines 6 and 7) are satisfied. In this case, we say that the circuit is *safe*, but not *strongly safe*, since there is more than one valid solution for `inv`. We analyse this type of situations in greater detail in Section 7.

Example 10. The template `IsZero` is used very frequently. An illustrative example, is to use it to modify our first template `Multiplier` from Example 3 to enforce that none

of its inputs is 1. For that, we use the fact that `a` is not 1 if and only if `a-1` is not zero, and the same stands for `b`.

```
include "iszero.circom"

template Factorization() {
   signal private input a;
   signal private input b;
   signal output c;

   component isz1 = IsZero();
   component isz2 = IsZero();

   isz1.in <== a-1;
   isz2.in <== b-1;
   isz1.out === 0; // enforce that a-1 != 0
   isz2.out === 0; // enforce that b-1 != 0

   c <== a * b;
}

component main = Factorization();
```

4.11 Functions and Constants

The CIRCOM language also allows the usage of *functions* to encapsulate computation logic. Functions in CIRCOM have a syntax similar to functions in the C programming language. In the body of a function, we can use control flow statements and variables. However, functions should only be used for computational purposes, so contrary to circuit templates, functions cannot create new constraints or use signals.

Example 11. An example of a basic function is the following one, which adds one to a given value:

```
function my_function(x){
    return x+1;
}
```

The use of functions is not strictly necessary to define circuit templates and their main usage in CIRCOM is to define global constants. The reason for this, is that CIRCOM does not admit the definition of global constants. Thus, whenever we want to have a global constant we can just define a function that always returns the same value, and call it every time we need it in our circuit.

Example 12. The following function will be later used in Section 5.2 and it returns a parameter of an elliptic curve.

```
function baby_const_a(){
   return 168700;
}
```

4.12 Symbolic Variables

In Section 4.8, we explained several uses of the variables when building circuits. However, variables have another important use, which is to store symbolic expressions when building the constraints. We call *symbolic variables* to those variables that contain symbolic expressions on signals.

Example 13. Let us analyse an example of a template that uses symbolic variables. The following template implements a multiAND circuit that depends on a parameter n. That is, `MultiAND` is a template that takes an array of `n` binary inputs and outputs 1 if and only if all inputs are 1.

```
include "iszero.circom"

template MultiAND(n) {
signal input in[n];
signal output out;
var sum = 0;

for(var i=0; i<n; i++) {
sum = sum + in[i];
}

component isz = IsZero();

sum - n ==> isz.in;
isz.out ==> out;
}

component main = MultiAND(4);
```

In the previous code snippet, we implemented a multiAND gate for four binary inputs (line 15). To do so, we add the values of the inputs and checked if the result was equal to the number of inputs by subtracting and checking if the result is zero. If the result is zero, the output should be one and zero, otherwise.

Notice that we used two variables: `i` and `sum`. The variable `i` is a regular index variable used in the `for` loop, while `sum` is a symbolic variable that is used to create a constraint in which we add up the values of the `n` input signals. Inside the loop, the symbolic variable `sum` is used to create the sum of signals `in[0] + ... + in[n-1]`. In line 9, `sum` is finally used to generate the constraint:

```
in[0] + ... + in[n-1] - n = isz.in.
```

Example 14. In the following example, we analyse a template that given an input signal `in`, it outputs the binary representation of `in` as an `n`-array of signals called `out[n]`. For a given number `n`, we could use the following list of quadratic constraints:

```
out[0] * (out[0]-1) === 0
...
out[n-1] * (out[n-1]-1) === 0

out[0] * 2^0 +...+ out[n-1] * 2^(n-1) - in === 0
```

The first lines guarantee that all elements of the array `out` are binary, and the last line, that `out` is indeed the binary representation of the input `in`. We can rewrite the previous code using a loop:

```
signal input in;
signal output out[n];
var bsum = 0;
var exp2 = 1;

for (var i = 0; i<n; i+=1){
    out[i] * (out[i]-1) === 0;
    bsum += out[i] * exp2;
    exp2  *= 2;
}
bsum === in;
```

Note that, in the previous code, we used the individual symbolic operator `===`. We cannot use the dual operator because the constraints to check the binary representation of `in` cannot be computed using quadratic expressions. For this reason, we need to build the constraints without providing a way to compute their values. This has to be done separately with the following simple algorithm that extracts one by one the bits of `in`:

```
for (var i = 0; i<n; i+=1) {
    out[i] <-- (in >> i) & 1;
}
```

Notice how we used the individual operator for computation `<--` to assign computed values to signals without generating new constraints.

Now, putting the two pieces together, we can implement a circuit template called `Num2Bits(n)` that outputs the bit representation of up to `n` bits of an input signal.

```
1  template Num2Bits(n) {
2    signal input in;
3    signal output out[n];
4    var bsum = 0;
5    var exp2 = 1;
6    for (var i = 0; i<n; i+=1){
7       out[i] <-- (in >> i) & 1;
8       out[i] * (out[i]-1) === 0;
9       bsum += out[i] * exp2;
10      exp2  *= 2;
11   }
12   bsum === in;
13 }
```

Note that in the body of control flow statements we can have both symbolic and computational expressions (lines 7-10). In general, CIRCOM programmers can write constraints and signal computations together, even when the symbolic and computational descriptions differ.

4.13 Dealing with the *unknown*

Recall that, when writing CIRCOM programs, it is useful to think of them as physical circuits of wires and gates. As with physical circuits, CIRCOM circuit descriptions cannot depend on the value of its wires. That is, the R1CS of any CIRCOM program must be the same for any set of inputs. In fact, the compiler builds the R1CS without knowing the values of the inputs, and hence, it considers the values of the signals *unknown* at compilation time. As a result, since Boolean expressions on conditional expressions and loops can only depend on values known at compilation time (i.e. template parameters but no signal values), if we try to add a constraint inside a conditional or a loop that depends on unknown expressions, CIRCOM will output a compilation error.

