Doctoral Thesis

A comprehensive study of arithmetic circuits and elliptic curves for efficient and scalable zero-knowledge proof systems

Marta Bellés Muñoz

June 2023

Dr. Vanesa Daza Fernández
Pompeu Fabra University
Supervisor

Dr. Jose Luis Muñoz Tapia
Polytechnic University of Catalonia
Co-supervisor

Department of Information and Communication Technologies

Trust, but verify

Russian proverb

Agraïments

La tesi que teniu a les vostres mans no hagués estat possible sense el suport i l'ajuda de companys, col·legues, amics i família.

M'agradaria agrair l'excepcional atenció i eficàcia del personal administratiu de Tànger, en especial la feina de la Lydia, que ha facilitat tots els processos administratius que comporta un doctorat.

Part important d'aquesta tesi va començar durant el meu pas per iden3. Vull agrair al Jordi l'oportunitat d'aprendre des de zero i per ser una font constant d'inspiració. La meva visió sobre els circuits te la dec a tu. També vull agrair a l'Arnau i l'Edu el seu entusiasme contagiós pel software lliure i el dret a la privacitat, i per trobar sempre un moment per donar-me un cop de mà.

Gràcies als companys de la Pompeu per compartir inquietuds acadèmiques i fer-me un lloc a les vostres converses, sempre interessants i plenes d'idees. Gràcies en especial a l'Alex, l'Arantxa, el Fede, el Rasoul, el Sergi i la Zaira.

L'última contribució d'aquesta tesi no hagués estat possible sense la família de Dusk. Gràcies a tot l'equip, en especial gràcies a l'Ema, la Jeske i el Matteo per l'oportunitat que m'heu donat i per la confiança dipositada. Sí puc dir que aquesta tesi no seria la mateixa sense el Xavi i el Javi. Us estic profundament agraïda per haver fet equip amb mi i per ensenyar-me tantes coses que no sabia.

Gràcies a la Vanesa per animar-me a començar el doctorat i per acollir-me al departament. Més enllà d'haver-me ajudat professionalment, em quedo amb les converses, els consells, la força i el suport que m'has donat durant aquests anys.

No puc posar en paraules tot el meu agraïment cap al Jose. No seria on sóc sense la teva inestimable ajuda, orientació i suport. Estaré sempre en deute en tu per obrir-me les portes a un món que he fet casa meva.

Agraeixo a les nenes, companys de carrera, de conservatori i veïnes, la paciència i comprensió que heu tingut amb mi. Gràcies per escoltar-me, fer-me tocar de peus a terra quan ho necessito i ser sempre allà.

Família, papa, mama, Albert i Agnès, sempre esteu disposats a ajudar en el que calgui, escoltar, donar consells i ajudar. Què puc dir, gràcies.

Per últim, gràcies a tu Miquel. Em sento molt afortunada d'haver pogut viure aquesta experiència al teu costat. Gràcies per escoltar-me, per compartir amb mi el que penses i per tenir sempre un punt de vista diferent.

Abstract

In recent years, zero-knowledge proofs have come to play a crucial role in distributed systems where there is no trust between the parties involved. Most popular proof systems are for the NP-complete language of arithmetic circuit satisfiability. Although there have been tremendous efforts in understanding, developing, and improving zero-knowledge proof systems, not much work has been done towards the study of arithmetic circuits. In this thesis, we contribute to this matter in three different aspects.

First, we present circom, a programming language for writing arithmetic circuits that abstracts the complexity of the proof system. Second, we provide a deterministic algorithm for generating twisted Edwards elliptic curves that can be used to prove elliptic-curve cryptography statements in zero knowledge efficiently. Finally, we explore recursive composition of pairing-based proof systems with native circuit arithmetic, delving into the study of cycles of pairing-friendly elliptic curves of prime order.

Resum

En els últims anys, les proves de coneixement zero han passat a tenir un paper crucial en el sistemes distribuïts on no hi ha confiança entre els participants. Els sistemes de prova més populars són pel llenguatge NP complet de satisfacibilitat de circuits aritmètics. Tot i que hi ha hagut molts esforços per entendre i millorar les proves de coneixement zero, no s'ha avançat tant en l'estudi dels circuits aritmètics. En aquesta tesi, contribuïm a aquest tema en tres aspectes.

Primerament, presentem CIRCOM, un llenguatge de programació per escriure circuits aritmètics que abstreu la complexitat del sistema de prova. Segonament, proporcionem un algorisme determinista per a generar corbes el·líptiques que permeten demostrar eficientment declaracions de criptografia de corba el·líptica. Finalment, explorem la composició recursiva de sistemes de prova basats en aparellaments utilitzant l'aritmètica nativa dels circuits, aprofundint en l'estudi de cicles de corbes el·líptiques d'ordre primer amb aparellaments adients.

Contents

List of figures

List of tables

Chapter 1

Introduction

> *A proof is a proof. What kind of a proof? It's a proof. A proof is a proof. And when you have a good proof, it's because it's proven.*
>
> – Jean Chretien

Computers and algorithms play an essential role in today's modern society. From smartphones and laptops, to medical devices and transportation systems, new technologies have transformed the way we shape our society. With the rapid growth in computing power, data availability, and recent advances in the field of artificial intelligence, it seems natural to think that a computer with enough memory and time can solve any problem, regardless of its complexity. Yet, in 1936 Alan Turing proved the existence of a problem that even the most powerful modern computer cannot solve.

We say that a problem is *solvable* if there exists and *algorithm*, that is, a finite sequence of instructions that, once it is executed on a machine, leads to the result of the problem [BC94]. Depending on the capabilities of the machine they are executed on, a problem may be solvable or not. In his seminal paper [Tur36], Turing introduced a set of theoretical machines that manipulated symbols on an infinite tape according to a table of rules. Although these machines, later known as *Turing machines*, were simple, they turned out to be a very powerful computation model: anything that a real computer can solve can also be solved by a Turing machine [HU79]. Turing showed that no Turing machine, and hence,

no modern computer, could solve *the halting problem*. This result is one of the most philosophically important theorems of the theory of computation, because it proves that computers are limited in a fundamental way [Sip13].

For those problems that *are* solvable over a Turing machine, it is interesting to study the complexity of finding a solution based on the computational resources needed to solve them. The fundamental complexity classes P and NP are based on the time needed to solve a problem. The class P contains those problems that can be solved by a *deterministic* Turing machine using a polynomial amount of computation time, while NP contains those that can be solved by a *non-deterministic* Turing machine in polynomial time. In general, problems in P are considered *tractable*, while many problems in NP are regarded as not realistically solvable on a computer [BC94, Sch96]. Clearly, the class NP includes P, but whether P = NP is one of most important questions in computer science and mathematics.

One important advance on this question was the discovery of certain problems in NP that were at least as difficult as any other problem in the class [Sch96]. More precisely, if a polynomial time algorithm exists for solving any of these problems, *all* problems in NP would also be in P. These problems are known as *NP-complete problems*. In this thesis we focus on an NP-complete problem called *arithmetic circuit satisfiability*. An arithmetic circuit consists of wires (also called *signals*) that carry values from a prime finite field $\mathbb{F}_p$. These wires are connected to gates representing additions and multiplications modulo p. The arithmetic circuit satisfiability problem asks if, given a circuit, there exists a valid assignment of the signals that make the circuit satisfiable.

