Extensión de CIRCOM con buses: expresividad y seguridad
Adding buses to circom: expressiveness and security

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Lucas Cuesta Araujo

Directores
Clara Rodríguez Núñez
Albert Rubio Gimeno

Doble grado en Ingeniería Informática - Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Extensión de CIRCOM con buses: expresividad y seguridad
Adding buses to circom: expressiveness and security

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Lucas Cuesta Araujo

Directores
Clara Rodríguez Núñez
Albert Rubio Gimeno

Convocatoria: *Junio 2024*

Doble grado en Ingeniería Informática - Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

May 27, 2024

Acknowledgements

To Clara Rodríguez and Albert Rubio, for the enthusiasm they have conveyed to me on the topic of this project. And to Miguel Isabel, for his patience and invaluable help.

Resumen

Extensión de CIRCOM con buses: expresividad y seguridad

A lo largo de este proyecto se ha desarrollado una extensión de CIRCOM, un lenguaje de dominio específico ampliamente extendido para la definición de circuitos aritméticos orientados a demostraciones de "conocimiento nulo" (protocolos criptográficos que permiten verificar la veracidad de un predicado sin revelar información sobre el mismo). El alcance de dicha extensión incluye la incorporación de sintaxis para poder definir conjuntos de señales personalizados llamados buses, los cuales hacen las veces de los *structs* de la mayoría de lenguajes de programación. Los buses permiten realizar el etiquetado de grupos de señales mediante *tags*, un elemento de CIRCOM gracias al cual se pueden anotar propiedades sobre las señales. Los *tags* funcionan como un tipado débil orientado a reforzar la corrección y seguridad de los circuitos aritméticos generados. Para añadir las nuevas funcionalidades mencionadas a CIRCOM se ha trabajado sobre un fork del compilador original con vistas a incorporar estos cambios a la versión oficial proximamente. También se han llevado a cabo mejoras sobre circuitos de la CIRCOMLIB (la librería oficial del lenguaje CIRCOM) empleando los nuevos buses y *tags* de buses con el objetivo de mejorar la comprensión y seguridad de los programas. Estos circuitos actualizados serán incluidos en la nueva versión oficial de la CIRCOMLIB. Además, se han realizado pruebas de rendimiento con las que se ha comprobado que el uso de buses no tiene un impacto negativo en el tamaño de las pruebas generadas por circuitos de CIRCOM. Una versión extendida de este trabajo se tratará de publicar en una revista internacional de alto impacto, como *IEEE Transactions on Dependable and Secure Computing*.

Palabras clave

Circom, circomlib, compilador, pruebas de conocimiento nulo, circuitos aritméticos, criptografía, tags, buses, Rust.

Abstract

Adding buses to circom: expressiveness and security

Throughout this project, an extension of CIRCOM has been developed. CIRCOM is a widely used domain-specific language for defining arithmetic circuits aimed at zero-knowledge proofs (cryptographic protocols that allow the verification of a predicate's truth without revealing information about it). The scope of this extension includes the incorporation of syntax to define custom signal sets called buses, which function similarly to the *structs* found in most programming languages. Buses allow the labelling of signal groups using *tags*, a feature of CIRCOM that enables the annotation of properties on signals. *Tags* function as a weak typing mechanism aimed at reinforcing the correctness and security of the generated arithmetic circuits. To add the mentioned new functionalities to CIRCOM, work has been done on a fork of the original compiler in view of incorporating these changes into the official version soon. Improvements have also been made to circuits in the CIRCOMLIB (the official library of the CIRCOM language) using the new buses and bus *tags* with the aim of improving the understanding and security of the programs. These updated circuits will be included in the new official version of the CIRCOMLIB. Additionally, the performance tests conducted have shown that the use of buses does to not have a negative impact on the size of the proofs generated by CIRCOM circuits. An extended version of this work will be submitted for publication in a high-impact international journal, such as *IEEE Transactions on Dependable and Secure Computing*.

Keywords

CIRCOM, CIRCOMLIB, compiler, zero-knowledge proofs, arithmetic circuits, cryptography, tags, buses, Rust.

Contents

List of figures

List of tables

Chapter 1

Introduction

We will use this first chapter to provide an overview of the project and its background. The opening section will offer some context to introduce the underlying ideas behind the work done. Later, we will describe the objectives pursued and the tasks completed throughout the development of the project.

1.1 Motivation

Cryptography is the study and practice of secure communication techniques in the presence of adversarial behaviour [1]. With the growing need to protect data, especially with the widespread use of the Internet and the emergence of machine learning, new cryptographic methods are required to ensure robust security for all online activities taking place. In this regard, Zero-Knowledge protocols have been a significant breakthrough.

A Zero-Knowledge (ZK) protocol is a cryptographic method that allows one party, called the prover, to convince another party, called the verifier, that a given statement is true without revealing any additional information beyond the statement's validity [2, 3, 4]. ZK protocols must meet three essential criteria: soundness (ensuring that invalid proofs are accepted only with negligible probability), completeness (guaranteeing that valid proofs are always accepted), and zero-knowledge (disclosing no information to the verifier beyond the fact that the statement is true). ZK protocols have become a favoured solution for boosting security and privacy in distributed ledgers, where transparency is a valued feature [5, 6, 7, 8].

Distributed ledgers are decentralised databases shared and synchronised across multiple locations or participants, eliminating the need for a central authority [9]. Each participant has an identical copy of the ledger, which is maintained through a consensus mechanism, ensuring consistency and transparency. Transactions recorded on the ledger are immutable and publicly visible to all participants, enhancing security and trust. Distributed ledgers are widely used in applications like cryptocurrencies, supply chain management, financial services, voting systems, and smart contracts, providing reliable and tamper-proof records across various industries.

The privacy, transparency, and security features of ZK protocols make them highly attractive for use in distributed ledgers. When implemented in blockchain systems, ZK protocols are additionally required to be non-interactive, meaning the prover can send the proof to the verifier without any further communication, and succinct, fundamental for enhancing the performance, scalability, and privacy of blockchain technologies, making them more practical and efficient for widespread use. SNARK protocols [10, 11] fulfil these requirements, and have been successfully incorporated into blockchains due to their small proof sizes. However, SNARKs are dependent on an initial trusted setup phase [12], which raises security concerns. To address this problem, STARK protocols were introduced in 2018 [13]. They eliminate the need for a trusted setup, but in exchange generate larger proof sizes compared to those of SNARKs, which makes them difficult to implement into blockchains. Recently, recursive proofs [14] have emerged as a promising solution, aiming to combine the benefits of both SNARKs and STARKs while overcoming their respective limitations. They involve applying recursively one zero-knowledge proof (like a SNARK or a STARK) to verify the correctness of another zero-knowledge proof. This recursive application allows the construction of very efficient and scalable cryptographic protocols, which are particularly useful in blockchain technologies and privacy-preserving computation.

Like most ZK protocols, SNARKs and STARKs function within the model of circuit satisfiability [15, 16, 17, 18]. This is a classical NP-complete problem in computer science where it has to be determined if an assignment to the inputs of a circuit exists that makes the output true. When that turns to be the case we call the circuit satisfiable, and otherwise the circuit is called unsatisfiable. SNARKs and STARKs work with a special kind of circuits called arithmetic circuits. The wires of arithmetic circuits carry values from a large prime finite field $\mathbb{F}_p$, and its gates perform the arithmetic operations of adding and multiplying. In a ZK proof, the prover must demonstrate knowledge of private inputs that, combined with public inputs, produce specific outputs within a certain arithmetic circuit. The proof statement is translated into a system of polynomial equations (constraints) within the finite field $\mathbb{F}_p$ symbolising the relationships between the signals of the circuit (operands and results of arithmetic gates). However, representing an arithmetic circuit correctly using constraints is a process that becomes increasingly complex the more intricate the proof statement gets, especially for recursive proofs requiring validation across multiple iterations.

To help cryptographers in this task, high-level domain-specific languages like Zokrates and Leo have been developed [19, 20], providing classical programming instructions tailored for arithmetic circuits and compiling into the polynomial constraints used in ZK proofs. These languages support standard arithmetic operations, logical operations, comparisons, control-flow statements, and custom data structures through classical *structs* for a more structured approach to complex data.

Despite their easy usage, these kind of languages can lead to a loss of control

over the final generated constraint system, potentially making it too large for efficient handling by the ZK protocol. Consequently, many cryptographers prefer other languages like CIRCOM [21, 22], which offers precise control over constraints but sacrifices some expressiveness. CIRCOM does not support custom types using *structs* and only provides signals, arrays, and tuples. Despite these limitations, CIRCOM is widely used in multiple projects due to its scalability. The use of languages like CIRCOM enhances efficiency but also increases the risk of failing to meet specifications or introducing security flaws. Bugs in the arithmetic circuit definition, especially in under-constrained circuits where more solutions than intended are allowed, pose significant security risks.

Large circuits with thousands of constraints are prone to vulnerabilities, so CIRCOM simplifies the task of writing constraints by enabling the creation of small, modular circuits called *templates* that can be interconnected to form more complex systems. These templates can be custom-built by programmers or sourced from ZK libraries like CIRCOMLIB (the official CIRCOM library) [23], which offers a variety of pre-audited arithmetic circuits, such as comparators, hash functions, digital signatures, and converters. Despite these safeguards, errors can still occur if templates are not used correctly. For example, a vulnerability in the Baby-Jubjub elliptic curve library has been identified involving the handling of private keys in nullifier computations. The Baby-Jubjub curve uses a prime number r for defining the order of a subgroup where the public-key computation from a private key x is performed. If a malicious user employs an alternative private key x' related to the original private key x by $x' = x + i \cdot r$ (where i is a number from the finite field $\mathbb{F}_p$), they can generate identical public keys for different private keys and distinct nullifiers for the same transaction, potentially enabling undetected double spending.

With such problems in mind, the CIRCOM 2.1.0 update introduced *tags* [24], a basic form of type annotations for circuit signals, allowing programmers to express the intended behaviour of a signal. For instance, by using tags we can assign a max value label to the input signal `in` so other programmers using the template can understand that the values of `in` have to be confined to the interval $(0, max)$, as demonstrated in the `babypbk` template of figure 1.1.

Although at first glance it may seem the code should work fine, a compilation error will arise because the component `bby` instantiated from the template `babypbk` receives as input the value of the signal `key`, which does not include the *max* tag. Thus we cannot be sure if `key` will have values greater than r. To solve this problem, we have to replace line 6 with the commented code in lines 7-9. What the template `MaxValueCheck(r)` will accomplish is adding the *max* tag to the `key` signal by imposing the necessary constraints to guarantee that the signal adheres to a maximum value of r. Notwithstanding, CIRCOM will also allow us to attach the tag *max* to the private key directly, like in figure 1.2.

That code will not issue any compilation error, but the underlying security flaw will still be there, as no additional constraints have been implemented to ensure that

```
template main() { //beginning of template's definition
  signal input pvt_key; //input signal of the circuit
  signal output pbl_key; //output signal of the circuit
  var r = ...; //normal variable storing the prime number for
     the curve
  signal key; //intermediate signal
  key <== pvt_key; //connection between key and pvt_key
  //component maxvCheck = MaxValueCheck(r);
  //maxvCheck.in <== pvt_key;
  //key <== maxvCheck.out;
  component bby = babyPbk(r); //instantiation of babypbk
     template circuit
  bby.in <== key; //connection between key and the input signal
     of bby
  pbl_key <== bby.out; //connection between pbl_key and the
     output signal of bby
}

template babypbk(r) { //this circuit computes the public key
    from the private key
  signal input {max} in; //input signal with maximum value tag
     attached
  assert(in.max < r);
  //...standard code for the public-key computation
}
```

Figure 1.1: Example of a CIRCOM program using tags.

```
signal {max} key;
key.max = r;
key <== pvt_key;
```

Figure 1.2: Example of how CIRCOM allows the addition of tags.

the `key` value is less that r. It is important to note that CIRCOM compiler only carries out syntactic checks. The onus is on the programmer to verify the semantics of tags.

As mentioned, tags are an excellent tool to annotate the expected properties of individual signals. But despite their great potential of expressiveness, they cannot represent relationships between multiple signals. To show this idea, let us take a look at the template from the CIRCOMLIB in figure 1.3.

```
1 template Edwards2Montgomery() {
2   signal input in[2]; //array of input signals
3   signal output out[2]; //array of output signals
4
5   //tells the compiler how to compute the value of signals out
        [0] and out[1]
6   out[0] <-- (1 + in[1]) / (1 - in[1]);
7   out[1] <-- out[0] / in[0];
8
9   //declaration of constraints without associated computations
10  out[0]  *  (1-in[1]) === (1 + in[1]);
11  out[1]  *  in[0] === out[0];
12 }
```

Figure 1.3: Example of what CIRCOM tags cannot be used for.

The `Edwards2Montgomery` template works on the assumption that the signal array `in[2]` represents a point of the Baby-Jubjub elliptic curve in twisted Edwards form [25, 26]. Therefore, the template will not function as intended if these signals do not conform to the expected format, compromising the security of the entire arithmetic circuit. Up to this moment CIRCOM did not offer any feature that could allow programmers express semantic relationships amongst multiple signals. Our goal in this project is trying to solve this problem with the implementation of *buses*, a set of related signals. Just like in real electronic circuits, a bus will extend the functionality of individual signals, allowing programmers to use them as inputs, outputs or intermediates in templates. Additionally, they will let the association of semantics embodied by tags to groups of signals, providing a tool to solve problems such as the one shown in the previous example. Let us see how that error could be avoided with the implementation of buses in figure 1.4.