Formally, a block of code is *unknown* if it depends on a Boolean expression which is unknown at the program point where it was evaluated. For instance, the body of a loop is unknown, if its condition depends on the value of an input. An expression is unknown at a program point pp, if there is a variable involved in the expression which is unknown at pp. Finally, a variable x is unknown at pp if, for a given instantiation of the template, there exists a path in the control-flow graph ending at pp in which, for the last assignment modifying x, the new value depends on an unknown expression or such an assignment belongs to an unknown block.

Notice this definition is recursive and thus, the CIRCOM compiler performs a fixed-point analysis to detect the unknown variables present in the program. A hint for the programmer when getting a compilation error for an unknown variable is to pay attention to two common situations:

1) The addition of a constraint that depends on a Boolean condition involving an unknown variable.
2) The addition of a constraint with an array access using as index an unknown variable or a signal.

Example 15. Let us see an example of a CIRCOM program that does not compile because of the unknown.

```
1  template ErroneousTemplate1(n) {
2    signal input in;
3    signal output out1;
4    signal output out2;
5    for(var i=0; i<n; i++) {
6      out1 <== in  *  in;
7      if(in >= 0){
8        out2 <== in + 2;
9      }
10   }
11 }
12
13 component main = ErroneousTemplate1(4);
```

When compiling this program, we obtain an error derived from the instruction in line 8, where we are trying to add a new constraint to the R1CS only if the value of signal `in` is greater or equal than 0. In this case, the compiler detects that the execution of line 8 depends on the condition from line 7, but signal `in` has an unknown value at compilation time, and hence, the compiler throws an error. Notice that line 6 is correct, since it is inside the loop from line 5, whose Boolean condition depends on the value of `n`, which is a template parameter known at compilation time.

Example 16. In this other example, we illustrate the situation in which, to create a constraint, a symbolic variable (unknown) is used to access an array.

```
1  template ErroneousTemplate2(n) {
2    signal input in[5];
3    signal output out;
4    var aux;
5
6    if(n > 0)
7      aux = in[0] + 3;
8    else
9      aux =  2;
10   out <== in[aux];
11 }
12
13 component main = ErroneousTemplate2(4);
```

Observe that at line 10, the variable `aux` is unknown, since for the given instantiation of the template ($n = 4$), `aux` is modified (line 7) and its new value depends on the value of the signal `in[0]`. Therefore, we will get a compilation error, since the constraint `out = in[aux]` cannot be added to the R1CS without knowing the value of `aux` used to index the array `in`.

As a result of the previous discussion, circuits, which are defined by a set of R1CS constraints, must be known at compilation time. However, in certain occasions, it may be useful to do computations requiring accesses to positions in arrays or memories that unknown at compilation time, e.g. depending on the value of an input signal. When using CIRCOM, the user has to build the circuits that arithmetize this type of computations. These arithmetizations are not built-in features of CIRCOM, because by design, CIRCOM is unopinionated in how arithmetizations are implemented and rather, these arithmetizations should be part of template libraries. In the literature, we find several approaches for such types of arithmetizations. For instance, [57] uses a line-by-line compilation approach with instructions for memory reads and writes, and a hash structure to store the current memory state, while [13] uses a permutation network to verify that the sequence of memory reads and writes is consistent. Buffet [39], [40] uses a combination of [57], and [13] to build an efficient arithmetization of the random access memory.

4.14 CIRCOMLIB

As we have explained in the previous sections, the use of templates allows CIRCOM developers to build large circuits from smaller individual subcircuits. In this regard, CIRCOM users can create their own custom templates, but in addition to the language and the compiler, we also provide an open-source library of CIRCOM templates called CIRCOMLIB [17], with hundreds of different circuits. On the one side, CIRCOMLIB has the implementation of basic operations, such as binary logic gates, comparators, conversions between field elements and their binary representations, and multiplexers. On the other side, the library contains more complex circuit structures that are used in the context of distributed ledgers and cryptocurrencies, such as digital signatures, elliptic curve-based cryptographic schemes, hash functions, and Merkle tree structures. We would like to remark that apart from CIRCOMLIB, there is a community actively using CIRCOM for building their custom own templates. Remarkable examples are an elliptic-curve pairing implementation from 0xParc [58] and a CIRCOM-based library from Electron Labs that allows to generate proofs for a batch of Ed25519 signatures [59]. In the following Section 5, we show how to make use of CIRCOMLIB templates and present some practical applications of CIRCOM.

5 APPLICATIONS

In this section, we present some examples that illustrate the potential of the CIRCOM language. In Section 5.1, we give an example of a circuit that allows us to prove that we know the preimage of a hash value using templates from CIRCOMLIB. In Section 5.2, we introduce templates that implement the arithmetic operations on an elliptic curve called Baby Jubjub [60]. In Section 5.3, we explain how to use the previous curve operations to verify that a private key corresponds to a public key without revealing the private key. Finally, in Section 5.4, we explain how to verify a signature with templates from CIRCOMLIB and give an example of a circuit that verifies that a given message has been signed by a public key from a pair of authorized public keys, but without revealing which of the two was used.

5.1 Hashing

A *cryptographic hash function* is a deterministic one-way function that maps data of an arbitrary size to a bit array of a fixed size. Hash functions are widely used in authentication systems to avoid storing plaintext passwords in databases, but are also used to identify and validate the integrity of files, documents, and other types of data. One of the main uses of hash functions is in digital signatures, where the hash is used to create a cryptographic digest of the data being signed (see Section 5.4).

CIRCOMLIB provides circuits for several hash functions. For example, the template `Sha256(nBits)` is an implementation of SHA-256, which is defined as a hash function

$$\mathcal{H} : \{0,1\}^{\texttt{nBits}} \rightarrow \{0,1\}^{256}.$$

The next example shows a circuit that you can use to prove that you know the preimage of a given hash without revealing it. The following piece of code creates a circuit that takes a binary array `in` of 256 bits and returns $\texttt{out} = \mathcal{H}(\texttt{in})$.

```
1  include "sha256.circom";
2
3  template Main() {
4    signal input in[300];
5    signal output out[256];
6
7    component sha256 = Sha256(300);
8
9    for (var i=0; i<300; i++){
10     sha256.in[i] <== in[i];
11   }
12
13   for (var i=0; i<256; i++){
14     out[i] <== sha256.out[i];
15   }
16 }
17
18 component main = Main();
```

In line 7, we instantiate the template `Sha256(nBits)` with $\texttt{nBits} = 300$. In this case, we have to assign the values of the signal array bit by bit (line 10). Finally, we set each bit of `out` to each bit of the output of the `sha256` component (line 14).