Another characterization of the NP complexity class is as the set of problems for which solutions can be verified in polynomial time in a deterministic Turing machine [AB07]. In other words, we can think of this class as the set of problems such that checking the validity of a potential solution can be done efficiently, but finding the solution may require a more extensive search or computation. To make this definition precise, we associate to each class of problems in NP a *language* that identifies statements that are true. For example, the arithmetic circuit satisfiability problem can be captured by the language

$$CSAT = \{\ C \text{ arithmetic circuit } \mid \text{“}C \text{ is satisfiable” is true } \}.$$

Now, the arithmetic circuit satisfiability problem is the decision problem of determining whether $C \in CSAT$ or not.

Formally, we define NP as exactly the class of languages $\mathcal{L}$ for which there exists a deterministic polynomial-time verification algorithm V such that $x \in \mathcal{L}$

if and only if there exists a *witness* w such that $V(x, w) = \texttt{accept}$. This may be viewed as a very simple *(classical) proof system* between a prover P with unbounded power or in possession of w, and a verifier V as described above. The prover P can always try to prove to V that $x \in \mathcal{L}$ by sending w to V. If a valid w exists, P is always able to make the verifier accept (*completeness*), and if no such w exists, then there is no way in which P can trick V to accept (*soundness*). This system captures the nature of a mathematical proof.

In the 1980s, some works [BS84, GMR85] proposed randomized and interactive verification procedures that lead to *(probabilistic) proof systems* with a relaxed version of the soundness property with a small but controllable *soundness error*. In this setting, Goldwasser, Micali, and Rackoff [GMR85] proved that it was possible to have proofs that efficiently demonstrate membership in a language without conveying any additional knowledge. That is, *zero-knowledge* (ZK) proofs that yield nothing beyond the validity of the assertion being proved. The *non-interactive* version of ZK proof systems (NIZK) was introduced shortly after in [BFM88]. The subsequent works [Dam92, FLS99, KP98] proved the existence of NIZK arguments for all NP languages. $CSAT$ is of particular interest, because circuits encode many types of computation in a natural way.

For example, in the blockchain space, Zcash [HBHW19] uses NIZKs for $CSAT$ to process *confidential transactions*. In a transparent network, anyone can check if a transaction satisfies the conditions of a valid transaction (e.g. the sender does not spend more than they have). In the confidential case, addresses and values are hidden. In order to prove the correctness of these transactions, they are accompanied by a NIZK proof that proves that the conditions for a valid transaction are met. By verifying the NIZK proof, the network can check if the transaction is valid without learning any details.

The Ethereum network uses highly expressive *smart contracts* to enable complex transactions. Smart contracts are public and provide no inherent privacy [BBB+17]. Moreover, the interaction with an smart contract is expensive, and the smart contract's own computational power is highly limited. To bring privacy to smart contracts, it is not sufficient to use a NIZK, we need a NIZK with small proof size and a fast verification algorithm. These type of proofs are called *ZK succinct non-interactive arguments of knowledge* (ZK-SNARKs).

In general, ZK-SNARKs for $CSAT$ require arithmetic circuits to be translated into a more mathematical description [GGPR13]. A common encoding is as a set of quadratic constraints called *rank-1 constraint system* (R1CS). In practice, ZK-SNARK proof shows, without revealing secret values, that the prover knows an assignment to all wires of the circuit that fulfill all constraints of the R1CS. In the first part of this thesis, we present CIRCOM, a programming

language for writing arithmetic circuits. Unlike other software, CIRCOM gives programmers total control about the signals and the constraints of the R1CS that define the computation.

Most efficient ZK-SNARK constructions [PHGR13, Gro16, GWC19] make use of bilinear maps (*pairings*) on elliptic-curve groups for verification of proofs, achieving verification time that does not depend on the size of circuit associated to the statement being proven. More precisely, a pairing-based SNARK relies on an elliptic curve $E/\mathbb{F}_q$ for some prime q such that the group $E(\mathbb{F}_q)$ has a large subgroup of prime order p. With this setting, the SNARK is able to prove satisfiability of arithmetic circuits over $\mathbb{F}_p$. Since *elliptic-curve cryptography* (ECC) works in large prime fields, elliptic curves come as the natural representation of circuits. The second part of this thesis is driven by the search of curves $E'/\mathbb{F}_p$ that allow to describe ECC statements. More precisely, we look for twisted Edwards curves that allow an efficient implementation of ECC schemes such as the Bowe–Hopwood–Pedersen hash or the Edwards digital signature algorithm (EdDSA). We propose a deterministic algorithm for generating such curves and give a concrete curve $E'/\mathbb{F}_p$ with p being the order of BN-256.

In the last part of the thesis, motivated by recursive composition of pairing-based SNARKs, we study 2-cycles of pairing-friendly elliptic curves. We show that families of elliptic curves parameterized by low-degree polynomials, which is the only known approach at generating pairing-friendly elliptic curves of prime order, are unlikely to yield new 2-cycles. In particular, we show that such cycles do not exist unless a strong condition holds.

1.1 Contributions and organization

The thesis is structured as follows. In Chapter 2, we give the context and background to understand the subsequent chapters. The following Chapters 3-5 result from three lines of research. Each chapter corresponds to one paper, some already published, some in preprint stage. The contents are essentially the same as in the papers, with only minor modifications to remove redundancies, unify notation, and ensure a more cohesive document.

- Chapter 3: we present CIRCOM, a novel circuit description language that programmers can use to implement arithmetic circuits describing complex computations. We provide a detailed description and analysis of the language, including its syntax, semantics, and expressive power. This chapter is based on the articles [BBDM22] and [BMIMT+22].

- Chapter 4: we present our work on the deterministic generation of suitable twisted Edwards curves for $\mathbb{F}_p$-arithmetic circuits for a given prime p. The contents of this chapter are the result of the common efforts from the ZK community to standardize these procedures. The chapter is based on the article [BWB+21].

- Chapter 5: we explore recursive composition of pairing-based SNARKs with cycles of pairing-friendly elliptic curves of prime order. The content of this chapter is based on [BMUS22] that has been accepted to *Crypto'23*.

In Chapter 6, we close with some general conclusions. The thesis also includes an Appendix A with code from Chapters 4 and 5, and an Appendix B with the list of publications and their abstracts.

Chapter 2

Preliminaries

> *What is intuitively required from a theorem-proving procedure? First, that it is possible to "prove" a true theorem. Second, that it is impossible to "prove" a false theorem. Third, that communicating the proof should be efficient, in the following sense. It does not matter how long must the prover compute during the proving process, but it is essential that the computation required from the verifier is easy.*
>
> – Shafi Goldwasser, Silvio Micali, and Charles Rackoff

2.1 Zero-knowledge proofs

In 1985, Goldwasser, Micali, and Rackoff [GMR85] introduced *zero-knowledge* (ZK) *proofs* that allow one party, called *prover*, to convince another one, called *verifier*, that a *statement* is true without revealing any information beyond the veracity of the statement. In this context, a statement is usually associated to an *instance*, a public input known to both prover and verifier, and a *witness*, a private input known only by the prover. Informally, a ZK proof should satisfy three properties:

- *Completeness*: given a statement and a witness, the prover can convince the verifier that the statement is true.
- *Soundness*: a malicious prover cannot convince the verifier of a false statement.
- *Zero-knowledge*: the proof does not reveal anything else but the truth of the statement, in particular, it does not reveal the prover's witness.

Note that the first two properties protect the verifier against dishonest provers, while the *zero-knowledge* property guarantees the prover's privacy against malicious verifiers.