We have modified the template to now accept a bus of the type `Point` as input and return another `Point` as output. The input `Point` is required to implement the *edwards_point* tag as a way to ensure it will indeed be a point in twisted Edwards form, and the output `Point` will denote its Montgomery form through the *montgomery_point* tag. This way, if the input bus adheres to the specified requirements, the output will always be granted to meet its own specifications.

```
1 bus Point {
2     signal x;
3     signal y;
4 }
5
6 template Edwards2Montgomery() {
7     Point input {edwards_point} in;
8     Point output {montgomery_point} out;
9
10    out.x <-- (1 + in.y) / (1 - in.y);
11    out.y <-- out.x / in.x;
12
13    out.x  *  (1-in.y) === (1 + in.y);
14    out.y  *  in.x === out.x;
15 }
```

Figure 1.4: Example of a CIRCOM template using buses.

1.2 Objectives

The final goal of all the work presented here is implementing buses in the CIRCOM language as a means to provide two new features: the creation of custom signal types, improving the expressiveness of the language, and the bonding of tags to groups of signals with the purpose of annotating relationships between them. This will improve the safety of circuits developed in CIRCOM by allowing programmers to annotate expected properties involving multiple signals. We will accomplish this task by implementing the syntax to enable the use of the aforementioned buses. In addition, we want to apply the benefits of buses to the circuits in the CIRCOMLIB, the CIRCOM official library, in view of the release of a new version. In order to do that, we will have to prove the correctness of the new modified circuits. Our targets can be broken down into the following specific objectives:

I) Equipping the CIRCOM language with the necessary syntax to define groups of signals called buses, serving as the hardware equivalent of *structs*.

II) Extending the use of tags to bus types as a way to express group attributes.

III) Updating CIRCOMLIB templates with the new CIRCOM add-ons to improve its clarity and reliability.

IV) Verifying the correctness of the new template versions in the CIRCOMLIB.

V) Testing the impact of buses on the sizes of the ZK proofs generated by circuits using them.

1.3 Work description

When I started working on this project, the first thing to do for me was learning the characteristics of the CIRCOM language, which are quite unique. After getting

used to the syntax of CIRCOM, I began working with the base code of the CIRCOM compiler written in Rust. Rust is a general-purpose programming language designed for systems development, like OS and compilers, with a strong emphasis on memory safety. Its unique characteristics make the language have a very steep learning curve. It took me a while to get familiar with the Rust syntax and concepts like the borrow checker. The first part of the compiler that I worked on was the parser, where I defined the syntax of the new bus feature that would be added to CIRCOM. Then we moved to the definition of the AST nodes and other objects that would encapsulate all the information related to buses. After this stage, I developed a series of static analyses over the AST in order to ensure the correct use of buses. While the rest of the parts of the compiler were being modified, I focused on the revision of circuits in the CIRCOMLIB to create the new official version of the library coming out. I adapted the templates to the addition of buses and the association of tags to them. To ensure everything would work as expected, I proved the correctness of some of the circuits affected by the modifications carried out, but found potential bugs that will be examined in the future. Finally, once a functional version of the compiler was complete, I tested the new updated versions of the CIRCOMLIB circuits to have a first glance at the benefits and drawbacks brought by buses.

The project's code can be found in the forked repository https://github.com/Sacul231/circom. The `/tests/circomlib` folder contains all the modified versions of the CIRCOMLIB templates, while the parts of the compiler developed during the project are distributed between the folders `/parser`, `/program_structure` and `/type_analysis`.

1.4 In the following chapters

Having introduced the topic and goals of this project, the chapters that follow will be used to further elaborate in the background of ZK proofs and CIRCOM as well as to explain the details of the work done. In the second chapter we will see in depth what a zero-knowledge proof is and why is it so useful. We will describe how do we work with ZK proofs as arithmetic circuits and the features offered by CIRCOM and CIRCOMLIB to encode and generate ZK proofs. Finally, an overview of the parts that make up the CIRCOM compiler and their purposes will be given.

The third chapter will begin with a discussion on the need to introduce buses in CIRCOM, followed by an explanation of its implementation in the CIRCOM compiler. We will go step by step in the compilation process to show each modification needed. Some parts of the Rust code will be exposed to help the reader understand the ideas discussed.

In the fourth chapter we will look at the updated versions of some of the files in the official CIRCOM library, the CIRCOMLIB, the reasons why they had to be modified and some proofs of their correctness. Chapter 5 will be used to present the results in performance obtained with the updated versions of some CIRCOMLIB templates.

Finally, we will present our conclusions on the work carried out and suggest new lines of research.

Chapter 2

State of Art

The purpose of this chapter is to provide the reader with an understanding of the background surrounding CIRCOM and its practical applications. We will begin by introducing the concepts of Zero-Knowledge proofs and arithmetic circuits, followed by an overview of the CIRCOM language and its compiler.

2.1 Zero-knowledge proofs

One of the main issues concerning the current cryptographic protocols is revealing sensitive information to prove some statement. Imagine you want to book your holidays in advance through an online website. Although the company may let you delay the time to pay for later, you will be required to provide your credit card number as "evidence" of having a bank account. This number will be stored in a central database, which is vulnerable to hacking attacks and information leaks.

Zero-knowledge proofs represented a breakthrough in this regard, as they aim at improving security of information for individuals. A zero-knowledge proof offers a means to validate the veracity of a statement without disclosing the statement itself [2, 3, 4]. In our previous example, ZK protocols would empower the company to check the existence of your bank account but the proof would not reveal the credit card number. The 'prover' is the party trying to prove a claim, while the 'verifier' is responsible for validating the claim. To understand how this process works let's take a look at the following classical example from "How to Explain Zero-Knowledge Protocols to Your Children" [27].

In this story Ali Baba, a merchant from the Eastern city of Baghdad long time ago, founds himself robbed in the bazaar. He follows the thief into a cave whose entryway forks into two paths, both of which end in a dead end, and loses track of the thief. For the next 39 days he keeps getting robbed at the bazaar, but the thieves mysteriously disappear at one of the caves' ends. Finally he decides to unveil the secret of the cave and hides under some sacks at the end of the right-hand passage. He discovers the thieves could open the walls of the cave by whispering the magic words 'Open sesame'. Moreover, he finds out the secret door connects

the left-hand passage with the right-hand passage. After some experimentation Ali Baba manages to change the magic words and writes down his discovery with clues about the new password in a manuscript.

In the present the manuscript is found by US researchers who manage to recover the new magic words. The mysterious cave is also located, and one of the researchers, a certain Mick Ali, wants to demonstrate he knows the secret. However, he has no intention of revealing the magic words, so he comes up with a plan. First, a television crew films a detailed tour of the cave with the two dead-end passages. Later, everybody exits the cave, and Mick Ali goes back in alone through one of the passages. Then the reporter, accompanied by the camera, enters the cave only as far as the fork. There he flips a coin to choose between right and left. Depending on the result of the coin flip, he tells Mick to go out on the left or the right, and Mick does just that.

If Mick Ali did not know the magic words and could not open the door, there would be a 50% chance of him being exposed. Each new test divides by two the chances of success for someone who ignores the secret. By repeating this experiment n times, the probability that Mick does not know the magic words but keeps coming from the correct passage is 2^{-n}. On the other hand, the secret allows Mick to come out each time by the required exit. So after several tries, the television crew is convinced that Mick knows the code despite not knowing it themselves.

A zero-knowledge protocol must satisfy the following principles:

- **Completeness**: The protocol will always return 'true' if the input is valid.
- **Soundness**: It will be almost impossible, with an arbitrary small margin of probability, for a dishonest prover to mislead an honest verifier into accepting an invalid statement.
- **Zero-knowledge**: The verifier gains no additional information about the statement beyond its validity or falsity; they possess "zero knowledge" regarding the statement. This condition also prevents the verifier from deducing the original input (i.e., the contents of the statement) from the proof.

In the previous example, interaction between the prover and the verifier was needed as well. While revolutionary, the practical utility of interactive proving was limited due to the requirement for both parties to be continuously available for interaction. Even if a verifier was convinced of a prover's honesty, the proof lacked independent verification capability, as generating a new proof necessitated a fresh set of messages between the prover and verifier. In response, Manuel Blum, Paul Feldman, and Silvio Micali proposed the first non-interactive zero-knowledge proofs [10], where the prover and verifier share a key. This enables the prover to demonstrate his knowledge of certain information without providing the information itself [11].

Unlike interactive proofs, non-interactive proofs entail only a single round of communication between the participants. Here, the prover employs a specialised algorithm to compute a zero-knowledge proof based on the secret information, which is then transmitted to the verifier. The verifier, in turn, employs another algorithm to confirm that the prover possesses the secret information.

Non-interactive proving reduces communication between the prover and verifier, thereby enhancing the efficiency of ZK-proofs. Additionally, once generated, a proof becomes available for verification by anyone with access to the shared key and verification algorithm. Amongst this type of zero-knowledge proofs we find the so-called ZK-SNARKs.

ZK-SNARK stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. A ZK-SNARK protocol fulfils the same requirements as the rest of zero-knowledge protocols, with the addition of being non-interactive and having succinct proof size regardless of the size of the statement to prove. Most ZK-SNARKs also guarantee short verification times. However, not everything about ZK-SNARKs is perfect. The setting up of this kind of protocols requires an initial phase called trusted setup. This step is dependent on some values generated randomly, and the exposure of the random values could compromise the security of the whole scheme. To improve the security of the protocol, most implementations carry out this setup phase applying multi-party computation (MPC), in which multiple independent parties collaboratively to build up the trusted setup parameters. In this process, it is enough that one of the participants destroys its secret counterpart of the contribution to keep the whole scheme secure.

ZK-SNARK protocols are mostly employed to verify the correctness of a computation. In this context, computations can be represented by defining them as arithmetic circuits [15].

2.2 Arithmetic circuits

Arithmetic circuits work just like normal electronic circuits but with numbers from a finite field instead of electric signals. They consist of a set of wires connected to gates that perform operations of addition and multiplication modulo p, where p is usually a large prime number of approximately 254 bits [26]. Within the signals of the circuit we can distinguish between input signals, intermediate signals and output signals. Arithmetic circuits are used in the context of ZK-SNARK proofs because they provide a language to encode ZK proofs as statements involving the inputs and outputs of a circuit. Circuit satisfiability (determining whether the input signals of a given circuit have an assignment that results in a certain output) is a classical NP-complete problem of computability theory. Thus, finding a correct assignment for the inputs of a circuit that matches the given outputs is very expensive computationally. This fact makes arithmetic circuits a fantastic option for ZK protocols [16, 17, 18].

Usually, in ZK-SNARK proofs intermediate signals are only known to the prover, while output signals are public both to the prover and verifier. Amongst the input signals, both public and private ones can be found. Therefore, the prover must provide evidence that, with the public information available, he knows a valid assignment to the rest of signals that makes the circuit satisfiable, named a *witness*.

The encoding of this arithmetic circuits has been an area of study in recent years [28]. A representation is required that both simulates accurately the underlying arithmetic circuit and does not add new possible solutions that could compromise the security of a cryptographic protocol using it. One of the most widespread ways of encoding is an algebraic form called Rank-1 Constraint System (R1CS). An R1CS encodes a program as a set of conditions or constraints over some variables, so that a correct execution of a circuit is equivalent to finding a satisfiable variable assignment. Due to the interchangeable use of arithmetic circuits and R1CS, programs specified in R1CS are often referred to as circuits, and their variables as signals.

R1CS use quadratic constraints to represent arithmetic circuits. Formally, a quadratic constraint over a group of signals $S = \{s_1, ..., s_n\}$ is an equation of the form

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

where $a_i, b_i, c_i \in \mathbb{F}_p \; \forall i \in \{1, ..., n\}$. We can summarise this equation as $A \times B - C = 0$, with A, B and C representing linear combinations of the signals $s_1, ..., s_n$ over $\mathbb{F}_p$. A Rank-1 Constraint System (R1CS) is then defined as a finite collection of quadratic constraints over S.

Before continuing, we are going to examine an example of an arithmetic circuit and its R1CS form to get familiar with both concepts.

Example 2.2.1. Let C be the circuit from figure 2.1 where values of signals belong to the finite field $\mathbb{F}_{11}$. The circuit outputs the result of the operation

$$s_1 \times s_2 \times s_3 + s_4 \times s_5$$

The arithmetic circuits we will be working with use gates which always have two input signals and one output signal. Therefore, to calculate the value of the output signal s_9, C uses three multiplication gates and one addition gate, requiring three intermediate signals s_6, s_7 and s_8. All in all, C is a circuit defined by the values of the set of signals

$$S := \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7,\ s_8,\ s_9\}$$

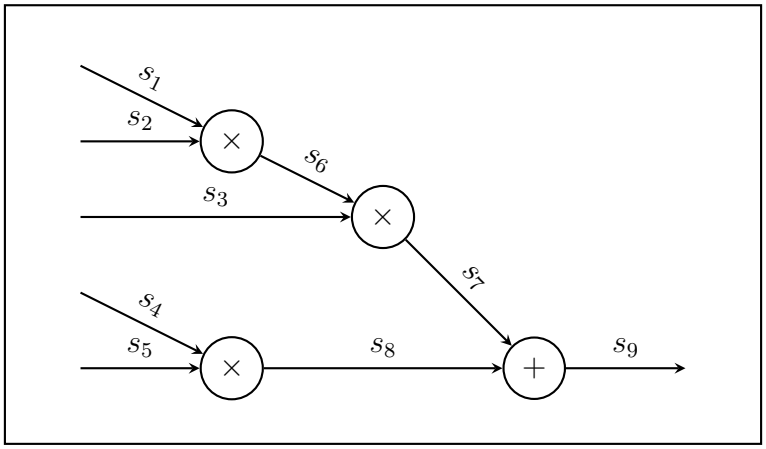

Figure 2.1: Representation of an arithmetic circuit C defined over the finite field $\mathbb{F}_{11}$ that outputs the result of the operation $s_1 \times s_2 \times s_3 + s_4 \times s_5 \; mod \; 11$.