Classical hash functions, such as the family of SHA functions [61], are heavy on bit operations, which makes them very inefficient to implement inside arithmetic circuits. For example, the previous template `sha256` from CIRCOMLIB for an input of 300 bits is described by 29,450 constraints. Recently, there have been efforts to develop new hash functions that optimize their representation inside arithmetic circuits. In this regard, CIRCOMLIB also contains the implementation the Pedersen hash [62] (`pedersen`), two hash functions from the MiMC family [63] (`mimc`, `mimc_sponge`), and Poseidon [64] (`poseidon`).

5.2 Elliptic–Curve Arithmetic

A classical use of ZK protocols is to prove ownership of a public key without revealing the secret key. For that, we need to be able to write the logic of verifying that a given secret key corresponds to a given public key inside an $\mathbb{F}_p$-arithmetic circuit. This logic is usually implemented by means of arithmetic operations of an elliptic curve. In this section, we show how to implement the arithmetic operations of an elliptic curve called *Baby Jubjub* [60], used in the Ethereum blockchain to implement elliptic-curve cryptography inside circuits [22].

Definition of parameters of the curve

Baby Jubjub is an elliptic curve defined over the prime field $\mathbb{F}_p$ with

$$\begin{aligned} p =& 21888242871839275222246405745257275088548364400416034343698204186575808495617, \end{aligned}$$

and described by equation

$$ax^2 + y^2 = 1 + dx^2y^2, \tag{1}$$

with $a = 168700$ and $d = 168696$. More precisely, Baby Jubjub is defined as the set of points $(x, y) \in \mathbb{F}_p^2$ that satisfy Eq. (1) together with a special point, called *point at infinity*, which does not satisfy the equation but still belongs to the curve, and which is typically represented by the point $(1, 0)$.

To avoid replicating the values of a and d from Eq. (1) in every template to the curve, it is useful to define them only once. As we explained in Section 4.11, CIRCOM does not admit the definition of global constants and, instead,

we have to to define two functions that always return these values.

```
function baby_const_a(){
    return 168700;
}

function baby_const_d(){
    return 168696;
}
```

This way, every time we need the coefficients of the elliptic curve, we can call these two functions.

Checking if a point belongs to the curve

We start by checking if a pair of coordinates (x, y) correspond to a point on the curve that safisfies Eq. (1). For that, we create a template called `BabyCheck()`, that verifies if a pair of x and y are a solution to the equation.

```
1  template BabyCheck() {
2    signal input x;
3    signal input y;
4    var a = baby_const_a();
5    var d = baby_const_d();
6    signal x2;
7    signal y2;
8    x2 <== x * x;
9    y2 <== y * y;
10   a * x2 + y2 === 1 + d * x2 * y2;
11 }
```

In the previous template, first, we declare two input signals `x` and `y`, one per each coordinate. Then, we get the values of the coefficients a and d from the functions we previously defined and assign them to two variables `a` and `d`, respectively. Now, note that we cannot write directly the constraint

```
a*x*x + y*y === 1 + d*x*x*y*y,
```

as in Eq. (1), since it is not a quadratic expression. Instead, we use two new intermediate signals, `x2` and `y2`, to represent x^2 and y^2 (lines 8- 9). Once these signals are defined, we can check if the point `(x,y)` belongs to the curve using the quadratic constraint

```
a*x2 + y2 === 1 + d*x2*y2.
```

Alternatively, we could have defined a signal `u`, enforced `u <== x*y`, and then used `u` to rewrite the curve equation using an equivalent constraint of the form

```
a*x2 + y2 === 1 + d*u*u.
```

Addition of points in the curve

Now, we define how to operate in the elliptic-curve group. For that, we use that the addition of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on Baby Jubjub is defined [60] as a third point $P_3 = (x_3, y_3)$ with coordinates

$$x_3 = \frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2} \text{ and } y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}. \quad (2)$$

The following piece of code consists of a template, called `BabyAdd`, that takes two points and outputs their addition using the formula from Eq. (2).

```
1  template BabyAdd() {
2    signal input p1[2];
3    signal input p2[2];
4    signal output pout[2];
5
6    signal beta;
7    signal gamma;
8    signal delta;
9    signal tau;
10   var a = baby_const_a();
11   var d = baby_const_d();
12
13   beta <== p1[0] * p2[1];
14   gamma <== p1[1] * p2[0];
15   delta <== (-a * p1[0]+p1[1]) * (p2[0] + p2[1]);
16   tau <== beta  *  gamma;
17
18   pout[0] <-- (beta + gamma) / (1+ d * tau);
19   (1+ d * tau)  *  pout[0] === (beta + gamma);
20
21   pout[1] <-- (delta + a * beta - gamma) / (1-d * tau);
22   (1-d * tau) * pout[1] === (delta + a * beta - gamma);
23 }
```

In this template, we define points using 2-dimensional arrays of signals. In particular, we have two points as input signals (`p1` and `p2`), and a third point as output signal (`pout`). We also have four intermediate signals (`beta`, `gamma`, `delta`, and `tau`), and two variables (`a` and `d`) with the coefficients from Eq. (1). Since both expressions from Eq. (2) involve a division by signals, we cannot write the formulas directly using the dual operator `<==`. Instead, we first use the individual computational operator `<--` to compute the denominators, and then, use the individual symbolic operator `===` to enforce a multiplicative relation between the numerator and the denominator.