In this work we focus on ZK-SNARKs [PHGR13,Gro10,Gro16], which belong to group of ZK proofs known as *arguments of knowledge*. Informally, while a *proof* proves that there exists a valid witness, an argument of knowledge proves that, with very high probability, the prover does *know* a concrete valid witness. An argument of knowledge is considered a SNARK if it is non-interactive and, regardless of the size of the statement being proved, has succinct proof size (e.g. [Gro16]-proofs are ≈ 200 bytes). Many ZK-SNARKs use pairing functions over elliptic curve groups to also guarantee short verification time [Gro16, GWC19]. We give a formal definition in the following section.

The main downside ZK-SNARKs for general statements is that they are not possible without using a common reference string (CRS), which should be known by both the prover and the verifier. Essentially, a CRS is constructed from a set of random values (also called *toxic waste*) that should not be known by the prover nor the verifier. Security proofs assume that the CRS was honestly generated. In practice, the CRS can be generated by a trusted third party, or using a secure multi-party computation (MPC) protocol to construct the CRS. MPC allows multiple independent parties to collaboratively construct the parameters in away that it is enough that one single participant deletes its secret counterpart of the contribution to keep the whole scheme secure [Can01]. The parameters involved in this initial phase are also known as *trusted setup*.

2.1.1 ZK-SNARKs

Let $\mathcal{R}$ be a relation generator that given a security parameter λ returns a polynomial time decidable binary relation R. For pairs $(\phi, w) \in R$, we call ϕ the *statement* and w the *witness*. We define $\mathcal{R}_\lambda$ to be set of possible relations R that $\mathcal{R}$ may output given 1^λ. The relation generator may also output some side information, an auxiliary input z, which will be given to the adversary.

An *efficient prover publicly verifiable non-interactive argument for* $\mathcal{R}$ is a quadruple of probabilistic polynomial algorithms (Setup, Prove, Vfy, Sim) such that

- $(\sigma, \tau) \leftarrow \mathsf{Setup}(R)$: the setup produces a common reference string σ and a simulation trapdoor τ for the relation R.
- $\pi \leftarrow \mathsf{Prove}(R, \sigma, \phi, w)$: the prover algorithm takes as input a common reference string σ and $(\phi, w) \in R$ and returns an argument π.
- $0/1 \leftarrow \mathsf{Vfy}(R, \sigma, \phi, \pi)$: the verification algorithm takes as input a common reference string σ, a statement ϕ, and an argument π, and returns 0 (reject) or 1 (accept).
- $\pi \leftarrow \mathsf{Sim}(R, \tau, \phi)$: the simulator takes as input a simulation trapdoor and statement ϕ and returns an argument π.

Definition 2.1 (Non-interactive zero-knowledge argument of knowledge). We say that (Setup, Prove, Vfy, Sim) is a *non-interactive zero-knowledge argument of knowledge for* $\mathcal{R}$ if it has perfect completeness, perfect zero-knowledge, and computational knowledge soundness as defined below.

Completeness says that, given any true statement, an honest prover should be able to convince an honest verifier.

Definition 2.2 (Perfect completeness). We say that (Setup, Prove, Vfy) has *perfect completeness* if, for all $\lambda \in \mathbb{N}, R \in \mathcal{R}_\lambda$, and $(\phi, w) \in R$,

$$\Pr\left[(\sigma, \tau) \leftarrow \mathsf{Setup}(R); \pi \leftarrow \mathsf{Prove}(R, \sigma, \phi, w) \mid \mathsf{Vfy}(R, \sigma, \phi, \pi) = 1\right] = 1.$$

An argument is zero-knowledge if it does not leak any information besides the truth of the statement.

Definition 2.3 (Perfect zero-knowledge). We say that (Setup, Prove, Vfy, Sim) is *perfect zero-knowledge* if, for all $\lambda \in \mathbb{N}, (R, z) \leftarrow\in \mathcal{R}(1^\lambda), (\phi, w) \in R$ and all adversaries $\mathcal{A}$,

$$\begin{aligned}&\Pr\left[(\sigma, \tau) \leftarrow \mathsf{Setup}(R); \pi \leftarrow \mathsf{Prove}(R, \sigma, \phi, w) \mid \mathcal{A}(R, z, \sigma, \tau, \pi) = 1\right] \\ &= \Pr\left[(\sigma, \tau) \leftarrow \mathsf{Setup}(R); \pi \leftarrow \mathsf{Sim}(R, \sigma, \phi) \mid \mathcal{A}(R, z, \sigma, \tau, \pi) = 1\right].\end{aligned}$$

Soundness examines the probability of proving a false statement, that is, of convincing a verifier if no witness exists.

Definition 2.4 (Computational knowledge soundness). We call (Setup, Prove, Vfy, Sim) an *argument of knowledge* if there is an extractor that can compute a witness whenever the adversary produces a valid argument. The extractor gets full access to the adversary's state, including any random coins. Formally, we require that for all non-uniform polynomial time adversaries $\mathcal{A}$ there exists a non-uniform polynomial time extractor $\chi_{\mathcal{A}}$ such that

$$\Pr\left[\begin{array}{r}(R,z) \leftarrow R(1^\lambda); (\sigma,\tau) \leftarrow \mathsf{Setup}(R); ((\phi,\pi);w) \leftarrow (\mathcal{A} \,||\, \chi_{\mathcal{A}})(R,z,\sigma) \\ \mid (\phi,w) \notin R \text{ and } \mathsf{Vfy}(R,\sigma,\phi,\pi) = 1\end{array}\right] \approx 0.$$

Finally, we say that a non-interactive zero-knowledge argument of knowledge is *succinct* (ZK-SNARK), if the verifier runs in polynomial time in $\lambda + |\phi|$ and the proof size is polynomial in λ.

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

2.1.2 Arithmetic-circuit satisfiability

The most widely studied language in the context of ZK-SNARK proofs is the NP-complete language of *circuit satisfiability* [BCC+16, PHGR13, BSCTV14b]. Essentially, a *circuit* consists of a set of wires connected to gates that perform some operation. Circuit satisfiability is a classical problem of computability theory that consists of determining whether a given circuit has an assignment of its inputs that makes the output true. If that is the case, the circuit is called *satisfiable*. Otherwise, the circuit is called *unsatisfiable*.

In cryptographic implementations of this problem, we use a particular type of circuits called *arithmetic circuits* (also called *circuits*, *ZK circuits*, or *ZK-SNARK circuits*). The gates of an arithmetic circuit consist on additions and multiplications modulo p, where p is typically a large prime number of approximately 254 bits [WBB20]. The wires of an arithmetic circuit, often called *signals*, can carry any value from the prime finite field $\mathbb{F}_p$. As with electronic circuits, we can distinguish between *input*, *intermediate*, and *output signals*.

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

Example 2.1. Circuit C from Figure 2.1 is an arithmetic circuit defined over the prime finite field $\mathbb{F}_{11}$ that, given four private inputs s_1, s_2, s_3, s_4, it outputs the result of the operation

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

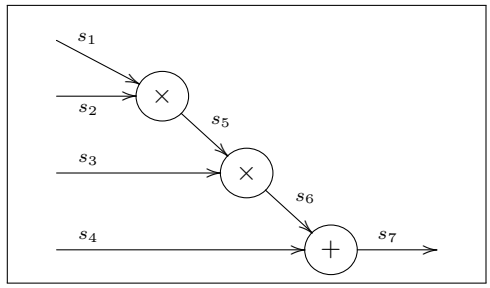

Figure 2.1: Representation of an arithmetic circuit C defined over the finite field $\mathbb{F}_{11}$ that outputs the result of the operation $s_1 \times s_2 \times s_3 + s_4 \bmod 11$.