An example of a witness for C could be the set of values

$$w = \{1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 6,\ 9,\ 10\}.$$

We cannot take equation $s_1 \times s_2 \times s_3 + s_4 \times s_5 \; mod \; 11$ as the R1CS representation of circuit C because it does not have the shape of a quadratic constraint. To get a proper R1CS form of the circuit we need to split the equation in intermediate steps expressed as constraints.

$$\begin{cases} s_1 \times s_2 - s_6 = 0 \;\; mod \; 11 \\ s_3 \times s_6 - s_7 = 0 \;\; mod \; 11 \\ s_4 \times s_5 - s_8 = 0 \;\; mod \; 11 \\ s_7 + s_8 - s_9 = 0 \;\; mod \; 11 \end{cases}$$

This representation can be further simplified as follows

$$\begin{cases} s_1 \times s_2 - s_6 = 0 \;\; mod \; 11 \\ s_3 \times s_6 - s_7 = 0 \;\; mod \; 11 \\ s_4 \times s_5 + s_7 - s_9 = 0 \;\; mod \; 11 \end{cases}$$

Here we have joined two constraints by replacing the intermediate signal s_8 with the equivalent expression $s_9 - s7$. We can do this because the new constraint keeps the structure of a quadratic constraint. If that were not the case we would not have been able to reduce the number of constraints in the system.

R1CS are a very natural way of describing arithmetic circuits since they rely on additions and multiplications. In fact, every transformation over a set of signals performed by an arithmetic circuit can be described with enough quadratic constrains. Therefore, if a signal assignment provides a solution for the constraint system, the same assignment will be valid for the circuit, making it satisfiable. Proving satisfiability in R1CS encoded form requires to check all gates of a circuit. With this in

mind, most ZK protocols use aggregation techniques, such as quadratic arithmetic programs, to check all gates at once [15].

There exits a group of software tools developed for designing arithmetic circuits and creating ZK proofs. They can be divided in two main groups. On one hand we find libraries, also known as *backends*. Libraries are typically written in a general purpose programming language and offer features that allow users to define statements, as well as to generate and validate ZK proofs. On the other hand we have domain specific languages (DSL), in which users can express statements in a higher level language that is later translated by the corresponding DSL compiler to lower level functions provided by a library. DSLs provide a more natural manner of designing ZK proofs since statements are formulated at the level of abstraction of the problem domain. As a result, domain experts (in our context, cryptographers and developers) can more easily understand, validate, modify, and develop programs [21]. Two kinds of DSLs for ZK can be distinguished: program-based DSLs and constraint-based DSLs.

In a program-based DSL, statements are defined as "programs" written in a regular programming language. The instructions of the program are then converted into primitives by the backend system, and the frontend compiler transcribes the program into a circuit definition comprising a set of constraints. Meanwhile, a constraint-based DSL requires users to code their statements as circuits using arithmetic constraints. Writing large constraint systems is streamlined by enabling the creation of small circuits that function as modular components, which can be interconnected to form larger circuits as in any hardware description language like VHDL. The constraint-based DSL compiler may apply simplification techniques to the circuit's constraint set, reducing the size of the statement generated. In general, constraint-based DSLs produce much smaller ZK statements than program-based DSLs as they provide programmers with a greater control over the constraint system [21].

2.3 CIRCOM

CIRCOM is a contraint-based domain specific language for designing arithmetic circuits in R1CS form [21, 22]. Despite its low-level approach, letting programmers specify and modify each constraint of the statement to be proved, it aims at making the process of designing large circuits as simple as possible. In this regard, CIRCOM borrows from hardware description languages the idea of defining smaller pieces of the circuit as modules that can be interconnected to describe more complex behaviours. CIRCOM's modules are named templates, conveying the meaning of defining a circuit mould with some parameters that can later be instantiated with concrete values of its parameters. CIRCOM refers to these template instantiations as components.

Amongst its unique characteristics is its capability to divide the circuit description in two different parts: one for constraints and the other for witness computation. This division becomes necessary when witness computation involves operations beyond quadratic constraints. The ability to split the two parts enables the compiler to automatically generate a program to efficiently compute the witness of a ZK proof. When compiling a circuit with CIRCOM, the compiler outputs the associated R1CS constraints and can additionally provide programs in C++ or WebAssembly for computing the witness.

In addition, we mentioned in section 1.1 how CIRCOM 2.1.0 introduced *tags* as a mechanism to augment security measures and ensure type safety within arithmetic circuits [24]. Tags in CIRCOM serve as simple type annotations for input and output signals associated to semantic meanings that help define and constrain the data being processed. The integration of tags facilitates the compiler's ability to verify the correctness of connections between signals. For instance, if a particular input in a subcircuit requires a tag, any signal connected to this input must also exhibit the same tag, ensuring type consistency across the circuit. This feature primarily adds a syntactic level of checking where the compiler ensures that the tags on connected signals match as specified. We will illustrate this idea with a simple example:

```
template Bits2Num(n) {
    signal input {binary} in[n];
    signal output out;
    var lc1=0;

    var e2 = 1;
    for (var i = 0; i<n; i++) {
        lc1 += in[i] * e2;
        e2 = e2 + e2;
    }

    lc1 ==> out;
}

tempate Example {
    signal output num;

    signal bits[2];
    //signal {binary} bits[2];
    component b2n = Bits2Num(2);

    bits[0] <== 0;
    bits[1] <== 1;
    //(bits[0] - 1) * bits[0] === 0;
    //(bits[1] - 1) * bits[1] === 0;
    b2n.in <== bits;
    num <== b2n.out;
}
```

Figure 2.2: Example to understand the purpose of tags in CIRCOM.

Template `Bits2Num` converts a number in binary form represented by an array of signals to its $\mathbb{F}_p$ counterpart (it is important to recall that the signals of an arithmetic circuit carry values from a large prime finite field). Therefore, `Bits2Num` assumes that the values of `in[n]` signals will be either 0 or 1. To communicate that assumption to other circuits using `Bits2Num` its input signal `in[n]` is defined with the tag *binary*. But as we can see the signals of `bits[2]` in the `Example` template do not exhibit the *binary* tag, so this code will result in a compilation error although the values we are assigning to `bits[0]` and `bits[1]` are 0 and 1 respectively.

We could solve this issue by adding the tag *binary* to the declaration of `bits[2]`, as done in line 19. However, the underlying semantic problem would persist since we have included no new constraints to check if the signals of `bits[2]` carry values other than 0 or 1. If we replaced the values assigned to `bits[0]` and `bits[1]` with 3 and 4, we would get no compilation errors despite not fulfilling the preconditions of `Bits2Num`. The constraints we should add to perform that verification can be seen in the commented lines 24 and 25.

It is critical to note that while the compiler enforces syntactic correctness, it does not delve into verifying the semantic correctness of the tags. The responsibility to validate that the tags accurately represent the underlying semantics of the data and adhere to the intended logic of the circuit rests with the programmer. This approach underscores the importance of the programmer's role in not only designing the circuits but also in ensuring that the semantic integrity of the tags is maintained, thus preventing potential security vulnerabilities that could arise from improper implementations.

2.4 CIRCOMLIB

We have already explained how CIRCOM promotes the use of templates to define small individual circuits that later can be combined to form larger ones. In this regard, CIRCOM users have two different alternatives. On one hand, they can design their own templates. On the other hand, they can draw on circuits provided by libraries. And amongst those libraries, one of the most reliable and widely used ones is CIRCOMLIB [23].

CIRCOMLIB is an open-source library that enhances the functionality of CIRCOM. This library includes a wide range of pre-built and rigorously audited circuits, allowing developers to construct complex cryptographic protocols more efficiently by leveraging existing components rather than creating each element from scratch. The circuits available in CIRCOMLIB cover various common cryptographic operations. For instance, it includes implementations of some of the most frequently used cryptographic hash functions like MiMC, Poseidon, and SHA-256. It also provides circuits for elliptic curve operations, such as point addition and scalar multiplication, which are essential for modern cryptographic schemes like digital signatures and key

exchange protocols. Additionally, CIRCOMLIB offers components for verifying digital signatures, basic arithmetic operations in finite fields, and integer comparison circuits.

One significant advantage of CIRCOMLIB is the fact that all its circuits are thoroughly tested and audited, ensuring a high level of security and reliability. This reduces the risk of introducing errors or vulnerabilities into cryptographic applications, a crucial characteristic for maintaining the integrity and trustworthiness of these systems. CIRCOMLIB also promotes a modular approach to circuit design. Developers can combine smaller, well-defined components provided by the library to build larger and more complex circuits. This modularity not only simplifies the development process but also improves the maintainability and scalability of the circuits.

CIRCOMLIB is commonly found in blockchain projects to implement ZK-SNARKs and other Zero-Knowledge proof systems. These applications often require privacy-preserving transaction protocols, and CIRCOMLIB's pre-built circuits make it easier to achieve these goals. Beyond blockchain, CIRCOMLIB is also used for creating privacy solutions like secure voting systems, private identity verification, and confidential data sharing, as well as in scalability solutions that enhance the performance of blockchain networks by enabling efficient off-chain computations that can be verified on-chain.

In the final stages of this project we will look at some examples of circuits from the CIRCOMLIB repository and how they could benefit from the use of buses. We will modify some of them and show the advantages of the updated templates implementing buses, which will be released with the new official version of the CIRCOMLIB. Lastly, we will put the new circuits to the test to check if buses affect the performance of CIRCOM programs.

2.5 The CIRCOM compiler

The goal of this project is to expand the CIRCOM language by adding the new bus feature, which will improve the language expressiveness and enhance the safety of all code generated. In order to do that some parts of the CIRCOM compiler will need to change so as to equip it with the necessary tools to recognise and compute buses correctly [29]. To carry out this task we will first need to understand how the CIRCOM compiler works and what are its component parts. We will take a look at each stage of the compilation process and show the characteristics of the software involved [30].

The first thing the compiler must do is dividing the characters that make up a program into tokens, which are the units of meaning of the program. Afterwards, the compiler will assess the syntactical correctness of the program in a process called parsing. An incontextual grammar must be defined so as to create the pushdown automaton (PDA) that will perform the parsing. In the case of the CIRCOM com-

piler, both of this stages are joined together in a file that directly defines the lexicon and the LR(1) grammar of the CIRCOM language using the tools provided by the Rust crate "lalrpop".

During the parsing, the compiler is also building a tree whose nodes correspond to the expressions, statements and declarations of the program, known as the Abstract Syntax Tree (AST). Each node contains the information of the part of the program it represents. After the parsing process is finished and the AST is assembled, a series of static analyses are performed on the nodes of the tree. The CIRCOM compiler includes the usual binding analysis, where every identifier is attached to its declaration, and typing analysis, helpful for finding type-mismatch errors. In addition, other more specific static analyses are carried out, including:

- **Return statements in functions**: CIRCOM allows the declaration of usual functions to perform complex operations over a set of numerical values. All functions must return a value or array of values, so a return statement must be found in each of its possible paths of execution.
- **Functions free of template elements**: As mentioned before, CIRCOM functions are only able to perform operations over numerical values. Therefore, it is not possible for a function to declare signals or components inside its body. This static analysis ensures that this restriction is met.
- **Signal declaration**: A circuit structure cannot change over time or be dependent on the values of its signals. This is why CIRCOM does not allow the declaration of signals inside while statements. In CIRCOM, signals must be declared at the top scope of templates, and this analysis will check templates to not declare signals inside an inner block. The only exception are if-else blocks, where signals can be declared as long as the value of the condition can be known at compilation time.
- **Constant propagation**: For performance reasons and other important issues, variables detected to be constant are computed and propagated through the AST.
- **Unknown-Known analysis**: We brought up in the *signal declaration analysis* the importance of keeping circuits' structure stable. If a component were instantiated using the values of signals as arguments of its template, or the size of an array were determined by signals, we would not be able to guarantee that the structure of the circuit would not be altered. In this analysis, each expression in a CIRCOM program is classified as *Known* or *Unknown* at compile time. Hence the compiler will issue an error if a constraint declaration depends on an *Unknown* expression.

The static analyses phase is followed by a rather special symbolic execution of the code. Here, the compiler will disassemble the component structure of the program to gather all constraints that form the ZK proof statement in a single list. To do that, it must first compute the concrete values of each expression used as a template

parameter in a component instantiation so as to know the actual structure of the component. Notice that all of these expression have been previously checked to be known at compilation time thanks to the unknown-known analysis. At the same time this process takes place, the compiler will create the Directed Acyclic Graph (DAG). This graph will contain information on every component and its associated template. That will be of great help for the compiler to know which components have the same template and parameter values linked to them, since this information can be used to reduce the amount of memory needed for the instantiation of templates.

After the DAG is created, all of the constraints will be extracted from their components, and the compiler will try to simplify the system without changing its semantics to reduce the size of the statement. Finally, an intermediate representation of the program is generated to later be used to compute the witness in the desired back-end language.