To illustrate the definition of public input signals, let us suppose that we want to use circuit `BabyAdd` to prove that given an initial point P_1 and a final point P_{out}, we know the point P_2 such that $P_1 + P_2 = P_{\text{out}}$, where all the points belong to the curve.

```
component main {public [p1]} = BabyAdd();
```

We indicate that the first point `p1` is public thanks to the tag `public` that precedes the list of public input signals in the declaration of the `main` component. An array of signals must have all elements public or all elements private. In this case, both signals of `p1` (`p1[0]` and `p1[1]`) are public. The two coordinates of the output point `pout` are also public, since they are output signals. Finally, the two coordinates of the second point `p2` remain private, since they do not appear in the previous list.

5.3 Public–Key Cryptography

To build public-key cryptography using elliptic curves, the participants must agree on a publicly known point called *generator*. In this setting, a private key is a randomly chosen scalar and its corresponding public key is computed by multiplying the generator point by the private key. This scheme achieves the properties of public-key cryptography because point multiplication by a scalar can be efficiently computed with algorithms like double-and-add [65, Algorithm 7.6], while computing the private key from the generator and the public key point is computationally unfeasible. This computational problem is widely known as the *discrete logarithm problem* [65, Problem 7.1]. In the following code snippet, we use the `ScalarMulFix` template from CIRCOMLIB to compute a public key from a private key provided as input.

```
1  template BabyPbk() {
2    signal input in;
3    var GEN[2] = [
4      5299619240641551281634865583518297030 2
5      8287447219077289408652114448272100155 3,
6      1695015079846065771795862556782183455 0
```

```
    301663161624707787222815936182638968203
  ];
  signal output Ax;
  signal output Ay;
  component pvkBits = Num2Bits(253);
  pvkBits.in <== in;
  component mulFix = ScalarMulFix(253, BASE8);
  var i;
  for (i=0; i<253; i++) {
      mulFix.e[i] <== pvkBits.out[i];
  }
  Ax  <== mulFix.out[0];
  Ay  <== mulFix.out[1];
}
component main = BabyPbk();
```

Let us identify the main parts of this template. First, we have an input signal `in`, which is the scalar (private key) used to generate the new point (public key); the generator point `GEN[2]`; and two output signals, `Ax` and `Ay`, which are the coordinates of the public key point generated from multiplying `GEN[2]` by `in`.

After these definitions, we declare a component (`Num2Bits`) to transform the scalar `in` to its 253-bit representation, and assign `in` as its input signal. Right after, we declare a new component (`ScalarMulFix`) to perform the multiplication of the generator by the scalar. Details about how this multiplication is performed can be found in [17]. In line 16, each of the bits from the representation of the scalar are set to its corresponding input of the component. Finally, we assign the output signals of this component to the final output signals `Ax` and `Ay`.

If we compile the program without declaring any input signal as public, then they all remain as private signals. In particular, `in` is a private signal whose value should not be known because it is a private key. Finally, note that `BabyPbk()` only has two public output signals, `Ax` and `Ay`, which are not explicitly declared as public, since the output signals of the `main` component are always considered as public signals.

5.4 Digital Signatures

A popular elliptic curve-based signature scheme is the *Edwards-curve digital signature algorithm* (EdDSA) [66], which is a digital signature scheme based on twisted Edwards curves, such as Baby Jubjub. Given a public key as defined in Section 5.3, and a message, the EdDSA protocol uses a public cryptographic hash function to bind the signature to a given message and public key.

CIRCOMLIB has different implementations of EdDSA based on Baby Jubjub which differ in the hash functions being used. The template `eddsa` uses the Pedersen hash, `eddsamimc` is implemented using MiMC, and `eddsaposeidon` is a variation with Poseidon. All these templates output 1 if the signature is valid, and 0 otherwise.

Users can use these templates to validate that a signature of a message is valid, but they can also use them to prove more elaborated statements. For example, that a message has been signed with a public key that belongs to a list of authorized public keys but without revealing which specific one. In the following example, we define a template that validates if a message has been correctly signed by one of two public keys {`pk1`, `pk2`}.

```
include "eddsa-simplified.circom";

template VerifyAuthorizedSignature() {
  signal input pk1[2]; // public key 1
  signal input pk2[2]; // public key 2
  signal input msg; // message
  signal input sig; // signature

  signal out1;
  signal out2;

  component verify1 = EdDSAVerifier();
  component verify2 = EdDSAVerifier();

  // verify signature with pk1
  verify1.pk <== pk1;
  verify1.msg <== msg;
  verify1.sig <== sig;
  out1 <== verify1.out;

  // verify signature with pk2
  verify2.pk <== pk2;
  verify2.msg <== msg;
  verify2.sig <== sig;
  out2 <== verify2.out;

  out1 + out2 === 1;
}

component main {public [pk1[0],pk1[1],pk2[0],pk2[1],msg]}
              = VerifyAuthorizedSignature();
```

Notice that we used the `EdDSAVerifier()` template as a black box that returns a signal that determines if a signature is valid for a given message and public key. Since we need to verify the signature twice, one per each key, the template `EdDSAVerifier` is instantiated in two different components (`verify1`, `verify2`). The constraint `out1 + out2 === 1` imposes that either `verify1` or `verify2` is 1. In other words, this constraint ensures that the message has been signed with one of either `pk1` or `pk2` keys, which are public input signals of the circuit (line 30).

6 CIRCOM PERFORMANCE ON LARGE CIRCUITS

One of the main advantages of CIRCOM is its modularity. With CIRCOM, users can define parameterized independent templates that can later be instantiated and combined to produce large circuits describing complex operations. However, combining components significantly increases the number of constraints describing the circuit. Specially, when connecting the output of a component as an input of another component, the developer needs to introduce linear constraints that capture this binding. This situation is aggravated when working with large circuits, which can entail hundreds of millions extra constraints.