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

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

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

Recent years have seen a concentration of efforts towards different encodings of arithmetic circuits [GGPR13]. In the following, we define a classical form for encoding circuits in an algebraically useful way called *rank-1 constraint system* (R1CS). An R1CS encodes a program as a set of conditions over its variables, so that a correct execution of a circuit is equivalent to finding a satisfiable variable assignment. Due to the transformability of arithmetic circuits into R1CS, programs specified in R1CS are often referred to as *circuits*, and their variables as *signals*.

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

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

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

Example 2.2. We can represent the circuit C from Figure 2.1 as the following R1CS over S:

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

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

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

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

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

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

Since arithmetic circuits are composed by additions and multiplications, the representation of arithmetic circuits as R1CS is a natural transformation. Moreover, a valid witness for an arithmetic circuit translates naturally into a solution of the R1CS representing the circuit. This way, we say that an arithmetic circuit is *satisfiable* if there exists a solution to the R1CS representing the circuit. Checking satisfiability in R1CS encoded form requires to check all gates of a circuit. Most ZK protocols use aggregation techniques, such as *quadratic arithmetic programs*, to check all gates at once [PHGR13].

2.2 Elliptic curves

Elliptic curves that are defined over prime fields $\mathbb{F}_p$, with $p \geq 3$ prime, play an important role in this thesis. In Chapter 3, we use a specific curve called Baby Jubjub to illustrate the power and expressiveness of the CIRCOM language in some ECC constructions. In Chapter 4, we focus on twisted Edwards and Montgomery elliptic curves, which allow efficient circuit implementations. In Chapter 5, we study cycles of pairing-friendly curves, which leads us to review families of pairing-friendly curves of prime order. Since we focus on different types and aspects of elliptic curves in each chapter, in this section we give a very short introduction to elliptic curves defined over $\mathbb{F}_p$. Then, in each chapter, we include the material that covers the specifics for that chapter. In this section, we follow [MJ16].

Generally speaking, elliptic curves are geometric objects in projective planes over some given field, made up of points that satisfy certain equations. It is possible to describe an elliptic curves using different systems of coordinates. In this section, we use Weierstrass equations, which are the more general form to describe them. We assume $p > 3$ is prime and denote $\mathbb{F}_p$ the finite field with p elements.

Definition 2.5 (Projective Weierstrass equation). An *elliptic curve E over a finite field $\mathbb{F}_p$* (denoted $E/\mathbb{F}_p$) is a smooth projective curve associated to an equation of the form

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3, \tag{2.1}$$

with $a_1, \ldots, a_6 \in \mathbb{F}_p$.

Using a change of variables, we can simplify Equation (2.1).

Definition 2.6 (Projective short Weierstrass equation). An *elliptic curve $E/\mathbb{F}_p$* is the set of projective points in the projective plane $[X, Y, Z] \in \mathbb{P}_2$ satisfying

$$Y^2Z = X^3 + aXZ^2 + bZ^3, \tag{2.2}$$

with $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$. We call $d = 4a^3 + 27b^2$ the *discriminant* of the curve. The condition $d \neq 0$ ensures that Equation (2.2) has no double root.

Note that if $Z \neq 0$, a projective point (X, Y, Z) admits a representative with $z = 1$ with the change of coordinates $x = X/Y$ and $y = Y/Z$. The only point on E with $Z = 0$ is the point on the line at infinity representing all points whose coordinates are equivalent to $[0, 1, 0]$. This leads us to the affine definition of elliptic curves.

Definition 2.7 (Affine short Weierstrass equation). An elliptic curve $E/\mathbb{F}_p$ is the set defined by

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 \mid y^2 + x + ax^2 + b\} \cup \{O\}, \tag{2.3}$$

where $d = 4a^3 + 27b^2 \neq 0$ and O is an additional point called *the point at infinity* of the curve. We refer to $\#E(\mathbb{F}_q)$ as the *order* of the curve.

In the projective plane, an elliptic curve and a line have exactly three points of intersection [Ful08, Sec. 5.3]. This allows us to define a composition law, denoted as an addition, on the points of an elliptic curve.

Definition 2.8 (Group law). Let E be an elliptic curve defined as in Definition 2.7. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points E with $P_1, P_2 \neq \mathcal{O}$. We define $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

- If $x_1 \neq x_2$, then

$$\begin{aligned} m &= (y_2 - y_1)/(x_2 - x_1), \\ x_3 &= m^2 - A - x_1 - x_2, \\ y_3 &= m(x_1 - x_3) - y_1. \end{aligned} \tag{2.4}$$

- If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = O$.
- If $P_1 = P_2$ and $y_1 \neq 0$, then

$$\begin{aligned} m &= (3x_1^2 + A)/(2y_1), \\ x_3 &= m^2 - 2x_1, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{2.5}$$

- If $P_1 = P_2$ and $y_1 = 0$, then $P_1 + P_2 = O$.

Moreover, we define $P + O = P$ for all points P on E.

Theorem 2.1. The composition of points an elliptic curve defined as in Definition 2.8 makes E into an additive Abelian group with identity O.

Chapter 3

A circuit language for zero-knowledge applications

> *With software there are only two possibilities: either the users control the program or the program controls the users. If the program controls the users, and the developer controls the program, then the program is an instrument of unjust power.*
>
> – Richard Stallman

3.1 Introduction

Informally, a ZK proof system allows a prover to prove to a verifier that a statement is true without revealing any knowledge beyond the veracity of the statement [GMR85, GMW91, GO94]. In general, efficiency of ZK proof systems is measured considering three parameters: the computational cost of generating a proof, the size of the proof, and the time required to verify it. In the context of distributed ledgers, it is specially important to have small proof sizes and short verification times. The most popular, efficient, and general-purpose ZK protocols are ZK-SNARKs. Prominent applications of ZK-SNARKs are privacy-preserving blockchains such as Zcash [MGGR13, BSCG+14], which uses these proofs for verifying that private transaction have been computed correctly while providing complete anonymity to the participants of the network. More recently,

ZK-SNARKs are also used in conjunction with smart contracts for enhancing the scalability of distributed ledgers with solutions such as ZK-rollups. These applications bundle thousands of transactions into a single batch transaction with a ZK-SNARK proof that is verified by a smart contract.

Generally, ZK-SNARK protocols are used to prove the correctness of a computation. In this context, the way of expressing a computation is by defining it as an *arithmetic circuit* [BCC+16, PHGR13, BSCTV14b]. As we explained in Section 2.1.2, an arithmetic circuit is a circuit built with addition and multiplication gates, and wires that carry values from a prime finite field $\mathbb{F}_p$, where p is typically a very large prime number. A prover uses a circuit to prove that he knows a valid assignment to all wires of the circuit, and if the proof is correct, the verifier is convinced that the computation expressed as a circuit is valid, but learns nothing about the wires' specific assignment. The common encoding of this type of circuits is a R1CS, which is later used by the ZK-SNARK protocol to generate a proof. A valid proof shows, without revealing secret values, that the prover knows an assignment to all wires of the circuit that fulfill all constraints of the R1CS. An issue that appears when applying ZK protocols to complex computations, like a circuit describing the logic of a ZK-rollup, is that the amount of constraints to be verified is extremely large, up to hundreds of millions of constraints. In these cases, it is impractical to define circuits manually and we need tools for that. We can classify the tools in the ZK ecosystem in two main categories: *frontends* (languages) and *backends* (libraries).