Originally the CIRCOM compiler was written in JavaScript, a general-purpose programming language widely used in web software because of its multiple features but with some flaws related to performance and security. Through versions from 0.0 to 2.0 the compiler continued the use of this language for its software files [31, 32]. Then, in version 2.0, it was decided to rewrite the official CIRCOM compiler in Rust, a systems programming language focused on performance and safety. This is the language we will be working with.

Rust has a lot of fancy characteristics that allow its usage in many different kinds of projects [33, 34]. Perhaps the most important of those characteristics is its concept of ownership and borrowing. *Ownership* is a set of rules that govern how a Rust program manages memory. With the aim of avoiding *dangling references* (pointers to locations in memory that may have already been given to someone else) the solution that Rust brings to the table is having a single owner for each piece of data. When that data has to be used by other part of the program, either the owner has to change or a borrowed reference has to be created. The rules regarding ownership and borrowing can resemble the algorithms applied in concurrent problems such as the classic readers and writers. In fact, Rust also has strong concurrent applications. Since no garbage collector is needed to ensure the safety of references, Rust performance ranks higher than that of most of its peers.

The Rust language offers many other interesting features, such as pattern matching and ideas from functional programming and the object-oriented paradigm. Perhaps it is not the easiest programming language to start working with, as its radically different ideas about references and types can sometimes seem little intuitive or arbitrary, but once they are internalised memory safety is almost 100% guaranteed.

Chapter 3

CIRCOM Development

In this chapter, we will take a look at the implementation of the new bus data structure in the CIRCOM language. We will start by defining precisely what we want to add to the syntax of the language and why do we need this new feature. Afterwards, we will describe what changes have been made to each part of the compiler in order to achieve the desired result. Finally, we will cover the details of the implementation and the dificulties found during the whole process.

3.1 Definition of buses

Usually, when we refer to a bus of signals we want to express the idea of a bunch of signals grouped together. This concept is very useful in many different ways. Not only it allows us to define a custom data type that lets us package together and name multiple related values that make up a meaningful group, as a *struct* or *structure* would do in most programming languages, but it also opens the door to defining semantic statements applied to the whole bus that can be used to ensure the circuit's safety.

```
1 include "compconstant.circom";
2
3 template AliasCheck() {
4     signal input {binary} in[254];
5
6     component compConstant = CompConstant(-1);
7     for (var i=0; i<254; i++) in[i] ==> compConstant.in[i];
8     compConstant.out === 0;
9 }
```

Figure 3.1: Template that checks if a number in binary format has an alias in $\mathbb{F}_p$.

To further motivate the importance of defining buses, let us take a look at figure 3.1, which displays a CIRCOMLIB circuit used in many projects. The purpose of this circuit is adding the necessary constraints to ensure a number N coded in binary

with 254 bits belongs to the finite field $\mathbb{F}_p$. The question is, how could this not be possible? Imagine we are working with three-bit numbers and our finite field is $\mathbb{F}_5$. We will not have any problems with the numbers $0 = (000)_2$, $1 = (001)_2$, $2 = (010)_2$, $3 = (011)_2$ and $4 = (100)_2$. However, although we can represent number 5 with our three bits as $(101)_2$, that number does not belong to our field, it has an 'alias', which is 0, the representative of the equivalence class 0 *mod* 5.

To solve this issue, the template `AliasCheck` uses an instance of another template called `CompConstant`. This second template checks whether a number in binary notation is greater than the parameter of the component, and returns 1 if that happens to be the case or 0 otherwise. `AliasCheck` creates an instance of `CompConstant` with the parameter -1 to compare the input number N with $p-1$. If $N > p-1$ it means N is not a valid number of our field. In our example, since $5 > 4$, `AliasCheck` will include a constraint `compConstant.out === 0;` that will not be met by the input 5 (remember we are working with numbers from $\mathbb{F}_p$).

The problem with this solution lies in its implementation. Considering we do not have a way of attaching a tag to a group of signals, we will be forced to undergo this test for every new number in its binary representation, although that number may come as the result of operations within $\mathbb{F}_p$ over other numbers that have been checked, and so the property of belonging to the field is preserved. We will be adding unnecessary constraints that will increase the size of the ZK proofs. In this context, the addition of buses to the syntax of CIRCOM shines.

Let us see how we could change the above code using the concept of buses. First, we will have to establish the syntax used for defining bus types. We have chosen to work with the one shown in figure 3.2.

```
1 bus BinaryNumber {
2     signal {binary} bits[254];
3 }
```

Figure 3.2: Bus type for binary numbers.

Our new bus will represent a number in binary notation. We have stated the bus to have an array of 254 signals implementing the tag *binary* (they should only carry the values 0 and 1). Right now it seems that we have not gained anything by defining this bus type *BinaryNumber*, as we could represent the binary number with the same array of signals we have declared inside the bus. But we now can attach a tag to the *BinaryNumber* bus, so everyone using it later will know the number represented by the binary code actually belongs to $\mathbb{F}_p$.

The *unique* tag added to the *out* declaration encapsulates the semantics of being the representative of its equivalent class in $\mathbb{F}_p$ and not having a different representation. This characteristic of *out* is forced by the constrains of `AliasCheck`, and can

```
include "compconstant.circom";

template AliasCheck() {
    signal input {binary} in[254];
    BinaryNumber output {unique} out;

    component compConstant = CompConstant(-1);

    for (var i=0; i<254; i++) {
        in[i] ==> compConstant.in[i];
        in[i] ==> out.bits[i];
    }

    compConstant.out === 0;
}
```

Figure 3.3: Template that checks if a number in binary format has an alias in $\mathbb{F}_p$ using buses.

be later assumed by any other template using the value of *out*.

Now that we have motivated the inclusion of buses into the CIRCOM language, it is high time to pore over the modifications of the CIRCOM compiler we need to make in order to allow their use.

3.2 Implementation of buses: Parser

As mentioned in section 2.5, the CIRCOM compiler consists of multiple parts that work together to infer the semantics of the program, detect possible errors and produce code in some backend language. We will go through each of those parts and the tweaks made to implement the bus construction.

Let us begin with the parser. To enable CIRCOM users to define their own bus types and declare buses of signals for the defined types, the lexicon and grammar of the language must be extended with new rules. We will start by focusing on the rules for defining new bus types. A new keyword *bus* must be included to indicate the beginning of the definition of a bus type. The identifier of the bus type will be written after the *bus* keyword, followed by an optional list of parameters inside parenthesis. These parameters will allow the programmer to create different instantiations of the same bus type instead of having to define a new type for each instantiation. An example where this feature could be useful is when defining a bus type containing an array of signals. Perhaps we want to have objects of that bus type with arrays of different sizes, but without the aid of parameters we would be forced to define a new type for each length. Since buses function as signals, the compiler needs to know at compilation time their internal structure because the arithmetic circuit's shape can not depend on the values of signals. Consequently, the values of bus parameters have to be known at compilation type. We will cover this fact

in more detail in section 3.4. For the purpose of making bus syntax less verbose we will allow the definition of buses without parenthesis with the same meaning as writing empty parenthesis. The last part of the bus definition will display the body of the bus, a list of signal, array and bus declarations inside brackets denoting the fields that the bus will encompass. The described rule looks like this:

$$\text{BusDefinition} \rightarrow \text{`}bus\text{'}\ \text{Identifier}\ (\text{`(' ParametersList `)'} \mid \epsilon)\ \text{`\{' Body `\}'}$$

Our new rule will be connected to the definition rule in the sense that the non-terminal *BusDefinition* will be generated by the non-terminal *Definition* as happens with *TemplateDefinition* and *FunctionDefinition* as well.

$$\text{Definition} \rightarrow \text{BusDefinition} \mid \text{TemplateDefinition} \mid \text{FunctionDefinition}$$

Another rule is needed for declaring buses of a certain type. When we declare signals in CIRCOM we use the keyword *signal* followed by the group to which the signal belongs (which can be *input* for input signals, *output* for output signals or none for intermediate signals), a list of tag names inside brackets (can be omitted if the signal has no tags attached) and the name of the signal. We will use the same syntax for bus declarations, but replace the *signal* keyword by the identifier of the bus type we want to instantiate and, if needed, the arguments passed as its parameters:

$$\text{SignalDeclaration} \rightarrow \text{Identifier}\ (\text{`}input\text{'} \mid \text{`}output\text{'} \mid \epsilon)\ (\text{`\{' TagsList `\}'} \mid \epsilon)\ \text{Identifier}$$

$$\begin{aligned}\text{BusDeclaration} \rightarrow \text{Identifier}\ &(\text{`(' ArgumentsList `)'} \mid \epsilon)\ (\text{`}input\text{'} \mid \text{`}output\text{'} \mid \epsilon) \\ &(\text{`\{' TagsList `\}'} \mid \epsilon)\ \text{Identifier}\end{aligned}$$

Like we did with the definition rule, we have connected the new rule to the rule for declarations.

$$\begin{aligned}\text{Declaration} \rightarrow \text{BusDeclaration} \mid \text{ComponentDeclaration} \mid \text{SignalDeclaration} \\ \mid \text{VarDeclaration}\end{aligned}$$

We will also need a rule for accesses to the fields of a bus variable. However, since we are using the same dot syntax as for accesses to input and output signals of components, we will use the already defined rules for component accesses, and thus

we will not add any further rules on this matter. We have covered every possible case of use of our new CIRCOM feature. The CIRCOM compiler has all the necessary tools to build up the Abstract Symbol Tree (AST) with new nodes related to buses. That is our next stop in the compilation process.

3.3 Implementation of buses: AST

The abstract syntax of a programming language refers to a simplified, structural representation of the program's source code, stripped of unnecessary details such as specific punctuation and formatting that are present in the concrete syntax (the literal syntax used when writing the code). The abstract syntax focuses on the logical structure of the code, highlighting the relationships and hierarchy between different elements of the program. This syntax is encoded in the AST, so its rules will serve as the guidelines to build the nodes of the tree. In order to represent bus types and instances in the AST, we have to define new nodes that will contain all the information needed to later perform the desired static analyses and the rules to construct them. In other words, we need to define the abstract syntax of buses.

When a bus type is defined, the information of its parameters and fields must be stored with a view to accessing it whenever we need to perform some analysis. For example, if we want to determine whether an access to a bus field is correct, we will check if the type of that bus contains a field with the same identifier as the one in the access studied. To keep that information, we will define a new AST node `BusDefinition` with attributes where we will store all the characteristics of the bus type. Since nodes like `FunctionDefinition` and `TemplateDefinition` store similar information for functions and templates respectively, we will copy their structure in `BusDefinition`.

Apart from other attributes which are not important for our discussion, four are the features essential to store: the name of the bus type, the list of parameters of the bus type, the body with the declarations of the fields and the list of those fields. However, the list of bus fields is not assembled during the AST construction, so we will have to create it in a later step. Thus, the constructor of the node (which we will call *buildBus*) will receive the first three attributes as arguments and will build the corresponding `BusDefinition` node.

$$buildBus : \text{Identifier} \times \text{ParametersList} \times \text{Statement} \longrightarrow \text{BusDefinition}$$

We have only defined one rule for bus definitions in the non-contextual grammar of our compiler, so the abstract syntax `BusDefinition` node will always be constructed with the equivalent abstract rule:

$$\begin{aligned} \text{BusDefinition} \rightarrow\ & \text{Name ParametersList Body} \\ & \text{BusDefinition}.AST = buildBus(\text{Name}, \text{ParametersList}, \text{Body}) \end{aligned}$$

In the same way that we saw for the parser context-free grammar, the new rule will be connected to the definition rule because the `BusDefinition` node will be a type of `Definition` node, like `TemplateDefinition` and `FunctionDefinition`.

$$\begin{aligned} \text{Definition} &\rightarrow \text{BusDefinition} \\ \text{Definition}.AST &= \text{BusDefinition} \end{aligned}$$

After the AST construction, we will have to call a function *findFields* over the `Body` statement that will put together the list of fields in our bus type. This function will do a tree traversal over the AST nodes starting at `Body`, and will add to the list each signal or bus declaration found in the process. Finally, the list will be stored as an attribute of our `BusDefinition` node. Internally, this list will also be stored as a hash map to enable constant time accesses to the fields of the bus, but we will not go any further into the details of implementation.

Before continuing, it is worth mentioning that we have also be forced to change the abstract syntax of templates, as now they accept both buses and signals as inputs and outputs. Although a minor change, during the process of finding the input and output lists of a template, the equivalent template function of *findFields* will now look for declarations of buses as well, and not just signals like it did previously.

We will now describe how we are going to store information on bus declarations. Due to reasons of code reuse we have decided to internally break up declarations of buses of the form `Point(3) input {tag} p;` in two different statements `bus p;` and `p = Point(3) input {tag};`, similarly to how templates are instantiated as components. In fact, the code behind both types of declarations has many common parts. This approach has led us to the following abstract syntax rule:

$$\begin{aligned} \text{BusDeclaration} &\rightarrow \text{BusVariableDeclaration BusTypeInstantiation} \\ \text{BusVariableDeclaration} &\rightarrow \text{Identifier Type Dimensions} \\ \text{BusTypeInstantiation} &\rightarrow \text{Identifier ArgumentsList} \end{aligned}$$

Here we have divided the declaration in two nodes of different types. The node `BusVariableDeclaration` is a declaration type node that will contain the usual information of this kind of nodes: type of variable, name of the variable, dimensions (in case it is an array), etc. The node `BusTypeInstantiation` corresponds to an expression node associated with the type of the bus instantiated with concrete parameters.

Having decided the representation of bus definitions and declarations inside the AST, it is time to have a look at the static analyses required to ensure a correct and safe use of buses in CIRCOM programs. Throughout the next section we will examine each of those analyses and motivate their importance in the compilation process.