To reduce the amount of constraints describing a circuit, the CIRCOM compiler simplifies the linear constraints. More specifically, the compiler divides the set of constraints into clusters of related linear constraints and then applies the classical Gauss-Jordan elimination to each of them. These optimizations are iterated until it is no longer possible to optimize more linear constraints. The compiler treats clusters independently which allows to parallelize the optimization subprocesses. The compiler runs these optimizations by default but the user can choose to turn them off with a command-line option. Currently, there is also an ongoing work on non-trivial optimization techniques applied to R1CS [67].

6.1 ZK-Rollup Circuits

To evaluate the performance of CIRCOM with large circuits, the language and the compiler have been tested with the ZK-rollup circuits of the Hermez [55] project. A ZK-rollup [10] is a construction intended to increase the scalability of Ethereum by performing calculations off-chain, rolling

Circuit	Number of constraints			Size of `.r1cs` file			Compilation time		
	no-simpl.	simpl.	gain	no-simpl.	simpl.	gain	no-simpl.	simpl.	overhead
ZK-Rollup-256	134,267,317	24,301,347	81.9%	15.7GB	8.5GB	45.9%	12.1min	38.9min	$\times 3.22$
ZK-Rollup-512	197,926,325	37,792,099	80.9%	23.4GB	13.7GB	41.5%	17.5min	58.3min	$\times 3.33$
ZK-Rollup-1024	325,244,341	64,773,603	80.1%	38.8GB	24.1GB	37.9%	28.4min	111.2min	$\times 3.92$
ZK-Rollup-2048	579,880,373	118,736,611	79.5%	69.5GB	44.6GB	35.8%	50.6min	512.5min	$\times 10.14$
ZK-Rollup-2341	652,925,030	134,203,765	79.4%	78.4GB	51.1GB	34.8%	56.5min	618.9min	$\times 10.95$

Table 2: Comparison of different Hermez ZK rollup circuits before and after the CIRCOM compiler applies optimization techniques to reduce the number of constraints describing the circuits.

many transactions up into a single batch, and sending it to the main Ethereum chain for processing in one action. In more detail, a ZK proof is generated off-chain for every batch of transactions and it proves the validity of every transaction in the batch. This means that it is not necessary to rely on the Ethereum main chain to verify each signed transaction.

The key of ZK-rollups is that they allow verification to be carried out in constant time regardless of the number of transactions in the batch. This ability to verify proofs both efficiently and in constant time is at the heart of all ZK-rollups. In addition to this, all transactions' data can be published cheaply on-chain, so that anyone can reconstruct the current state and history retrieving the on-chain data. In the following section, we present some results for Hermez ZK-rollup circuits of different sizes (transactions per batch).

6.2 Performance Results

In Table 2, we show the number of generated constraints, the size of the R1CS file, and the compilation time for different instances of ZK-rollup circuits. We also show their corresponding gains and losses before and after applying the simplification of linear constraints. The results have been obtained from an AMD Ryzen Threadripper 3990X 64-Core Processor with 270GB of RAM (Linux Kernel 5.4.0-80-generic).

As the experimental evaluation shows, in circuits this large, the compiler's optimizations are crucial to handle the huge amount of constraints. For instance, for ZK-Rollup-256, CIRCOM without simplification generates 134,267,317 constraints whose file size is 15.7GB and the time needed for the compilation is 12.05min. On the other hand, the simplification allows us to reduce the number of constraints up to 24,301,347, whose file size is 8.5GB and the compilation time is increased up to 38.92min. Note that in this case, the reduction on the number of constraints is close to 82%, whereas the size of the `.r1cs` file is not reduced in the same proportion. This is due to the fact that constraint simplification often implies the addition of new variables in the remaining constraints.

The cost of simplification is that compilation time increases slightly more than three times. In the other circuits, the gain is similar, a reduction of around 80% in the number of constraints and 40% in the file size, but compilation time increases considerably more when dealing with more than 500 millions of constraints. The reason for this, is that the amount of RAM memory needed in the simplification process reaches a peak of around 750GB, which is far larger than the memory of the machine we used, which has 270GB of RAM, and hence, it needs to use a lot of swap memory. As observed in the table, this fact notably affects the performance in the last two circuits. In this sense, the job of the compiler is to keep a right balance between constraints reduction and the time needed for it.

Note that with circuits from Table 2, CIRCOM produces around 100 million constraints (500 million without simplification). With these numbers, ZK-rollups can handle around 2,000 transactions. Take into account that the software used afterwards to generate and validate ZK-SNARK proofs may also have bounds. In fact, thanks to simplification, we can handle up to 2,341 transactions without exceeding the limit of 2^{27} constraints that SNARKJS can handle. Processing a batch of this size needs less than 2.1 million gas, so with this amount of transactions per batch and the current Ethereum gas limit per block of 30 millions, we have that we can process 32,774 transfer transactions per block, which is around 23 times more transfers than if they were executed directly in the Ethereum blockchain.

7 Analysis

Let $\mathbb{C}^{n\times t\times m}$ be the set of all circuits that can be programmed in CIRCOM with n input signals, t intermediate signals, and m output signals. Given a circuit $C \in \mathbb{C}^{n\times t\times m}$, we denote by $\mathcal{C}(C)$ the set of constraints generated by CIRCOM after compiling C. Let $W : \mathbb{C}^{n\times t\times m} \times \mathbb{F}_p^n \to \mathbb{F}_p^t \times \mathbb{F}_p^m$ be a partial function that takes a circuit $C \in \mathbb{C}^{n\times t\times m}$ and n values for the input signals, such that it returns the t values of the intermediate signals and the m values of the output signals. The function W describes the computation made by the executable code obtained from compiling C after the input values are given. Note that W is a partial function, since not every input of a circuit produces a valid output.

Definition 17 (Correct CIRCOM program)**.** A CIRCOM program $C \in \mathbb{C}^{n\times t\times m}$ is said to be *correct* if for every given values $\vec{i} \in \mathbb{F}_p^n$ for the n input signals of C, we have that: if $\mathcal{C}(C)$ replacing the inputs signals by $\vec{i}$ is satisfiable, then $W(C, \vec{i}) = (\vec{t}, \vec{o}) \in \mathbb{F}_p^t \times \mathbb{F}_p^m$ and $(\vec{i}, \vec{t}, \vec{o})$ is a solution to the system $\mathcal{C}(C)$. Otherwise, we say that C is *incorrect*.