3.1.1 Contributions and organization

While frontends provide a way of specifying computational statements, backends are involved in the generation and verification of the corresponding ZK proof. In this chapter, we present CIRCOM, a low-level language, also known as *constraint-based* or *hardware* language [OBW22], for specifying statements as circuits but aided by a *domain-specific language* (DSL) and its corresponding compiler [Ide20a, BBDM22, BMIMT+22].

Programmers can use the CIRCOM language to define arithmetic circuits and the compiler generates a file with the set of associated R1CS constraints together with a program (written either in C++ or WebAssembly) that can be run to efficiently compute a valid assignment to all wires of the circuit. One of the main particularities of CIRCOM is that it is designed as a modular language that allows the definition of parameterizable small circuits called *templates*, which can be instantiated to form larger circuits. CIRCOM users can create their own custom templates, but they can also use templates from CIRCOMLIB [Ide20b], a

publicly available library with hundreds of circuits such as comparators, hash functions, digital signatures, binary and decimal converters, and many more. The architecture behind CIRCOM not only provides a simple interface to model arithmetic circuits and generate their corresponding constraints, but it also abstracts the complexity of the underlying ZK proving mechanism. In particular, the output files of CIRCOM can be used directly by SNARKJS [Ide20c], which is a JavaScript library we developed to automatize the generation and verification of ZK-SNARK proofs.

This chapter is organized as follows. In Section 3.2, we present the main existing tools in the ZK space and compare them to CIRCOM. In Section 3.3, we introduce the characteristics of CIRCOM and give several examples of a correct use of the language. In the following Section 3.4, we present some practical applications that illustrate the power of the CIRCOM language. In Section 3.5, we evaluate the performance of CIRCOM in large circuits described by millions of constraints. In Section 3.6, we define the concepts of *correct* and *safe* CIRCOM programs, which can help programmers understand the philosophy of the language and help them with the writing of circuits. We close this chapter with brief conclusions in Section 3.7.

3.2 Related work

The appealing properties of ZK-SNARKs set off the development of software tools that allow practical ways to define statements, and to generate and verify ZK proofs. In this section, we give an overview of the main tools that exist in the ZK-SNARK ecosystem. In Figure 3.1, we give a visual classification of the tools according to their functionality and in the following sections we describe and compare them in detail.

3.2.1 Libraries

The first type of tools that were developed were libraries, also known as *backends*. Libraries are written in some general purpose programming language and provide functions that help users describe statements, and also generate and verify ZK proofs. In Table 3.1, we summarize the main existing libraries and their characteristics. LIBSNARK [Suc] and BELLMAN [Zcab] were the first libraries that came out. The former is written in C++ and is used as backend by many other tools. The later is a Rust crate for building ZK-SNARK circuits that provides circuit traits, primitive structures, and basic gadget implementations. The

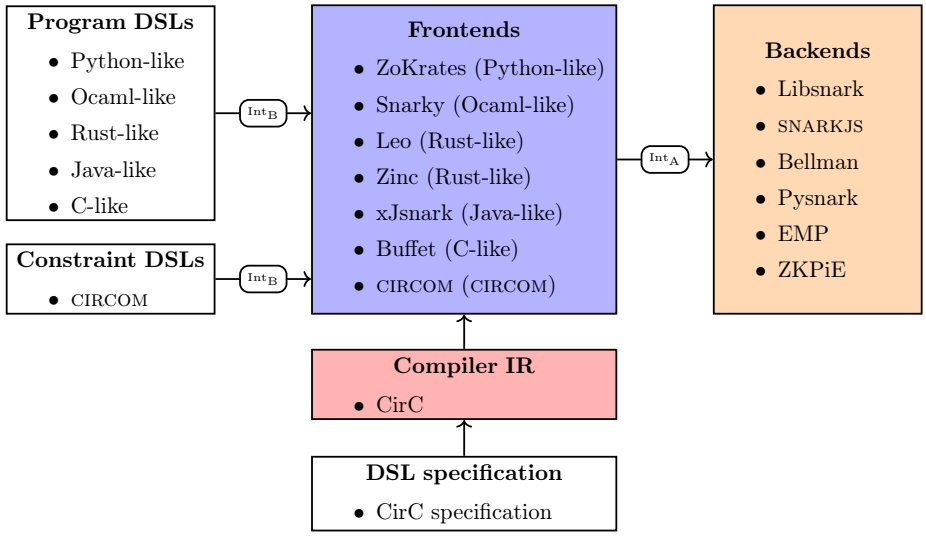

Figure 3.1: Classification of the main software tools for ZK-SNARKs.

second most popular backend in GitHub after LibSnark is snarkjs [Ide20c], a JavaScript library that we developed for generating and validating ZK proofs from a set of R1CS constraints. snarkjs can run in desktops and servers with NodeJS, and since it is written in JavaScript, it can also run seamlessly in browsers, bringing ZK proofs to the Web. A nice feature of this library is that it creates Solidity code to validate ZK-SNARK proofs within the Ethereum network. PySNARK [KXV] is a library for writing zk-SNARKs in Python 3. The library can be used in combination with LibSnark and snarkjs, and it also produces Solidity smart contracts automatically. There are also other backends that were designed for specific application scenarios. For instance, EMP [EMP] is a library that implements several interactive and communication-efficient ZK protocols [WYKW20, YSWW21, WYX+21] for proving statements in the context of neuronal networks. The protocols implemented by EMP are interactive and not based on R1CS. On the other hand, ZPiE [SD21, Sal] is an implementation of ZK-SNARKs based on R1CS compatible with snarkjs that is specifically designed for embedded systems. Since the application of these backends is narrowed, they are less popular and widespread than the aforementioned backends.

Backend	Application	Language	GitHub repository
LibSnark [Suc]	Desktop and server	C++	`scipr-lab/libsnark` (1,642 ⋆)
snarkjs [Ide20c]	Browser, desktop, and server	JavaScript	`iden3/snarkjs` (1,392 ⋆)
Bellman [Zcab]	Desktop and server	Rust	`zkcrypto/bellman` (777 ⋆)
PySNARK [KXV]	Python gadget library	Python	`meilof/pysnark` (132 ⋆)
EMP [EMP]	Interactive protocols	C++	`emp-toolkit/emp-zk` (55 ⋆)
ZPiE [SD21, Sal]	Embedded systems	C	`xevisalle/zpie` (16 ⋆)

Table 3.1: Classification of open-source backends for ZK applications. The stars from last column reflect the popularity of the tool in GitHub in June 2023.

3.2.2 Domain-specific languages

Over time, DSLs appeared as a more natural way of expressing computational statements being proved in ZK. In these cases, statements are expressed in a higher level DSL and compiled by the corresponding DSL compiler to lower level functions provided by a library. The main advantage of a DSL over a library is that a DSL allows users to express statements in the idiom and at the level of abstraction of the problem domain [MHS05]. Moreover, domain experts themselves (in our context, cryptographers and developers) can better understand, validate, modify, and develop DSL programs. Additionally, DSLs allow validation at the domain level, which is performed by a compiler. In the context of ZK, another advantage of a DSL is that the compiler can apply specific techniques to simplify the set of constraints. There are two categories of DSLs [OBW22] for ZK (interface named Int_B in Figure 3.1): program-based DSLs and constraint-based DSLs. We summarize the existing DSLs and their characteristics in Table 3.2.