3.4 Implementation of buses: Static analyses

Circom is a cryptography-oriented language; programmers using the language expect their programs to have a high level of security. However, programmers make mistakes, and in the case of ZK-proofs those mistakes can cause huge problems. We cannot just let them rely on their ability to detect errors. Moreover, with the addition of new functionalities it is even more likely that those mistakes appear in the code when trying to get used to the latest add-ons. Therefore, the compiler has some responsibility towards programmers to help them detect errors as early as possible.

Static analysis refers to the process in which the code of a program is analysed without its execution. Almost every modern compiler executes some sort of static analysis during compilation to help programmers detect errors and ease the task of debugging their code. In our case, those analyses are all the more necessary for the reasons mentioned above. With the addition of our new bus circom feature, up to four static analyses need to be implemented or modified to guarantee our code is free of the most common mistakes. We will have a look at each of them in order of execution.

3.4.1 Symbol analysis

The first analysis run by the circom compiler is the symbol analysis. This analysis is in charge of ensuring every identifier used in the program can be linked to an object visible in the current scope. When an attempt is made to use a symbol declared in an inner scope or apply a function that has not been previously defined, the symbol analysis will fail and an error message pointing to the identifier not defined will be thrown. The symbol analysis will also check that no identifier is used in two different definitions whatever the objects being defined so that coherence will be preserved and ambiguous identifiers will be avoided. Additionally, this analysis will ensure that all functions, templates and now buses are called or instantiated with the correct number of arguments.

A reader familiarised with compilers might think the previous analysis is no different thing than the usual binding static analysis found in most compilers. While both play a similar role, a normal binding analysis would link each identifier with its declaration or definition after checking it has been correctly used, but our symbol analysis has been relieved of that responsibility, which is performed by other processes. Nevertheless, the algorithm behind these two analyses is the same.

To keep track of the identifiers already defined or declared previously in the code, we need to make use of an environment, which you can imagine as a stack containing tables of symbols. Each time the algorithm enters an inner scope, for instance within a while loop, an empty table of symbols is pushed onto the environment stack to hold the identifiers of the declarations found in that scope. When a new identifier is declared, first the environment checks whether the symbol has already

been defined to throw an error message, and otherwise stores it in the current table of symbols. Then if a statement uses a certain identifier, the environment will search for that identifier in the current table of symbols, if it is not found there it will keep searching in the previous table of symbols, and this process will continue until the identifier is found in a table of symbols or the environment reaches the base of the stack. When the latter happens an error has been found because an identifier has been used without declaring it first, but the analysis will carry on in search of more possible errors. This whole process works because in a certain scope the identifiers allowed to be used are the ones declared within that scope or in outer scopes. Consequently the environment can search for an identifier in the symbol table at the top of the stack and all the ones below it. After the entire scope has been analysed the table of symbols representing that scope is popped from the top of the stack because those identifiers will no longer be available for use.

In the case of our symbol analysis, apart from checking if an identifier can be linked to a declaration or definition, the compiler will compare the structure of the statement where the symbol was found with its definition to look for possible mistakes in the number of parameters of a function or the arguments to instantiate a template. This analysis will be also necessary for buses as their types can be specified with parameters as well. On top of that, there is a special case with buses that can cause errors which were previously impossible, and that is when a tag attached to a bus uses the same name as one of the bus's fields. Remember that tags are labels used to tie semantics to signals, but we have not mentioned they can be associated with numbers to express properties such as carrying values smaller than a certain number. To access the value of a tag or assign a value to the tag we have to use the dot notation. It is clear that if we use an identifier for both the tag of a bus and one of its fields we could create ambiguous situations where the compiler would not know if we are referring to the value of the tag or the bus field. The symbol analysis is responsible for checking that no tag attached to a bus is called the same way as one of its fields to avoid this kind of situations.

3.4.2 Buses free of invalid statements

The next analysis we will be covering is the detection of invalid statements inside the body of a bus definition. Our idea of a bus is a set of signals (alone or forming arrays and other buses) grouped together for the purpose of representing a certain concept that can have properties attached to it. Buses like signals are immutable objects, so it does not make sense to let them have operations as classes in object-oriented languages do. Since we do not want (at least for the moment) to allow buses to include conditional declarations, we should not let programmers include statements different from declarations in the bodies of bus definitions.

Our first approach was to constrain the kind of statements allowed in the body of a bus definition through the parser. We created a new `BusBody` rule where the only possible statements to be used were declarations of signals, buses and arrays made up of elements from those two. Despite working, this idea had its drawbacks.

On one hand error messages would not have been very precise at pointing to the mistake causing the errors. On the other hand a lot of code from other parts of the compiler would have needed to be duplicated. In view of these problems we decided to use the same rule for the body of buses as for templates and functions, and detect any problem related to buses' bodies in a later stage. In fact, a similar situation happens with functions, where signal declarations are not allowed but the verification of this fact takes place at a later phase of compilation, with a static analysis.

Since the parser was now freed from any responsibility we had to implement a static analysis for detecting any forbidden statement inside a bus declaration. The analysis performs a tree traversal through the AST nodes of the tree with its root at the bus's body statement. In each node, the analysis will decide if the syntax represented by that node is allowed to be used inside a bus definition, and fail otherwise. The elements that will cause the analysis to fail are non-declaration statements, such as while loops, conditionals or constraints, declarations of input or output signals (it does makes no sense to declare signals inside buses as input or output since buses are not circuits) and expressions containing function calls where the function identifier is unknown or anonymous component calls.

3.4.3 Constant propagation

After the invalid statements analysis a constant propagation process is performed. During this process expressions in a bus definition will be examined to determine whether they maintain a constant value through the whole program. Those expressions found to be constant will be evaluated and their values will be broadcasted across the AST. This process has two objectives. On one side it allows for optimisations that will improve the performance of the final generated code. On the other side it facilitates the implementation of the unknown-known analysis that will be performed later.

3.4.4 Type analysis

We now enter the stage of the type analysis, perhaps the most important of all the analyses we will discuss. The goal of the type analysis is to check that all the statements of the program follow the rules imposed by the type system of the language. A type system is a set of rules that allow a type to be assigned to every syntactic construction in a language in order to later verify the correct formation of the constructions found in a program. Types restrict the operations that can be applied to an object. Not every syntactically correct construction is valid. In spite of the loss in flexibility, the use of types offers many advantages. To begin with, they enable early error detection, shortening the debugging phase and avoiding errors in production. They can also help the programmer to better convey his intentions and allow for deferring and encapsulating decisions about the representation of complex types, easing program maintenance and modification. In addition, a code fragment can be assigned many different types, which fosters code reuse and reduces costs of

production.

CIRCOM is a strongly and statically typed programming language that uses manifest typing. In CIRCOM, the programmer needs to indicate the type of each object (numerical variable, signal, signal array, component, ...). The type analysis we are discussing at this moment is used by the CIRCOM compiler to enforce the use of operations on the types they are designed for, like the use of arrows <== and <-- on signals, and executing the type checking in search of errors. This characteristics make the type system of CIRCOM a very restrictive one, but the security and efficiency obtained are worth the cost. Before the inclusion of buses in the language, CIRCOM did not provide support for the creation of custom types, but that changes with the addition of buses.

Ultimately buses work just like signals, so a programmer should be able to connect two buses of the same type directly, without having to worry about the connection of each of the signals that make up the whole bus. That is why we allow the constraint in line 17 of figure 3.4. In order to avoid type-related mistakes the compiler must check if the types of two connected buses are equal. We have decided to implement type equivalence by name, so two buses with different name types will be consider different even though they may display the same internal structure. As a result line 18 of figure 3.4 will produce a compilation error even though bus types `A` and `B` share the same internal structure. This type of equivalence is more restrictive, but is less prone to errors that can go undetected, and that is a key factor in a language designed for generating ZK proofs. It should be noted that two bus types will be considered equal even if they are instantiations of a certain bus definition with different values used as parameters. For example, if we define a bus type `bus BusT(N) {...}` with a parameter N to indicate the size of a signal array inside the bus and later declare `BusT(2) busA;` and `BusT(3) busB;`, an statement like `busA <== busB;` will not result in a type error because the compiler considers the types of those two buses to be the same.

The algorithm applied to perform the type checking traverses the AST bottom-up, inferring the type of a node from the types obtained for its children and the rules of the type system. The type inferred is attached to the node to be used when comparing the types of each expression looking for type mismatches. When all the children of a node have been typed, the algorithm will verify that the types of each expression are compatible and the global type structure makes sense. Therefore, if an expression like `list[i+j]` is found, the algorithm will first type the expressions `list` and `i+j`. Then it will verify that the type of `list` is in fact an array and the type of `i+j` is an arithmetic expression. Finally, it will attach the type of the objects inside the array `list` to the node representing `list[i+j]`. The algorithm will find that type looking at the declaration of `list`.

Buses generate a new problem when it comes to the type analysis: they force us to treat expressions with accesses to their fields. Accesses may be concatenated, so the process of typing them has to be recursive. What is worse, since the syntax

```
1  bus A {
2      signal s1;
3      signal s2[2];
4  }
5
6  bus B {
7      signal s1;
8      signal s2[2];
9  }
10
11 template Example() {
12     A input a1;
13     B output b;
14
15     A a2;
16
17     a2 <== a1;
18     b <== a2;
19 }
```

Figure 3.4: Example of type comparisons with buses.

to get to a certain field is the same as to get to the input and output signals of a component or to provide a tag with some value, we are required to distinguish between those cases based on the types of each identifier. To better understand the hurdles found when trying to check correct typing in buses, let us look at the example in figure 3.5. In template `Example()` we have three accesses with the dot syntax applied to the bus `out` of type `B`, but each of them serves a different purpose. The access in line 14 is used to assign a value to the *max_a* tag associated with `out`. In line 15 we access the signal field `b` of `out` and in line 16 we first access the bus field `busA` of `out` to later access the signal field `a` of `out.busA`, which is a bus of type `A`. As you can see, an expression using dot accesses over buses could have various meanings, and is the duty of the type analysis to determine which of them is the correct one.

When working with dot accesses in expressions, the type analysis algorithm first has to check that the variable type definition is that of a bus. After that, the algorithm is in charge of verifying the bus type includes a field by that name or has a tag named that way linked to it. It follows from this fact that there cannot be a field in a bus and a tag attached to it with the same identifier, but that situation was already prevented in the symbol analysis. If the identifier after the dot in the access expression is associated with a tag and no dot is found after the tag, the job is finished and the expression's type will be *tag*. If it is a signal, either the whole expression is a signal or another dot access is found after the signal's identifier, making it a *tag* type. Lastly, the identifier may belong to another bus variable, and the entire process has to be run again.

```
1  bus A {
2      signal a;
3  }
4
5  bus B {
6      signal b;
7      A busA;
8  }
9
10 template Example() {
11     signal input in;
12     B output {max_a} out;
13
14     out.max_a = 3;
15     out.b <== in;
16     out.busA.a <== in;
17 }
```

Figure 3.5: Example of dot accesses in buses with different meanings.

3.4.5 Unknown-known analysis

Leaving behind the type analysis, there is only one static analysis left we need to discuss: the unknown-known analysis. This static analysis is fairly particular to CIRCOM. As mentioned in section 2.5, circuits in real life are fixed structures whose composition cannot be altered by the values of its wires. No matter what values we use as inputs of an OR gate, it will remain an OR gate. It is a common mistake however to apply the same logic used for normal programming languages to hardware description languages. An inexperienced programmer may try to connect an output signal to one input signal of a circuit or another depending on the value of a signal used as a flag. The goal of the unknown-known analysis is to detect those kind of errors and prevent them from happening.

There are two main concerns that make it critical to detect these kind of mistakes. The first one is to keep the circuit's structure described by the R1CS of a program stable. If we allowed the structure of an arithmetic circuit to depend on the values of its signals, a program would no longer represent a certain circuit, but rather a group of them. The code would not be translated into an actual circuit until execution time. This is not the purpose of the CIRCOM language. The other problem is related to security. If we do not know what circuit a program represents until execution time, we cannot guarantee that a circuit generated that way will not have breaches that make it unsafe due to multiple witnesses that make the circuit satisfiable, and thus a deceitful prover could take advantage from it and fool the verifier. That is why a whole analysis is carried out to alert the programmer to such errors.

The unknown-known analysis works as follows. Each expression in the AST whose value cannot be known until the execution of the code is tagged as *Unknown*.

Such expressions include operations over signals, components or variables whose values depend on signals. At this point our previous constant analysis where we found out what expressions remain constant over the computation of the program is of great help. Those expressions whose values are known at compilation time are tagged as *Known*. The analysis traverses the AST downwards tagging all expressions. The goal is to find a constraint or signal declaration in a code block whose execution depends on an *Unknown* expression. As we have seen, we should not allow this to happen, and hence an error message will be thrown pointing to the part of the program which depends on an *Unknown* value. Array size declarations with an *Unknown* expression also cause the analysis to fail.

Since buses are treated like signals they and all their fields and tags must be labelled as *Unknown*. Another thing we have to pay attention to is bus types instantiations because the type of a bus signal should be known at compilation time in order to carry out type comparisons properly, so it cannot be dependent on *Unknown* arguments. And of course the expressions used in the bus definition to declare the sizes and arguments of each field have to be *Known*. Having verified all of these cases we can now be more confident about the security of the program.