A CIRCOM program is called *strongly safe* when the values computed by the executable code are the unique solution to the R1CS constraint system. However, sometimes this notion of safety involving all signals including the intermediate ones could be too strong for some components, as it happens with the `IsZero` circuit from Section 4.10. In that case, when the value of signal `in` is 0, the computation

sets the intermediate signal `inv` to be also 0, but `inv` could have taking any other value and still satisfy the constraints from the template. For these reasons, there is an alternative weaker notion of safety, which only requires the constraints and the code to meet on inputs and outputs, but not necessarily on the intermediate signals.

Definition 18 (Safe CIRCOM program)**.** A CIRCOM program $C \in \mathbb{C}^{n\times t\times m}$ is said to be *strongly safe* if for every given values $\vec{i} \in \mathbb{F}_p^n$ for the n input signals of c, we have that: if $W(C,\vec{i}) = (\vec{t},\vec{o}) \in \mathbb{F}_p^t \times \mathbb{F}_p^m$, then $(\vec{i},\vec{t},\vec{o})$ is the only solution of $\mathcal{C}(C)$, and it is said to be *safe* if all solutions of $\mathcal{C}(C)$ are of the form $(\vec{i},\vec{t'},\vec{o})$. Otherwise, the program is called *unsafe*.

Note that, by definition, every strongly safe CIRCOM program is also a safe program. Conversely, every safe program can always be converted into a strongly safe program by adding new constraints which enforce that, given the input values, the intermediate and output signals are a unique solution for the program. For instance, template `IsZero` can be converted into a strongly safe template by adding the constraint `inv * out === 0` to the R1CS.

Lemma 19. *A strongly safe* CIRCOM *program is deterministic.*

Proof. From definition 18, we deduce that, given a safe CIRCOM program C and an input $\vec{i}$ for this program, if there exists an output $\vec{o}$ and an intermediate $\vec{t}$ for C, then it must be unique. □

The following results show that many times both kinds of safety are guaranteed by construction.

Lemma 20. *A* CIRCOM *program is strongly safe if it is written without using <-- and --> and all its intermediate and output signals are the target of an assignment operation.*

Proof. Without loss of generality, let us assume the program only uses right-to-left operators. If a program does not use operator <--, the value of a signal can only be assigned using operator <==. At the computational level, this instruction is translated as an assignment where the signal on the left obtains the same value as the value of the expression on the right. At the constraint level, this instruction introduces a new constraint where both sides must have the same value. Consequently, this constraint is guaranteed once the assignment is executed and, since signals are immutable, and they can only have one single value assigned, this constraint remains true. Apart from <==, the operator === also adds new constraints to the constraint system, and an *assert* in the computational level. As a result, either the program has no result for the input or the constraints are guaranteed to be satisfied by the result. □

In a more intuitive way, the previous lemmas are just the consequence that CIRCOM has only three operators: <==, <--, and ===. With these operators, CIRCOM circuits can only do two things: calculations and constraints' definitions. If circuits are only built with the double operator <==, then both, calculations and constraints are equivalent because they derive from the same expression. As a result, in this type of circuits, the witness-calculator program will always produce values that will satisfy the set of R1CS constraints. Problems may arise when calculations and constraints are not aligned. This can only happen when <-- and === are used in the circuits, because their equivalence is not guaranteed by the compiler. In more detail, the use of the === operator imposes an isolated constraint over a set of signals. Here, we have two cases:

1) If the expression involves signals whose computation has been already defined, then it is the backend's job to ensure that the set of constraints is satisfied by the set of computed inputs. That is, given a set of inputs, if the witness-generator program produces a set of signals which do not satisfy the set of constraints, the backend will not be able to produce a valid proof. In other words, the security is guaranteed by the backend.
2) If the === involves a new signal that should be computed (using the <-- operator) but this computation is not defined in the CIRCOM description, the compiler will detect this fact and throw an error.

On the other hand, <-- allows users to compute values beyond quadratic expressions. Here, we also have two cases:

1) If the <-- operator is backed up by an associated constraint or constraints (using ===), then the CIRCOM description is as safe as using the double operator <==.
2) If the <-- operator is not backed up by its associated constraints, then this means that the CIRCOM description lacks constraints. In this case, the backend cannot help the user to detect missing constraints, it just generates proofs that can be verified but that have fewer constraints than might be necessary.

Regarding the field size and potential overflows, recall that CIRCOM informs the backend about the field size that must be used. It does so by including the prime number at the header of the R1CS file. On the other hand, the witness-calculator program is linked by the CIRCOM compiler to the proper modular arithmetic library, so that the computations are performed in the correct field.

8 CONCLUSIONS

In this article we presented CIRCOM, a constraint-based DSL for describing ZK circuits. CIRCOM is in a level of abstraction between a program DSL and a library and, to the best of our knowledge and according to the available literature [14], it is the only implemented DSL of this type. The CIRCOM compiler is responsible for generating all the necessary material to, later, generate and verify ZK-SNARK proofs. The philosophy of CIRCOM is that programmers have full control over the exact construction of arithmetic circuits and the resulting set of constraints which, at the end, are the ones used to build ZK proofs. CIRCOM is modular at many levels, and to deal with extra constraints introduced by the interconnection of templates, we implement several rounds of optimizations of linear constraints in the compiler, which are crucial for the use of CIRCOM in industrial circuits describing real-world problems.

ACKNOWLEDGMENTS

This research is supported by the Ethereum Foundation Ecosystem Support [68], TCO-RISEBLOCK (PID2019-110224RB-I00), PID2021-128521OB-I00, H2020-i3-MARKET, the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31, and by the CM projects S2018/TCS-4314 co-funded by EIE Funds of the European Union.

REFERENCES

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 291–304.

[2] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *Journal of the ACM*, vol. 38, no. 3, pp. 690–728, jul 1991.