Frontend	Type	DSL	Compiler	GitHub repository
ZoKrates [ET18, ZoKa]	Program	Python-like	Rust	`Zokrates/ZoKrates` (1,571 ⋆)
circom [BBDM22, Ide20a]	Hardware	circom	Rust	`iden3/circom` (858 ⋆)
Snarky [oL]	Program	OCaml-like	OCaml	`o1-labs/snarky` (447 ⋆)
Leo [CWC+21, Ale]	Program	Rust-like	Rust	`AleoHQ/leo` (440 ⋆)
Zinc [Mat19, Labb]	Program	Rust-like	Rust	`matter-labs/zinc` (309 ⋆)
xJsnark [KPS18, Kos]	Program	Java-like	Java	`akosba/xjsnark` (170 ⋆)
Buffet [WSR+15, Pepb]	Program	C-like	C, C++	`pepper-project/tinyram` (34 ⋆)

Table 3.2: Classification of open-source frontends for ZK applications. The stars from last column reflect the popularity of the tool in GitHub in June 2023.

In a program-based DSL, a statement is specified as a "program" written with a subset of instructions of a regular programming language that can be converted into primitives provided by the backend. In this case, the frontend compiler transcompiles the program and converts it into a circuit definition consisting of a set of constraints that can be proved by the backend. One of the first practical program-based DSL was ZoKrates [ET18], a Python-like DSL with a compiler written in Rust. This frontend was intended to help programmers use verifiable computation in their *decentralized application* (DApp) from the specification of a Python-like program for which proofs can be generated and finally verified by a Solidity smart contract. Snarky [oL] is an Ocaml-like program-based DSL with a backend based on LibSnark. To our knowledge, this is the only DSL built on top of a functional programming language. On the other hand, Zinc [Mat19, Labb] is a program-based DSL that borrows Rust's syntax and semantics with minor differences. In particular, Zinc does not allow recursion or variable loop indexes. Leo [CWC+21] is another Rust-like program-based DSL that abstracts the notions of native/non-native arithmetic and constraint types, which results in more expensive circuits in terms of constraints [CWC+21]. xJsnark is a Java-like program-based DSLs for zk-SNARKs that uses as cryptographic backend a Java interface to LibSnark. Finally, the Pepper project is an academic research project that has developed some tools for practical verifiable computation. In particular, their Buffet's C-to-C compiler [WSR+15, Pepb] supports proof-specific optimizations that is used by Pequin [Pepa], a toolchain to verifiably execute programs expressed in the C programming language.

On the other hand, in a constraint-based DSL, the statement being proved is specified directly as a circuit using arithmetic constraints [OBW22]. In this case, the DSL simplifies the task of writing constraints by allowing the definition of small circuits that act as components that can be connected with each other to form larger circuits. The constraint-based DSL compiler ensures that constraints are correctly specified and, like in all DSLs, it can also apply simplification techniques over the set of constraints defining a circuit.

3.2.3 Standardization tools

Finally, we would like to call attention to the efforts towards the creation of generic tools for building and prototyping compilers and towards the standardization of some of the interfaces in the ZK ecosystem. We summarize them in Table 3.3. CirC [OBW22, OCWS] is a shared compiler infrastructure for creating frontends that compile to constraint representations. To construct a

compiler with CirC, the developer essentially writes an interpreter for the DSL using the Circify library. The advantage is that the implementation using the *intermediate representation* (IR) provided by the Circify library is much shorter and faster to develop than building a full compiler from scratch. CirC is a very promising tool for prototyping and creating compilers quickly. Actually, the authors of CirC have created versions of ZoKrates and circom with less lines of code than the original compilers. Regarding the efforts towards standardizing interfaces between frontends and backends, a remarkable initiative is zkInterface [GKV+18, Qi], which specifies a protocol for communicating constraints, wire assignments, and proving protocols. These data are specified using language-agnostic calling conventions and formats to enable interoperability between different authors, frameworks, and languages.

Tool	Description	GitHub repository
CirC [OBW22, OCWS]	Tool for compilers.	`circify/circ` (207 ⋆)
zkInterface [GKV+18, Qi]	Standard tool for ZK interoperability.	`QED-it/zkinterface` (115 ⋆)

Table 3.3: Classification of open-source tools for software interoperability. The stars from last column reflect the popularity of the tool in GitHub in June 2023.

3.2.4 Comparative analysis

circom is a constraint-based description language for arithmetic circuits. To the best of our knowledge and according to the available literature [OBW22], circom is the only implemented DSL of this type. circom is in a level of abstraction between a library and a program-based DSL.

On the one side, as with libraries, circom users can specify the constraints of the circuits that define the statements. Moreover, libraries allow users to create or use already created gadgets (smaller circuits) and connect them with a program written in the language of the library. Similarly, circom users can make use of templates, which are small circuits that are parameterizable and can be instantiated to form larger circuits. circom also allows users to create their own custom templates or to use templates from circomlib. Among other things, the circom compiler takes care that template definitions, parameters, interconnections, and the R1CS constraints are correctly defined. Compared to libraries, with circom, the circuit building process is checked by the compiler

and possible errors are shown to users. This way, the CIRCOM language is a simple DSL for specifying templates and their interconnections.

On the other side, CIRCOM also supports splitting the circuit description into a pure *proving part* (constraints) and a pure *witness computation part*. This splitting is necessary when the witness computation requires operations that cannot be expressed as quadratic constraints. With this feature of the language, the compiler can automatically generate a program to efficiently compute a valid assignment to all wires of the circuit (the witness). That is, when we compile a circuit with CIRCOM, the compiler outputs the set of associated R1CS constraints and, if asked, it also can output programs for computing the witness efficiently. In particular, the compiler can output programs written in C++ (to be executed in a desktop/server) and in WebAssembly (to be executed in a browser). In comparison, program-based DSLs provide a higher level of abstraction by allowing users to specify statements as programs and the compiler transforms them into a circuit description. To do so, the compiler has to explore all paths through the program, unrolling all loops, considering all branches, while guarding all state modifications by the condition under which the corresponding path is taken [OBW22].

Although a program-based approach might offer the right level of abstraction to many programmers, they can actually not completely abstract themselves from the details of the underlying system. To illustrate this, we will make use of an example from the ZOKRATES official documentation [ZoKb, Section 3.4]: a circuit that given an input signal x, if $x \neq 0$ then the output is its inverse x^{-1}, and if $x = 0$, then the output is 0. A natural way of writing the corresponding circuit in ZOKRATES would be the following:

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

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

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

In the previous CIRCOM code, the first two constraints (lines 8 and 9) enforce that the signal named `iszero` is 1 if the input signal `in` is 0, and 0 otherwise (for a detailed explanation of the `iszero` constraints see Section 3.3.9). The last constraint (line 10), sets the output signal `out` to 0 if the input signal `in` is 0, and with any other number, `out` is set as the inverse of the given number. The main difference between the CIRCOM code and the ZOKRATES code is that in CIRCOM, the user can explicitly specify how exactly the `inv` intermediate signal is computed in the template (line 7), which avoids the division by zero issue of the ZOKRATES code and allows the witness computation part to run without issues. It is worth saying that the authors of ZOKRATES are currently working on an experimental feature that only activates constraints that are in a logically executed branch. However, this feature comes with a significant overhead of constraints [ZoKb].

On the other hand, the expressiveness and flexibility of constraint-based languages may also be a better option for those developers that, in order to create highly optimized circuits, wish to have greater control about the set of signals and constraints that define the computation. In this sense, the spirit of CIRCOM is to provide an unopinionated tool in which users can use the CIRCOM language to implement their optimizations at the template level. Moreover, as we show in Section 3.5, the CIRCOM compiler can apply several rounds of simplification of linear constraints, an option that can be activated or deactivated by the user.