During the next phases of the compilation process other things will be addressed such as tags management, constraints optimisations and code generation. All of these parts of the compiler need to undergo certain changes to allow the use of buses in CIRCOM programs. However, the modifications that need to be done are beyond the scope of this project and the work done by its author. Instead, during the next chapter we will focus on the implementation of buses in circuits from the CIRCOMLIB, the CIRCOM library, with the aim of enhancing safety and detection of errors for the next official version of the repository.

Chapter 4

CIRCOMLIB circuits

We will devote this section to present the changes made to the circuits of the CIRCOMLIB library with the objective of taking advantage of the benefits brought by the new bus tool. Since CIRCOMLIB is the official CIRCOM library, it is expected to grant a high level of reliability to its users. Buses can help achieve this purpose by eliminating some of the security flaws that programmers can introduce by accident in their codes associated with the supposition of preconditions on groups of signals that may not be met. In collaboration with tags, they can effectively be used to express relationships between elements of a circuit and therefore add an additional security check for detecting this kind of bugs. Moreover, buses provide a new means to represent complex ideas, the same way *structs* and *classes* allow the definition of new data structures from simpler data types. This can aid programmers at conveying ideas and all in all facilitating collaboration and code reuse.

Two major modifications have been carried out in the templates of CIRCOMLIB whose effects have spread through almost all of the files. One of the modifications has to do with the representation of numbers in binary form, and more precisely, the problems that conversions from numbers in $\mathbb{F}_p$ to numbers in binary form entail. A glimpse of the other major change was shown in section 1.1. Here we have defined two bus types for representing points in an elliptic curve. In combination with tags, this has allowed us to implement verifications of some properties regarding the coordinates of these points. Nonetheless, it is worth mentioning again that this implementation only adds a syntactic check, but no semantic test is performed, so that responsibility still belongs to the programmer.

4.1 Binary numbers

Let us commence by looking at the first modification mentioned. CIRCOM signals carry values from the finite field $\mathbb{F}_p$, where p is a prime number which is usually very large. This format can sometimes be inconvenient to perform certain operations using quadratic constraints, such as order comparisons. To overcome this issue, many CIRCOMLIB templates first transform the signals involved into their binary repre-

sentation, carry out the operation over the two numbers in binary format, and then undo the transformation. Therefore, a lot of circuits are implemented receiving as input an array of signals that is supposed to stand for the binary form of a number in A, where A can be $\mathbb{F}_p$ or a subset of $\mathbb{F}_p$. However, a problem arises from this solution: if one of the signals from the array carries a value other than 0 or 1, or if the number represented by the array of signals falls outside of A, preconditions are not met and the behaviour of the circuit may not be the expected one. To better understand the second case, imagine that $A = \mathbb{F}_5$. We will need 3 bits to represent all numbers from A in binary format. However, not every number whose binary representation employs 3 bits will be contained in A. For example, if we consider the number $(111)_2 = 7$, it does not belong to $\mathbb{F}_5$, so it is not supposed to be a valid input, but since $7 \equiv 2 \ mod \ 5$ maybe the operations applied to $(111)_2$ yield the same output as $(010)_2$. Having two different inputs that make a circuit satisfiable is the main issue in ZK proofs we want to avoid, as it can compromise the security of the whole protocol.

The problem with bit values outside $\{0, 1\}$ can be partially addressed with tags, included in CIRCOM from version 2.1.0 onwards. A *binary* tag can be added to the declaration of the input array of signals to ensure those signals have been associated with the *binary* tag by presumably implementing the necessary constraints to guarantee the values they carry are either 0 or 1. To tackle the other mentioned problem some templates like `CompConstant` and `AliasCheck` have been introduced. The `CompConstant` template must be instantiated with a number *ct* as parameter. It receives an array of input signals as the bits of some number n in $\mathbb{F}_p$ and returns an output signal that equals 1 if $n > ct$ in binary or 0 otherwise. `AliasCheck` takes advantage of this behaviour, and employs a component of `CompConstant` to check if a number received in binary form through an array of 254 input signals is smaller than $-1 \equiv p - 1 \ mod \ p$. With both circuits, we can insert the necessary constraints in our circuits to make sure we are not using any number outside our desired range.

Buses can help us improve the verifications performed by those two circuits. Imagine we had a manner of enveloping all the signals associated with the bits of a number in a certain data structure that we will call *BinaryNumber*. Then not only it would be far more clear what the idea behind those signals is, but we could modify `AliasCheck` to return a *BinaryNumber* signal *out* with a tag *unique* attached to it in order to inform other circuits manipulating *out* that the values carried by its bit signals have been checked to belong to a number in $\mathbb{F}_p$. That is exactly what buses allow us to do, as shown in figure 4.1.

The `BinaryNumber(n)` definition has been added to a different file of the CIRCOMLIB called *bitify.circom*, where most of the templates involving transformations of numbers in $\mathbb{F}_p$ to numbers in binary format and and vice versa can be found. Those templates have also been modified to receive or return `BinaryNumber` buses so as to keep coherence through all the repository.

```
template AliasCheck() {
    BinaryNumber(254) input in;
    BinaryNumber(254) output {unique} out;

    signal greater <== CompConstant(-1)(in);
    greater === 0;
    out <== in;
}
```

Figure 4.1: Template that checks if a number in binary format has an alias in $\mathbb{F}_p$.

4.2 Elliptic curves

It is time to focus on the modifications made to represent points of elliptic curves. One of the most important protocols provided by the CIRCOMLIB is public-private key encryption circuits. Public-private key encryption, also known as asymmetric cryptography, is a cryptographic system that uses pairs of keys: one public key that can be shared openly and one private key that is kept secret. In this system, anything encrypted with the public key can only be decrypted by the corresponding private key, and vice versa [35, 36]. This dual-key mechanism not only helps in encrypting messages but also in verifying the identity of the communicators, ensuring that the data remains secure from unauthorized access. Asymmetric cryptography is foundational in numerous security protocols, including those used for secure email transmissions, digital signatures, and secure web browsing.

In CIRCOM, those protocols are implemented using ECDSA (Elliptic Curve Digital Signature Algorithm). ECDSA is a variant of the DSA algorithm, but instead of functioning with exponentiations and discrete logarithms it employs operations over points in elliptic curves, which reduces the sizes of the numbers needed to guarantee the same security level as DSA or RSA [37]. The CIRCOM circuits work with the points of a particular elliptic curve defined in [26], known as the Baby-Jubjub curve. In order to understand how this elliptic curve is defined and used, we first need to review some definitions extracted from [25].

Definition 4.2.1 (Montgomery curve). Take a field $\mathbb{F}$ whose characteristic satisfies $char(\mathbb{F}) \neq 2$. Fix two parameters $A \in \mathbb{F} \setminus \{-2, 2\}$ and $B \in \mathbb{F} \setminus \{0\}$. The Montgomery curve with coefficients A and B over $\mathbb{F}$ is the elliptic curve

$$E_{M,A,B} : Bv^2 = u^3 + Au^2 + u$$

Definition 4.2.2 (Edwards curve). Take a field $\mathbb{F}$ whose characteristic satisfies $char(\mathbb{F}) \neq 2$. An Edwards curve over $\mathbb{F}$ is an elliptic curve

$$E : x^2 + y^2 = 1 + dx^2y^2$$

where $d \in \mathbb{F} \setminus \{0, 1\}$.

Definition 4.2.3 (Twisted Edwards curve). Take a field $\mathbb{F}$ whose characteristic satisfies $char(\mathbb{F}) \neq 2$. Fix distinct non-zero elements $a, d \in \mathbb{F}$. The twisted Edwards curve with coefficients a and d over $\mathbb{F}$ is the elliptic curve

$$E_{E,a,d} : ax^2 + y^2 = 1 + dx^2y^2$$

An Edwards curve is a twisted Edwards curve with $a = 1$.

Theorem 3.2. of **??** demonstrates that every twisted Edwards curve $E_{E,a,d}$ over a field $\mathbb{F}$ with $char(\mathbb{F}) \neq 2$ is birationally equivalent to a Montgomery curve $E_{M,A,B}$ over $\mathbb{F}$ via the map

$$(x, y) \mapsto (u, v) = \left(\frac{1+y}{1-y}, \frac{1+y}{(1-y)x}\right) \tag{4.1}$$

and its inverse

$$(u, v) \mapsto (x, y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right) \tag{4.2}$$

where $A = 2(a+d)/(a-d)$ and $B = 4/(a-d)$, or equivalently $a = (A+2)/B$ and $d = (A-2)/B$.

The Baby-Jubjub curve was discovered using a deterministic algorithm designed by a group of researchers of IRTF in 2016 [38]. Given a prime number p, the algorithm returns the Montgomery elliptic curve defined over $\mathbb{F}_p$ with the smallest coefficient A such that $A - 2$ is a multiple of 4. The requirement for $A - 2$ to be divisible by 4 is useful to simplify the operations involving points of the curve returned by the algorithm because these operations rely heavily on the value $A - 2$. The Baby-Jubjub elliptic curve was the first curve returned by the algorithm satisfying the criteria of SafeCurves, a project that performs some of the most common and known attacks on elliptic curves to verify their security level [39].

Definition 4.2.4 (Baby-Jubjub). Consider the field $\mathbb{F}_p$ where p is a huge prime number found in [26]. We define the **Baby-Jubjub** curve as the Montgomery elliptic curve over $\mathbb{F}_p$ given by the equation

$$BJ_M : v^2 = u^3 + 168698\,u^2 + u$$

Its equivalent form as a twisted Edwards curve over $\mathbb{F}_p$ is described by the equation

$$BJ_E : 168700\,x^2 + y^2 = 1 + 168696\,x^2y^2$$

The idea behind providing different equations to represent basically the same elliptic curve is not just mathematical curiosity. Each type of elliptic curve has its own laws to define the geometrical operations of adding two points or multiplying a point by an integer. As a result, some kinds of curves provide more efficient operations than others, so by changing from one representation to another we can improve the performance of the algorithms working with points of the curve.

CIRCOMLIB offers templates to apply these operations over points of the Baby-Jubjub curve either in twisted Edwards form or in Montgomery form. Templates to change from one form to the other are also found. However, we face the same problem as when we were working with the binary representation of numbers in $\mathbb{F}_p$: these templates operate on the assumption that their input signals represent the coordinates of a point of the Baby-Jubjub curve in the desired form, but there is no mechanism that guarantees that this precondition is met.

By using buses in the circuits dealing with elliptic curves points we will add another layer of verification to detect possible mistakes that can lead to security flaws. To illustrate how we will be doing this, we present the template provided in CIRCOMLIB to transform a point of the Baby-Jubjub curve in twisted Edwards form to its Montgomery form as it is in the current version of CIRCOMLIB:

```
1  template Edwards2Montgomery() {
2      signal input in[2];
3      signal output out[2];
4
5      out[0] <-- (1 + in[1]) / (1 - in[1]);
6      out[1] <-- out[0] / in[0];
7
8      out[0]  *  (1 - in[1]) === (1 + in[1]);
9      out[1]  *  in[0] === out[0];
10 }
```

Figure 4.2: Template to transform a point in twisted Edwards form to its Montgomery form.

As the reader can appreciate, the `Edwards2Montgomery` template receives two input signals standing for the x and y coordinates of a twisted Edwards curve point, encodes the equations given by the map defined in 4.1 and returns the u and v coordinates of the same point in the equivalent Montgomery curve as two output signals. The template has to declare separately the instructions to compute the witness of the arithmetic circuit and the constraints representing it because the operations carried out do not have a quadratic form. Nonetheless, both parts express the same equations moving terms properly.

Since the input of the circuit is a basic array of two signals, it is not difficult to wrongly wire them to other signals assuming they represent the coordinates of a point in twisted Edwards form but without implementing the necessary constraints to force that fact. To try to avoid these kind of situations we will replace the arrays of signals by a bus conveying the idea of a point in an elliptic curve. The bus will naturally have two signal fields `x` and `y`, one for each coordinate of the point.

Using this new bus type we can attach tags to the inputs and outputs of the template in 4.2 to characterise points in twisted Edwards form or in Montgomery form. We will assign the tag *babyedwards* to points of the twisted Edwards version

```
bus Point {
    signal x,y;
}
```

Figure 4.3: Point bus.

of the Baby-Jubjub elliptic curve, while we will use the tag *babymontgomery* for points of the Montgomery Baby-Jubjub curve. Therefore, the template will receive a `Point` bus with the tag *babyedwards* and will output a `Point` bus with the tag *babymontgomery*, which in theory should be equivalent to the point represented by the input.

```
template Edwards2Montgomery() {
    Point input {babyedwards} pin;
    Point output {babymontgomery} pout;

    pout.x <-- (1 + pin.y) / (1 - pin.y);
    pout.y <-- pout.x / pin.x;

    pout.x  *  (1 - pin.y) === (1 + pin.y);
    pout.y  *  pin.x === pout.x;
}
```

Figure 4.4: Template to transform a point in twisted Edwards form to its Montgomery form using buses.

With this new version of the circuit it is far more clear what the template is doing. Whoever employs the template will now automatically understand that the template receives a Baby-Jubjub point in twisted Edwards form and returns the same point in Montgomery form. Moreover, if the user forgets to add constraints to ensure the input bus is in fact a point of BJ_E and does not assign the tag *babyedwards* to the bus, the compiler will issue an error due to a tag mismatch between two wires connected. The only possible scenario where a bug can be introduced is when the programmer inserts the tag explicitly but does not implement the necessary constraints to force the fulfilment of preconditions or ensures that they are always met.

To show that the new version of the template guarantees that a point of the Montgomery Baby-Jubjub curve will always be returned if the precondition suggested by the *babyedwards* tag is fulfilled (i.e. the input point belongs to the twisted Edwards Baby-Jubjub curve) we will present a rigorous mathematical proof.