[3] O. Goldreich and Y. Oren, "Definitions and properties of zero-knowledge proof systems," *Journal of Cryptology*, vol. 7, no. 1, pp. 1–32, dec 1994.

[4] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013, Best Paper Award.

[5] J. Groth, "Short non-interactive zero-knowledge proofs," in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 341–358.

[6] ——, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 305–326.

[7] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin." in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 397–411.

[8] S. Ma, Y. Deng, D. He, J. Zhang, and X. Xie, "An efficient NIZK scheme for privacy-preserving transactions over account-model blockchain," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 641–651, 2021.

[9] Z. Wan, Y. Zhou, and K. Ren, "zk-authfeed: Protecting data feed to smart contracts with authenticated zero knowledge proof," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[10] V. Gramoli, L. Bass, A. D. Fekete, and D. W. Sun, "Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2711–2724, 2016.

[11] "ZK-Rollups," EthHub. [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/

[12] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, "Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting," in *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 327–357.

[13] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, pp. 781–796.

[14] A. Ozdemir, F. Brown, and R. S. Wahby, "CirC: Compiler infrastructure for proof systems, software verification, and more," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 2248–2266.

[15] M. Bellés–Muñoz, J. Baylina, V. Daza, and J. L. Muñoz-Tapia, "New privacy practices for blockchain software," *IEEE Software*, vol. 39, no. 03, pp. 43–49, may 2022.

[16] Iden3, "CIRCOM: Circuit compiler for zero-knowledge proofs," GitHub, 2020. [Online]. Available: https://github.com/iden3/circom

[17] ——, "CIRCOMLIB: Library of circom templates," GitHub, 2020. [Online]. Available: https://github.com/iden3/circomlib

[18] ——, "SNARKJS: JavaScript implementation of zk-SNARKs," GitHub, 2020. [Online]. Available: https://github.com/iden3/snarkjs

[19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PlonK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Report 2019/953, 2019. [Online]. Available: https://eprint.iacr/2019/953

[20] R. Canetti, "Universally composable security: a new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, 2001, pp. 136–145.

[21] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec, "Design of symmetric-key primitives for advanced cryptographic protocols," *IACR Transactions on Symmetric Cryptology*, vol. 2020, no. 3, pp. 1–45, Sep 2020.

[22] B. WhiteHat, M. Bellés, and J. Baylina, "Baby Jubjub elliptic curve," Ethereum Improvement Proposal, EIP-2494, January 29, 2020. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2494

[23] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 626–645.

[24] Succinct Computational Integrity and Privacy Research (SCIPR) Lab, "libsnark: a C++ library for zk-SNARK proofs." GitHub. [Online]. Available: https://github.com/scipr-lab/libsnark

[25] Zcash, "Bellman," GitHub. [Online]. Available: https://github.com/zkcrypto/bellman

[26] Koninklijke Philips N.V., G. Xavier, and M. Veeningen, "pysnark," GitHub. [Online]. Available: https://github.com/meilof/pysnark

[27] EMP, "Efficient multi-party computation toolkit," GitHub. [Online]. Available: https://github.com/emp-toolkit/emp-zk

[28] C. Weng, K. Yang, J. Katz, and X. Wang, "Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits," Cryptology ePrint Archive, Report 2020/925, 2020. [Online]. Available: https://eprint.iacr.org/2020/925

[29] K. Yang, P. Sarkar, C. Weng, and X. Wang, "Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field," Cryptology ePrint Archive, Report 2021/076, 2021. [Online]. Available: https://eprint.iacr.org/2021/076

[30] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, "Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning," Cryptology ePrint Archive, Report 2021/730, 2021. [Online]. Available: https://eprint.iacr.org/2021/730

[31] X. Salleras and V. Daza, "ZPiE: Zero-knowledge proofs in embedded systems," *Mathematics*, vol. 9, no. 20, 2021. [Online]. Available: https://www.mdpi.com/2227-7390/9/20/2569

[32] X. Salleras, "ZPiE: Zero-knowledge proofs in embedded systems," GitHub. [Online]. Available: https://github.com/xevisalle/zpie

[33] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, dec 2005.

[34] J. Eberhardt and S. Tai, "ZoKrates - scalable privacy-preserving off-chain computations," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1084–1091.

[35] o1 Labs, "snarky," GitHub. [Online]. Available: https://github.com/o1-labs/snarky

[36] Matter Labs, "Zinc v0.2.3," Cryptology ePrint Archive, Report 2019/953, 2019. [Online]. Available: https://eprint.iacr/2020/352

[37] M. Labs, "The Zinc language," GitHub. [Online]. Available: https://github.com/matter-labs/zinc

[38] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith, "Leo: A programming language for formally verified, zero-knowledge applications," Cryptology ePrint Archive, Paper 2021/651, 2021. [Online]. Available: https://eprint.iacr.org/2021/651

[39] R. Wahby, S. Setty, Z. Ren, A. Blumberg, and M. Walfish, "Efficient ram and control flow in verifiable outsourced computation," in *Network and Distributed System Security Symposium (NDSS 2015)*, 02 2015.

[40] Pepper Project, "tinyram," GitHub. [Online]. Available: https://github.com/pepper-project/tinyram

[41] ——, "Pequin: An end-to-end toolchain for verifiable computation, snarks, and probabilistic proofs," GitHub. [Online]. Available: https://github.com/pepper-project/pequin