Many projects in the Ethereum network are using the low-level approach provided by CIRCOM including payment mixers like Tornado cash [KV19], anonymous multi-asset pools like Zeropool [Zer], ZK signaling gadgets like Semaphore [WLG+], public-key cryptographic protocols like ECDSA [0xpa], decentralized ZK-RTS games like Dark Forest [DF], and ZK-rollups like Hermez [Her20], that use circuits described by hundreds of millions of constraints. Moreover, projects like zkREPL [zkR], which provide an online development environment for ZK-

SNARKs, are built on top of CIRCOM. As we mentioned in Section 3.2.3, the authors of CIRC have developed an alternative compiler implementation for the CIRCOM language that is considerably smaller than our Rust implementation of the compiler and achieves roughly the same performance. However, their comparison is with our previous version of the CIRCOM compiler (version 0.5), which was a prototype written in JavaScript. The compilation time of our current version written in Rust is, on average, 5 times faster than the JavaScript and CIRC versions for small and medium size circuits and can increase up to 10 times faster for large circuits.

3.3 CIRCOM

CIRCOM is a constraint-based DSL that allows programmers to design and create their own arithmetic circuits for ZK purposes. It is designed as a low-level circuit language, close to the design of electronic circuits. The CIRCOM compiler has more than 150K lines of Rust, WebAssembly, and C++ and is open source. CIRCOM allows programmers to define the constraints of an arithmetic circuit in a low-level but friendly way.

Recall from Section 2.1.2, that arithmetic circuits consist of operations in a finite field $\mathbb{F}_p$ that can be expressed as constraints of the form $\mathbf{a} \times \mathbf{b} - \mathbf{c} = 0$, where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are linear combinations over a set of signals $\{s_1, ..., s_n\}$. From a CIRCOM circuit description, the CIRCOM compiler outputs the corresponding set of constraints and a program that, given a set of input values, can compute an assignment to the rest of circuit signals. By default, CIRCOM takes p as the order of BN-128 elliptic curve, but the compiler also accepts the large prime dividing the order of BLS12-381, and the Goldilocks-like prime $2^{64} - 2^{32} + 1$. The user can select the prime to be used with a compiler's command-line option.

Example 3.1. Before going into details, we first illustrate how CIRCOM works with a circuit that will allow us to prove that the product of two secret input signals are equal to a certain public output.

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

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

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

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

3.3.1 Creating a circuit

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

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

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

3.3.2 Compiling a circuit

As we said as the beginning of Section 3.3, the use of the operator `<==` in the template named `Multiplier` has a double functionality: it captures the arithmetic relation between the signals, but it also provides a way of computing `c` from `a` and `b`. In general, the description of a CIRCOM circuit also keeps this double functionality. This way, the compiler can easily generate the R1CS describing a circuit but also the instructions to compute the intermediate and output values of a circuit. More specifically, given a circuit with the `.circom` extension the compiler can return four files. For example, we can compile `multiplier.circom` with the next options:

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

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

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

The prime number p being used to operate in the circuit is included in the header of the R1CS file, so that the backend can appropriately build the proof. The prime specification is also used by the compiler to generate the witness-calculator programs, which are linked to the correct modular arithmetic libraries to efficiently deal with the selected prime modular operations. Currently, we provide support and libraries for the three primes mentioned in at the beginning of Section 3.3.

3.3.3 Generating a ZK proof

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

In Figure 3.2 we show the complete process of generating and validating a ZK proof with our architecture. As we can see, we should first create a file containing the inputs written in the standard JSON format:

```
{"a": 3, "b": 11}.
```

Next, we pass the file with the inputs to the C++ or WebAssembly program generated by the compiler, which will generate a file containing the witness in binary format. After compiling the circuit and running the witness calculator with an appropriate input, we will have a file with extension `.wtns` that contains all the computed signals, and a file with the `.r1cs` extension that contains the constraints describing the circuit.

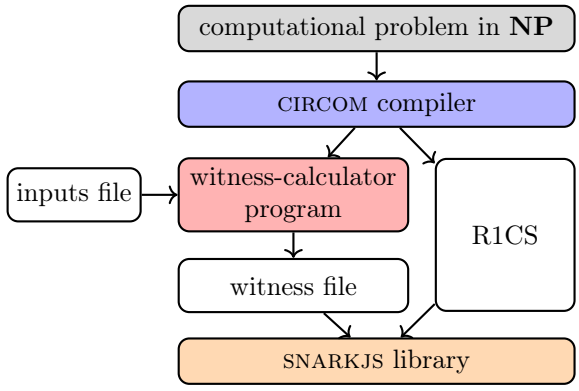

Figure 3.2: Our architecture for generating and verifying ZK-SNARK proofs using CIRCOM and SNARKJS software tools.

With the witness and the R1CS files, we can compute and verify ZK proofs using SNARKJS. All ZK protocols implemented in SNARKJS require a trusted

setup. In some cases, it is possible to reuse a trusted setup, like in [GWC19], whereas in others, it is necessary to generate a new trusted setup per circuit, as in [PHGR13] and [Gro16]. For this reason, SNARKJS already provides the necessary commands to create MPC ceremonies for generating the trusted setup and also verifying that an existing trusted setup has been computed correctly. From the R1CS and the MPC, SNARKJS produces a generation and a verification key for the circuit. Finally, with the generation key and the witness, the prover can generate a ZK-SNARK proof and send it to a verifier, who uses the verification key and a file with the public signals of the circuit to check if the prover's proof is valid. Further information about the creation of a trusted setup and the generation and verification of ZK-SNARK proofs with SNARKJS can be found in [Ide20c].

Note that we could have started the process choosing different input values. For example, we could have used `{"a": 1, "b": 33}` as input and generated a valid proof for our circuit. Hence, a proof for the circuit `Multiplier` would not really show that we know how to factor 33. In Section 3.3.9, we will use a template that checks if a signal is zero to modify the template `Multiplier` to only accept inputs that are not 1.

3.3.4 The `main` component

The CIRCOM compiler needs a specific component as entry point. This initial component is called `main` and, as we did in Example 3.1, it needs to be instantiated with some template.

Unlike other intermediate components that we will introduce later, the `main` component defines the global input and output signals of a circuit. As mentioned in Section 3.3.3, by default, the global inputs are considered private signals while the global outputs are considered public. However, the `main` component has a special attribute to set a list of global inputs as public signals. The general syntax to specify the `main` component is the following:

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

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

Example 3.2. Let us illustrate the use of public signals following the previous example. For simplicity, we will no longer start out code with the `pragma` instruction.

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

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

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

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

3.3.5 Connecting templates

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

Example 3.3. Let us illustrate how to connect templates by continuing our previous example. In Example 3.1, we created a template for a multiplier of two signals. In this case, we will extend this idea by connecting two of these 2-input multipliers to get a multiplier for three signals.

```
include "multiplier.circom";

template Multiplier3() {
  signal input in1;
  signal input in2;
  signal input in3;
  signal output out;

  component multiplierA = Multiplier();
  component multiplierB = Multiplier();

  multiplierA.a <== in1;
  multiplierA.b <== in2;
  multiplierB.a <== multiplierA.c; // in1 * in2
  multiplierB.b <== in3;
  out <== multiplierB.c; // (in1 * in2) * in3
}

component main {public [in1, in2]} = Multiplier3();
```

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

Remark 3.1. From a template, we can only access the inputs and outputs of its direct subcomponents.