Proposition 4.2.1. *If the* `Point` *bus* `pin` *belongs to the elliptic curve* BJ_E *described by the equation*

$$168700\,(\texttt{pin.x})^2 + (\texttt{pin.y})^2 = 1 + 168696\,(\texttt{pin.x})^2(\texttt{pin.y})^2 \tag{4.3}$$

and is different from the point $(0, -1)$*, then the* `Point` *bus* `pout` *returned by the arithmetic circuit* `Edwards2Montgomery()` *will belong to the elliptic curve* BJ_M *defined by the equation*

$$(\texttt{pout.y})^2 = (\texttt{pout.x})^3 + 168698\,(\texttt{pout.x})^2 + \texttt{pout.x} \tag{4.4}$$

Proof. Let us consider the first of the constraints declared by `Edwards2Montgomery()`.

$$\texttt{pout.x} \times (1 - \texttt{pin.y}) = 1 + \texttt{pin.y} \Leftrightarrow \texttt{pout.x} - 1 = \texttt{pin.y} \times (\texttt{pout.x} + 1)$$

Then taking $a = 168700$, $d = 168696$ and $A = 168698$ we can reason as follows on the twisted Edwards equation:

$$\begin{aligned}
&a\,(\texttt{pin.x})^2 + (\texttt{pin.y})^2 = 1 + d\,(\texttt{pin.x})^2(\texttt{pin.y})^2 \Leftrightarrow \\
&\quad\Leftrightarrow a\,(\texttt{pin.x})^2(\texttt{pout.x}+1)^2 + (\texttt{pin.y})^2(\texttt{pout.x}+1)^2 = \\
&\qquad = (\texttt{pout.x}+1)^2 + d\,(\texttt{pin.x})^2(\texttt{pin.y})^2(\texttt{pout.x}+1)^2 \Leftrightarrow \\
&\quad\Leftrightarrow a\,(\texttt{pin.x})^2(\texttt{pout.x}+1)^2 + (\texttt{pout.x}-1)^2 = \\
&\qquad = (\texttt{pout.x}+1)^2 + d\,(\texttt{pin.x})^2(\texttt{pout.x}-1)^2 \Leftrightarrow \\
&\quad\Leftrightarrow a\,(\texttt{pin.x})^2(\texttt{pout.x})^2 + 2a\,(\texttt{pin.x})^2\texttt{pout.x} + a\,(\texttt{pin.x})^2 + \\
&\qquad + (\texttt{pout.x})^2 - 2\,\texttt{pout.x} + 1 = (\texttt{pout.x})^2 + 2\,\texttt{pout.x} + 1 + \\
&\qquad + d\,(\texttt{pin.x})^2(\texttt{pout.x})^2 - 2d\,(\texttt{pin.x})^2\texttt{pout.x} + d\,(\texttt{pin.x})^2 \Leftrightarrow \\
&\quad\Leftrightarrow (a-d)\,(\texttt{pin.x})^2(\texttt{pout.x})^2 + 2\,(a+d)\,(\texttt{pin.x})^2\texttt{pout.x} + \\
&\qquad + (a-d)\,(\texttt{pin.x})^2 = 4\,\texttt{pout.x}
\end{aligned}$$

Notice that $a - d = 4$ and $a + d = 2A$. Now we can use the equivalence $\texttt{pout.y} \times \texttt{pin.x} = \texttt{pout.x}$ given by the second constraint of the template to multiply both terms of the equation by $(\texttt{pout.y})^2$ and obtain the formula

$$\begin{aligned}
&4\,(\texttt{pout.x})^4 + 4A\,(\texttt{pout.x})^3 + 4\,(\texttt{pout.x})^2 = 4\,\texttt{pout.x}\,(\texttt{pout.y})^2 \Leftrightarrow \\
&\quad\Leftrightarrow (\texttt{pout.x})^4 + A\,(\texttt{pout.x})^3 + (\texttt{pout.x})^2 = \texttt{pout.x}\,(\texttt{pout.y})^2
\end{aligned}$$

We are very close to getting the desired Montgomery curve equation 4.4. The only step left is dividing by `pout.x`, but in order to do that we first have to verify that `pout.x` can not be zero. Let us suppose that $\texttt{pout.x} = 0$. Then the first constraint of `Edwards2Montgomery()` forces $0 = 1 + \texttt{pin.y}$, and thus $\texttt{pin.y} = -1$. Since the point `pin` fulfils the Baby-Jubjub twisted Edwards equation 4.3, the coordinate `pin.x` can only be 0. However, one of our premises was $\texttt{pin} \neq (0, -1)$, so with our initial assumptions it is not possible for `pout.x` to be 0, and that concludes the proof. □

It is worth mentioning that the exclusion of the point $(0, -1)$ from the possible template inputs is not arbitrary. This point is one of the two exceptions of the map 4.1 (the other one being $(0, 1)$) because it would require a division by 0. Its corresponding point of the curve BJ_M is $(0, 0)$ [25]. The problem of accepting $(0, -1)$ as input of `Edwards2Montgomery()` is that taking $\texttt{pout.x} = 0$ all constraints are satisfied since the nullity of `pin.x` cancels out the left term of the second constraint,

and that allows `pout.y` to take any possible value, but only for `pout.y` $= 0$ the point `pin` belongs to BJ_M. The discovery of this exception has led to an investigation with the goal of determining if there exists a situation where `pin` could take the values $(0, -1)$. If that is the case, we will have found a bug in the circuit that should be immediately fixed, as any circuit of the CIRCOMLIB must have only one possible output in order to be secure, and it should meet the post-conditions required. On the other hand, if no circuit of the CIRCOMLIB uses `Edwards2Montgomery()` with `pin.x` $= 0$ and `pin.y` $= -1$, the addition of a tag signalling the new precondition `pin` $\neq (0, -1)$ will be enough. We will not have problems with the point $(0, 1)$ as it is ruled out by the constraints of the template.

To address the possible bug without excluding the input $(0, -1)$, a modification of `Edwards2Montgomery()` is proposed in figure 4.5. Lines 11-13 force `pout.y` to be 0 if `pin.x` $= 0$ ($(0, -1)$ is the only point of the twisted Edwards Baby-Jubjub curve with zero as its first coordinate that fulfils the constraints of `Edwards2Montgomery()`). We have also changed the computation of the witness in lines 5-6 to return `pout.x` $=$ 0 and `pout.y` $= 0$ if the inputs verify `pin.x` $= 0$ and `pin.y` $= -1$. The result would be the same if we replaced the constraint in line 12 by `isz.in <== pin.y + 1`.

```
1  template Edwards2Montgomery() {
2      Point input {babyedwards} pin;
3      Point output {babymontgomery} pout;
4
5      pout.x <-- pin.y != -1 ? (1 + pin.y) / (1 - pin.y) : 0;
6      pout.y <-- pin.x != 0 ? pout.x / pin.x : 0;
7
8      pout.x  *  (1 - pin.y) === (1 + pin.y);
9      pout.y  *  pin.x === pout.x;
10
11     component isz = isZero();
12     isz.in <== pin.x;
13     isz.out  *  pout.y === 0;
14 }
```

Figure 4.5: Modification of `Edwards2Montgomery()` solving the problem with the input point $(0, -1)$.

We will now turn our attention to the circuit of CIRCOMLIB that performs the opposite operation. The template `Montgomery2Edwards()` receives a point of the Montgomery Baby-Jubjub curve and returns the equivalent point of the twisted Edwards Baby-Jubjub curve. Figure 4.6 displays the updated version of the template using buses to represent points. We will conduct a similar proof to the one for proposition 4.2.1 to verify that tags are correctly assigned to the output wires.

Proposition 4.2.2. *If the* `Point` *bus* `pin` *belongs to the elliptic curve* BJ_M *described by the equation*

$$(\texttt{pin.y})^2 = (\texttt{pin.x})^3 + 168698\,(\texttt{pin.x})^2 + \texttt{pin.x} \tag{4.5}$$

```
template Montgomery2Edwards() {
    Point input {babymontgomery} pin;
    Point output {babyedwards} pout;

    pout.x <-- pin.x / pin.y;
    pout.y <-- (pin.x - 1) / (pin.x + 1);

    pout.x  *  pin.y === pin.x;
    pout.y  *  (pin.x + 1) === pin.x - 1;
}
```

Figure 4.6: Template to transform a point in Montgomery form to its twisted Edwards form using buses.

and is different from the point $(0, 0)$*, then the* `Point` *bus* `pout` *returned by the arithmetic circuit* `Montgomery2Edwards()` *will belong to the elliptic curve* BJ_M *defined by the equation*

$$168700\,(\texttt{pout.x})^2 + (\texttt{pout.y})^2 = 1 + 168696\,(\texttt{pout.x})^2(\texttt{pout.y})^2 \tag{4.6}$$

Proof. Let us consider the first of the constraints declared by `Montgomery2Edwards()`.

$$\texttt{pout.x} \times \texttt{pin.y} = \texttt{pin.x}$$

If we take $A = 168698$ we can reason as follows on the Montgomery equation:

$$\begin{aligned}(\texttt{pin.y})^2 = (\texttt{pin.x})^3 + A\,(\texttt{pin.x})^2 + \texttt{pin.x} \Leftrightarrow \\ \Leftrightarrow (\texttt{pout.x})^2(\texttt{pin.y})^2 = (\texttt{pout.x})^2((\texttt{pin.x})^3 + A\,(\texttt{pin.x})^2 + \texttt{pin.x}) \Leftrightarrow \\ \Leftrightarrow (\texttt{pin.x})^2 = (\texttt{pout.x})^2((\texttt{pin.x})^3 + A\,(\texttt{pin.x})^2 + \texttt{pin.x})\end{aligned}$$

Now we have to multiply both terms of the equation by $(1 - \texttt{pout.y})^3$ and simplify the resulting expression using the second constraint of `Montgomery2Edwards()`:

$$\texttt{pout.y} \times (\texttt{pin.x} + 1) = \texttt{pin.x} - 1 \Leftrightarrow \texttt{pin.x} \times (1 - \texttt{pout.y}) = 1 + \texttt{pout.y}$$

$$\begin{aligned}(\texttt{pin.x})^2 = (\texttt{pout.x})^2((\texttt{pin.x})^3 + A\,(\texttt{pin.x})^2 + \texttt{pin.x}) \Leftrightarrow \\ \Leftrightarrow (\texttt{pin.x})^2(1 - \texttt{pout.y})^3 = (\texttt{pout.x})^2(1 - \texttt{pout.y})^3((\texttt{pin.x})^3 + \\ + A\,(\texttt{pin.x})^2 + \texttt{pin.x}) \Leftrightarrow \\ \Leftrightarrow (1 + \texttt{pout.y})^2(1 - \texttt{pout.y}) = (\texttt{pout.x})^2((1 + \texttt{pout.y})^3 + \\ + A\,(1 + \texttt{pout.y})^2(1 - \texttt{pout.y}) + (1 + \texttt{pout.y})\,(1 - \texttt{pout.y})^2) \Leftrightarrow \\ \Leftrightarrow (1 + \texttt{pout.y})\,(1 - (\texttt{pout.y})^2) = (\texttt{pout.x})^2(2 + A + \\ + (2 + A)\,\texttt{pout.y} + (2 - A)\,(\texttt{pout.y})^2 + (2 - A)\,(\texttt{pout.y})^3)\end{aligned}$$

Since $a = 168700 = A + 2$ and $d = 168696 = A - 2$ we obtain the next equivalence:

$$\begin{aligned}(1 + \texttt{pout.y})\,(1 - (\texttt{pout.y})^2) &= (\texttt{pout.x})^2(2 + A + \\ &\quad + (2 + A)\,\texttt{pout.y} + (2 - A)\,(\texttt{pout.y})^2 + (2 - A)\,(\texttt{pout.y})^3) \Leftrightarrow \\ \Leftrightarrow (1 + \texttt{pout.y})\,(1 - (\texttt{pout.y})^2) &= \\ &= (\texttt{pout.x})^2(1 + \texttt{pout.y})\,(a - d\,(\texttt{pout.y})^2) \Leftrightarrow \\ \Leftrightarrow (1 + \texttt{pout.y})\,(a\,(\texttt{pout.x})^2 + (\texttt{pout.y})^2) &= \\ &= (1 + \texttt{pout.y})\,(1 + d\,(\texttt{pout.x})^2(\texttt{pout.y})^2)\end{aligned}$$

If we could divide the whole equation by $1 + \texttt{pout.y}$ we would obtain the desired expression and conclude the proof. To do that we need to rule out the case when $1 + \texttt{pout.y} = 0$, or in other words, when $\texttt{pout.y} = -1$. So let us suppose that in fact $\texttt{pout.y} = -1$. Then the second constraint of `Montgomery2Edwards()` forces $\texttt{pin.x} = 0$. Since the point `pin` fulfils the Baby-Jubjub Montgomery equation 4.5, the coordinate `pin.y` can only be 0. However, one of our premises was $\texttt{pin} \neq (0, 0)$, so we have reached a contradiction. Therefore, $1 + \texttt{pout.y} \neq 0$ and the proof is complete. □

As expected, the point $(0, 0)$ of BJ_M, which is an exception of the map 4.2, causes the same problems as its equivalent point in twisted Edwards form $(0, -1)$. The whole discussion of the $(0, -1)$ point of BJ_E is also valid for this point, and the solution suggested in figure 4.7 is similar.

```
1  template Montgomery2Edwards() {
2      Point input {babymontgomery} pin;
3      Point output {babyedwards} pout;
4
5      pout.x <-- pin.y != 0 ? pin.x / pin.y : 0;
6      pout.y <-- (pin.x - 1) / (pin.x + 1);
7
8      pout.x  *  pin.y === pin.x;
9      pout.y  *  (pin.x + 1) === pin.x - 1;
10
11     component isz = isZero();
12     isz.in <== pin.x;
13     isz.out  *  pout.x === 0;
14 }
```

Figure 4.7: Modification of `Montgomery2Edwards()` solving the problem with the input point $(0, 0)$.