[42] A. Ozdemir, E. Chen, R. S. Wahby, and N. W. Seo, "CirC: The circuit compiler," GitHub. [Online]. Available: https://github.com/circify/circ
[43] J. Groth, Y. Kalai, M. Venkitasubramaniam, N. Bitansky, R. Canetti, H. Corrigan-Gibbs, S. Goldwasser, C. Jutla, Y. Ishai, R. Ostrovsky, O. Paneth, T. Rabin, M. Raykova, R. Rothblum, A. Scafuro, E. Tromer, and D. Wikström, "Security track proceeding," ZKProof Standards, Berkeley, CA, Tech. Rep., May 2018. [Online]. Available: https://zkproof.org/documents.html
[44] QED-it, "zkInterface, a standard tool for zero-knowledge interoperability," GitHub. [Online]. Available: https://github.com/QED-it/zkinterface
[45] ZoKrates, "Zokrates," GitHub. [Online]. Available: https://github.com/Zokrates/ZoKrates
[46] Aleo, "The Leo programming language," GitHub. [Online]. Available: https://github.com/AleoHQ/leo
[47] A. Kosba, C. Papamanthou, and E. Shi, "xJsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 5 2018, pp. 944–961.
[48] A. Kosba, "xJsnark," GitHub. [Online]. Available: https://github.com/akosba/xjsnark
[49] ZoKrates, "ZoKrates documentation," GitHub Docs. [Online]. Available: https://zokrates.github.io/
[50] D. Khovratovich and M. Vladimirov, "Tornado Privacy Solution. Cryptographic Review. Version 1.1," ABDK Consulting, November 29, 2019. [Online]. Available: https://tornado.cash/audits/TornadoCash_cryptographic_review_ABDK.pdf
[51] ZeroPool, "Privacy solution for blockchain." [Online]. Available: https://zeropool.network/
[52] B. WhiteHat, C. C. Liang, K. Gurkan, K. W. Jie, and H. Robert, "Semaphore." [Online]. Available: https://semaphore.appliedzkp.org
[53] 0xparc, "zk–ECDSA: zk–SNARKs for EcDSA." [Online]. Available: https://0xparc.org/blog/zk-ecdsa-1
[54] Dark Forest, "zk–SNARK space warfare." [Online]. Available: https://zkga.me
[55] Hermez Network, "Hermez whitepaper," October, 2020. [Online]. Available: https://hermez.io/hermez-whitepaper.pdf
[56] zkREPL, "An online playground for zero–knowledge circuits." [Online]. Available: https://zkrepl.dev
[57] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state (extended version)," Cryptology ePrint Archive, Paper 2013/356, 2013, https://eprint.iacr.org/2013/356. [Online]. Available: https://eprint.iacr.org/2013/356
[58] 0xparc, "zk–SNARKs for elliptic-curve pairings." [Online]. Available: https://0xparc.org/blog/zk-pairing-1
[59] E. Labs, "Bringing IBC to Ethereum using zk-SNARKs," EthResearch. [Online]. Available: https://ethresear.ch/t/bringing-ibc-to-ethereum-using-zk-snarks/13634
[60] M. Bellés-Muñoz, B. Whitehat, J. Baylina, V. Daza, and J. L. Muñoz Tapia, "Twisted Edwards elliptic curves for zero-knowledge circuits," *Mathematics*, vol. 9, no. 23, 2021. [Online]. Available: https://www.mdpi.com/2227-7390/9/23/3022
[61] H. Handschuh, *SHA Family (Secure Hash Algorithm)*. Boston, MA: Springer US, 2005, pp. 565–567.
[62] B. Libert, F. Mouhartem, and D. Stehlé, "Tutorial 8," 1016-17, notes from the Master Course Cryptology and Security at the École Normale Supérieure de Lyon. [Online]. Available: https://fmouhart.epheme.re/Crypto-1617/TD08.pdf
[63] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," Cryptology ePrint Archive, Report 2016/492, 2016, https://eprint.iacr.org/2016/492.
[64] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 519–535.
[65] D. R. Stinson and M. B. Paterson, *Cryptography: theory and practice*, 4th ed. Boca Raton, FL, USA: CRC Press, 2019.
[66] S. Josefsson and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)," Internet Research Task Force (IRTF). Request for Comments: 8032, January, 2017. [Online]. Available: https://tools.ietf.org/html/8032
[67] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, and A. Rubio, "Distilling constraints in zero-knowledge protocols," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds. Cham: Springer International Publishing, 2022, pp. 430–443.
[68] Ethereum Foundation Ecosystem Support, "CIRCOM featured project," 2020. [Online]. Available: https://esp.ethereum.foundation/en/projects/circom/

Marta Bellés-Muñoz is a PhD student doing research in security and efficiency of arithmetic circuits for zero-knowledge proofs at Pompeu Fabra University. She received her B.S. degree in Mathematics at Autonomous University of Barcelona and continued her Master studies at Aarhus University, where she focused on the study of elliptic curves and isogeny-based cryptography. Her research interests include optimal design of arithmetic circuits, zero-knowledge proofs, and distributed ledger technologies.

Miguel Isabel received a Ph.D. in Computer Science in 2020 from Complutense University of Madrid, where he also obtained his B.S. degree and his Master studies. He is currently working as an assistant teacher at Technical University of Madrid in the area of computer networks and software-defined networks. His research interests include testing of concurrent programs, deadlock analysis, compiler construction, constraint programming, partial order reduction, and software verification.

Jose L. Muñoz-Tapia is a researcher of the Information Security Group, an associate professor of the Department of Network Engineering of UPC, and the director of the Master program in Blockchain technologies at UPC School. He holds a M.S. in Telecommunications Engineering (1999) and a PhD in Security Engineering (2003). His research interests include applied cryptography, network security, game theory models applied to networks and simulators, and distributed ledgers technologies.

Albert Rubio is Professor in Computer Science since 2008, currently in the Complutense University of Madrid and before in the Polytechnic University of Catalonia (UPC). Since 2004 he has been consecutively principal investigator of funded research projects. He is author of more than 60 papers published in the most prestigious conferences and journals, including LICS, CAV or the Journal of ACM and a chapter of the Handbook of Automated Reasoning. He has been cited more than 2200 times.

Jordi Baylina is one of the most outstanding members of the Ethereum community and part of the White Hat Group, with whom he participated in the DAOs and Parity Multisig hack rescues, recovering more than 300 million euros. He holds a B.S. degree in Telecommunications Engineering from UPC. His research interests are applied cryptography and distributed ledger technologies. He is the co-founder and technical lead of Hermez Network and the main contributor of CIRCOM and SNARKJS.