Chapter 5

Benchmarking

With all the work done until this point we have first provided the CIRCOM language with great new functionalities that improve clarity and reliability, and then applied those desirable traits to circuits of CIRCOMLIB. It is time to test whether the new features will have a negative impact on the sizes of the statements generated or if proof sizes will not be affected by the use of buses. For this purpose, we have selected a representative sample of the templates from CIRCOMLIB that have been modified to implement buses. We will run a series of tests employing different functionalities of the CIRCOM compiler to compare the results with the old version of each template.

The CIRCOM compiler is equipped with a set of tools that can be used to preform optimisations in the constraint systems described by a CIRCOM program. Specifically, it has three different command line options for choosing the desired level of optimisation. The first option that we will use is the one designated by the flag `--O0`. When selecting this option, the compiler applies no simplification whatsoever. The second option that will be applied corresponds to the flag `--O1`. This option tells the compiler to only simplify constraints of the kind `s1 === s2`, which are very common in CIRCOM circuits. Specially, when connecting the output of a component as an input of another component, the developer needs to introduce this kind of constraints to capture the binding. The situation is worsened when working with large circuits, which can entail hundreds of millions of extra constraints. The third option we will be working with is selected with the flag `--O2`. This option activates the compiler's full constraint simplification mode. Apart from the mentioned simplifications, with the `--O2` option the compiler will carry out a fixed-point algorithm to eliminate as many linear constraints as possible. Linear constraints correspond to equations where no product of signals is found, so every term of the equation is linear. As explained in [21], to reduce the amount of linear constraints describing a circuit the compiler divides the set of constraints into clusters of related linear constraints and then applies Gauss-Jordan elimination to each of the clusters. This process is iterated until it is no longer possible to optimise more linear constraints. The R1CS generated with the activation of this option will be the best one possible with the current version of the CIRCOM compiler.

We will divide the tests in two different groups: one for templates using the new `BinaryNumber` bus type (table 5.1) and the other for templates related to elliptic curves employing the `Point` bus type (table 5.2). For each of the circuits tested, we will apply the three levels of optimisation offered by the compiler and compare the results obtained by the old versions of the circuits not implementing buses and the updated versions using buses. We will examine three different metrics: the number of non-linear constraints of the final R1CS, the number of linear constraints and the total number of wires found in the arithmetic circuit.

Looking at the results obtained, a few conclusions can be drawn. First it can be noticed that buses do not introduce extra quadratic constraints, which is good news. In the simpler circuits, the number of constraints generated with or without buses is the same even before any kind of simplification is carried out. For the more complex templates, like `Num2Bits_strict` or `BabyPbk`, although without any simplification the implementation of buses increases the number of linear constraints, activating the first simplification option of the compiler eliminates those extra constraints. Remember that this option applies simplifications only over constraints like `s1 === s2`, but it seems that level of simplification is enough to obtain the same results as when not employing buses in the circuit. There are even cases where activating option `--O1` makes templates with buses generate systems with fewer constraints than those of the older versions. However, there are two exceptions to this rule, namely the templates `AliasCheck` and `BabyCheck`. The reason for this behaviour is the addition of outputs. While the old versions of the templates do not produce any outputs and just introduce constraints to force the signals fulfil certain properties, the newer versions output the input signals with tags linked to them so as to convey the semantics associated to those signals.

In order to reach stronger conclusions, more tests should be carried out with different templates. Nevertheless, the results shown in this section look very promising, as they suggest all the benefits that buses bring come with no drawbacks of bigger proof sizes.

CIRCUIT		NON-LINEAR		LINEAR		WIRES	
		Old	New	Old	New	Old	New
Num2Bits(254)	`--O0`	254	254	1	1	256	256
	`--O1`	254	254	1	1	256	256
	`--O2`	254	254	1	1	256	256
Num2Bits_strict()	`--O0`	516	516	769	**1024**	1284	**1539**
	`--O1`	515	515	3	3	518	518
	`--O2`	515	515	1	1	516	516
Bits2Num(254)	`--O0`	0	0	1	1	256	256
	`--O1`	0	0	1	1	256	256
	`--O2`	0	0	1	1	256	256
Bits2Num_strict()	`--O0`	262	262	769	**1024**	1284	**1539**
	`--O1`	261	261	3	3	518	518
	`--O2`	261	261	1	1	516	516
Num2BitsNeg(254)	`--O0`	256	256	2	2	259	259
	`--O1`	256	256	1	1	258	258
	`--O2`	256	256	0	0	257	257
BinSum(254,4)	`--O0`	252	252	1	1	1269	1269
	`--O1`	252	252	1	1	1269	1269
	`--O2`	252	252	1	1	1269	1269
BinSub(254)	`--O0`	255	255	1	1	764	764
	`--O1`	255	255	1	1	764	764
	`--O2`	255	255	0	0	763	763
AliasCheck()	`--O0`	262	262	259	**514**	774	**1029**
	`--O1`	261	261	2	**256**	517	**771**
	`--O2`	261	261	0	**254**	515	**769**

Table 5.1: Test results on templates using the `BinaryNumber` bus type.

CIRCUIT		NON-LINEAR		LINEAR		WIRES	
		Old	New	Old	New	Old	New
Edwards2Montgomery()	`--O0`	2	2	0	0	5	5
	`--O1`	2	2	0	0	5	5
	`--O2`	2	2	0	0	5	5
Montgomery2Edwards()	`--O0`	2	2	0	0	5	5
	`--O1`	2	2	0	0	5	5
	`--O2`	2	2	0	0	5	5
MontgomeryAdd()	`--O0`	3	3	0	0	8	8
	`--O1`	3	3	0	0	8	8
	`--O2`	3	3	0	0	8	8
MontgomeryDouble()	`--O0`	4	4	0	0	7	7
	`--O1`	4	4	0	0	7	7
	`--O2`	4	4	0	0	7	7
BabyAdd()	`--O0`	6	6	0	0	11	11
	`--O1`	6	6	0	0	11	11
	`--O2`	6	6	0	0	11	11
BabyCheck()	`--O0`	3	3	0	**2**	5	**7**
	`--O1`	3	3	0	**2**	5	**7**
	`--O2`	3	3	0	**2**	5	**7**
BabyPbk()	`--O0`	3948	3948	6170	**6173**	10119	**10122**
	`--O1`	3939	3939	**182**	180	**4122**	4120
	`--O2`	776	776	0	0	777	777
EscalarMulFix(254,B)	`--O0`	3695	3695	5909	**5912**	9859	**9862**
	`--O1`	3691	3691	**176**	174	**4122**	4120
	`--O2`	524	524	0	0	779	779

Table 5.2: Test results on templates using the `Point` bus type.

Chapter 6

Conclusions and Future Work

In this final chapter, we will present the conclusions drawn from this project. We will summarise the work completed and the objectives that have been achieved. Lastly, we will suggest potential avenues for further development that may stem from this project.

6.1 Conclusions

The initial phase of the project involved a thorough examination of an unfamiliar code base. This process was challenging due to the complexity typically associated with compiler code and the initial unfamiliarity with the programming language used. Effective strategies to navigate this included basic language tutorials followed by example-driven learning and active communication with the original developers.

An important element of this project was learning to program in Rust, which can be challenging due to its unique memory safety features and pattern matching capabilities. Despite the initial difficulties, mastering Rust proved beneficial, particularly due to the compiler's helpful debugging hints and the language's robust safety features, which ensure code reliability by preventing side effects and data races. The flexibility in creating structures, such as those used for AST components, and Rust's growing popularity suggest that knowledge of this language will be advantageous for future endeavours.

The primary goal of this project was the development of a tool in the CIRCOM language that could assist programmers in specifying properties of groups of signals. To that end, the characteristics of CIRCOM were explored, leading to the conception of signal buses. Buses allow for the application of existing tags to groups of signals, thereby helping clarify the semantics associated to these signals. Furthermore, the introduction of buses has enriched the CIRCOM language by providing the capability for defining custom signal types, thus enhancing the language's expressiveness and improving the clarity of code within circuit designs.

The implementation of buses required significant modifications to various components of the CIRCOM compiler, alongside the development of analyses to identify potential errors in their use. Each alteration was meticulously planned and executed to ensure optimal integration. Comprehensive testing was conducted to ensure the correct functionality of the updates and to address any undetected bugs. Finally, the compiler was updated to accurately interpret the bus syntax, fulfilling **objectives I) and II)** of section 1.2. To ascertain whether the use of buses affected the size of the generated Zero-Knowledge proofs, benchmark tests were conducted against code that did not use buses. These tests confirmed that, thanks to the compiler's optimisation features, the inclusion of buses did not impact the proof sizes at all.

With the purpose of taking advantage of the benefits brought by buses, many circuits within the official CIRCOM library (CIRCOMLIB) were revised in preparation for the release of its new version. These adjustments not only have enhanced the clarity of the circuits but also significantly strengthen the reliability of the library's templates thanks to enabling the attachment of tags to groups of signals expressing properties that have been enforced by the circuits of the CIRCOMLIB. Hence, we have managed to achieve **objective III)**. To make sure no property has been wrongly assigned, rigorous mathematical proofs have been employed to verify the correctness of these properties as stated in **objective IV)**. All in all, the security level of the templates provided by the CIRCOMLIB has been notably increased.

6.2 Future lines of work

Although the work conducted has been exhaustive, there remain several aspects that we have not had time to explore, which could serve as valuable directions for future research:

- The development of tools for automatically verifying the correctness of circuits that utilise the new tags and buses added to CIRCOMLIB. During the development of the project the idea came up to create a software that would verify if the templates of the CIRCOMLIB are used correctly in a program. This software would take advantage of the new tags implemented in the templates of the CIRCOMLIB and would check if no tags are added manually to a signal or bus, but rather obtained by means of another CIRCOMLIB circuit. When no possible problematic points were found the programmer would receive a message confirming the program is (almost) guaranteed to be safe.

- The extension of CIRCOM to admit conditional definitions on buses dependent on the parameters provided. We have explained in detail why buses are not like object-oriented languages' classes and the reasons to forbid statements different from declarations inside buses. However, it could make sense to allow the inclusion of conditional statements inside bus definitions with the purpose of providing more flexibility to bus types and even enabling some recursive implementations.

- The implementation of new statements to connect bus fields to a set of signals with a single instruction. Looking at other general-purpose programming languages, many of them feature instructions to initialise the fields of a *struct* or an equivalent object with a single assignment of the form `var a = b, c, d`. CIRCOM could add a similar statement to its repertoire as a less verbose manner of declaring linear constraints involving a bus.

Bibliography

[1] Ronald L Rivest. Cryptography. In *Algorithms and complexity*, pages 717–755. Elsevier, 1990.

[2] Shafi oldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *17th Annual ACM Symposium on Theory of Computing*, 1985.

[3] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[4] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[5] Vincent Gramoli, Len Bass, Alan Fekete, and Daniel W Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2711–2724, 2015.

[6] EthHub. Zk-rollups. Available at https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/.

[7] Polygon. Polygon zk-evm. Available at https://zkevm.polygon.technology/.

[8] Scroll. Scroll zk-evm. Available at https://scroll.io/blog/architecture.

[9] Roger Maull, Phil Godsiff, Catherine Mulligan, Alan Brown, and Beth Kewell. Distributed ledger technology: Applications and implications. *Strategic Change*, 26(5):481–489, 2017.

[10] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112, 1988.

[11] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks. *arXiv preprint arXiv:2202.06877*, 2022.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[13] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[14] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019.

[15] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2): 103–112, 2016.

[16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 327–357. Springer, 2016.

[17] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, pages 1–45, 2020.

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

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

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

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

[22] iden3. Circom documentation, . Available at https://docs.circom.io/.

[23] iden3. Circomlib repository, . Available at https://github.com/iden3/circomlib/.

[24] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Scalable verification of zero-knowledge protocols. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 133–133. IEEE Computer Society, 2024.

[25] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *Progress in Cryptology–AFRICACRYPT 2008: First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings 1*, pages 389–405. Springer, 2008.

[26] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve. *Ethereum Improvement Proposal, EIP-2494*, 29, 2020.

[27] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to explain zero-knowledge protocols to your children. In *Conference on the Theory and Application of Cryptology*, pages 628–631. Springer, 1989.

[28] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 626–645. Springer, 2013.

[29] iden3. Circom repository, . Available at https://github.com/iden3/circom.

[30] Michael Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

[31] Hermenegildo García Navarro. Design and implementation of the circom 1.0 compiler. 2020.

[32] iden3. Old circom repository, . Available at https://github.com/iden3/circom_old.

[33] Carol Nichols Steve Klabnik. The rust programming language. Available at https://doc.rust-lang.org/stable/book/.

[34] Rust Foundation. Rust by example. Available at https://doc.rust-lang.org/rust-by-example/.

[35] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[36] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549 (7671):188–194, 2017.

[37] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm fecdsah 1 2. 1999.

[38] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. Technical report, 2016.

[39] Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography daniel j. bernstein university of illinois at chicago &.