3.3.6 Debugging

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

Example 3.4. Following Example 3.3, we can use the logging function to show a string followed by the value of the signal `multiplierA.c`.

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

3.3.7 Building complex circuits

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

Example 3.5. Let us illustrate how to build more complex circuits by generalizing Example 3.3 to an n-multiplier. That is, we will create a parametrized

template that will allow the instantiation of circuits that will verify the multiplication of n input values.

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

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

Remark 3.2. It is useful to think of building CIRCOM circuits as a similar process of building electronic circuits. With CIRCOM circuits, the compiler must know all the required parameters of the circuit. As a result, in loops that involve constraints (*symbolic* part), CIRCOM only allows to define the loop condition based on the template parameters. In our previous example, the loop condition used `n`, which was perfectly valid. For further information, see Section 3.3.12.

3.3.8 Splitting between computation and constraints

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

Example 3.6. A division $c = a/b$ is an operation that cannot be computed using a quadratic formula but it can be checked using the quadratic expression $a = b \cdot c$.

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

In this case, we have to split the computational task, which instructs the compiler in how to compute signals, from the symbolic task, which instructs the compiler in how to create constraints that verify a computation (see Section 3.3.2). As we can see in the code, the computational task is expressed using the individual operator `<--` (line 5). The language also accepts the left-to-right operator `-->` with the same semantics. On the other side, the symbolic task is expressed separately using the individual operator `===`, which adds a constraint that captures the quadratic relation between signals (line 6).

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

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

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

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

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

In general, the *dual operator* `<==` is preferred whenever possible, because it always guarantees the equivalence between the computed witness and the constraints that check the computation. Notice that, if not handled with care,

the use of the *individual operators* `<--` and `===` might produce a situation in which the witness does not fulfil the constraints or in which constraints are disconnected from the witness.

Example 3.7. Let us look at the following template, which given two inputs `a` and `b`, it outputs `c = a + b`.

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

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

To avoid these cases, individual operators must only be used in cases in which the dual operator cannot express a computation like it happened in Example 3.6. In Section 3.6, we analyse these situations in greater detail.

Remark 3.3. Neither the operator `===` nor `<==` can be used with signal expressions that are not quadratic.

3.3.9 Checking if a signal is zero

Now that we know the basic syntax of the CIRCOM language, we present the template `IsZero`, which has some subtleties. `IsZero` checks if a certain signal in a circuit is zero or not. In this case, the output signal `out` is 1 if the input signal `in` is zero, and 0, otherwise. The circuit is based on a trick from [PHGR13].

```
template IsZero() {
  signal input in;
  signal output out;
  signal inv;
  inv <-- in!=0 ? 1/in : 0;
  out <== -in * inv + 1;
  in * out === 0;
}

component main = IsZero();
```

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

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

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

Note that when `in` is 0, we decided to assign 0 to `inv`, but in fact, we could have chosen any other value. Indeed, when `in` is zero, both constraints (lines 6 and 7) are satisfied. In this case, we say that the circuit is *safe*, but not *strongly safe*, since there is more than one valid solution for `inv`. We analyse this type of situations in greater detail in Section 3.6.

Example 3.8. The template `IsZero` is used very frequently. An illustrative example, is to use it to modify our first template `Multiplier` from Example 3.1 to enforce that none of its inputs is 1. For that, we use the fact that `a` is not 1 if and only if `a-1` is not zero, and the same stands for `b`.

```
1  include "iszero.circom"
2
3  template Factorization() {
4    signal private input a;
5    signal private input b;
6    signal output c;
7
8    component isz1 = IsZero();
9    component isz2 = IsZero();
10
11   isz1.in <== a-1;
12   isz2.in <== b-1;
13   isz1.out === 0; // enforce that a-1 != 0
14   isz2.out === 0; // enforce that b-1 != 0
15   c <== a * b;
16 }
17
18 component main = Factorization();
```

3.3.10 Functions and constants

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

Example 3.9. An example of a basic function is the following one, which adds one to a given value:

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

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

Example 3.10. The following function will be later used in Section 3.4.2 and it returns a parameter of an elliptic curve.

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

3.3.11 Symbolic variables

In Section 3.3.7, we explained several uses of the variables when building circuits. However, variables have another important use, which is to store symbolic expressions when building the constraints. We call *symbolic variables* to those variables that contain symbolic expressions on signals.

Example 3.11. Let us analyse an example of a template that uses symbolic variables. The following template implements a multiAND circuit that depends on a parameter n. That is, `MultiAND` is a template that takes an array of `n` binary inputs and outputs 1 if and only if all elements of the array are 1.

```
1  include "iszero.circom"
2
3  template MultiAND(n) {
4    signal input in[n];
5    signal output out;
6    var sum = 0;
7
8    for(var i=0; i<n; i++) {
9      sum = sum + in[i];
10   }
11
12   component isz = IsZero();
13
14   sum - n ==> isz.in;
15   isz.out ==> out;
16 }
17
18 component main = MultiAND(4);
```

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

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

$$\texttt{in[0] + ... + in[n-1] - n = isz.in}.$$

Example 3.12. In the following example, we analyze a template that given an input signal `in`, it outputs the binary representation of `in` as an `n`-array of signals called `out[n]`. For a given number `n`, we could use the following list of quadratic constraints:

```
1 out[0] * (out[0]-1) === 0
2 [...]
3 out[n-1] * (out[n-1]-1) === 0
4
5 out[0] * 2^0 +...+ out[n-1] * 2^(n-1) - in === 0
```

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

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

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

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

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

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

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

```
template Num2Bits(n) {
  signal input in;
  signal output out[n];
  var bsum = 0;
  var exp2 = 1;
  for (var i = 0; i<n; i+=1){
    out[i] <-- (in >> i) & 1;
    out[i] * (out[i]-1) === 0;
    bsum += out[i] * exp2;
    exp2  * = 2;
  }
  bsum === in;
}
```

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

3.3.12 Dealing with the *unknown*

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

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

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

1. The addition of a constraint that depends on a Boolean condition involving an unknown variable.

2. The addition of a constraint with an array access using as index an unknown variable or a signal.

Example 3.13. Let us see an example of a CIRCOM program that does not compile because of the unknown.

```
1 template ErroneousTemplate1(n) {
2   signal input in;
3   signal output out1;
4   signal output out2;
5   for(var i=0; i<n; i++) {
6     out1 <== in  *  in;
```

```
7       if(in >= 0){
8         out2 <== in + 2;
9       }
10    }
11  }
12
13  component main = ErroneousTemplate1(4);
```

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

Example 3.14. In this other example, we illustrate the situation in which, to create a constraint, a symbolic variable (unknown) is used to access an array.

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

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

As a result of the previous discussion, circuits, which are defined by a set of R1CS constraints, must be known at compilation time. However, in certain occasions, it may be useful to do computations requiring accesses to positions in arrays or memories that are unknown at compilation time, e.g. depending on the value of an input signal. When using CIRCOM, the user has to build the circuits that arithmetize this type of computations. These arithmetizations are

not built-in features of CIRCOM, because by design, CIRCOM is unopinionated in how arithmetizations are implemented and rather, these arithmetizations should be part of template libraries. In the literature, we find several approaches for such types of arithmetizations. For instance, [BFR+13] uses a line-by-line compilation approach with instructions for memory reads and writes, and a hash structure to store the current memory state, while [BSCTV14b] uses a permutation network to verify that the sequence of memory reads and writes is consistent. Buffet [WSR+15, Pepb] uses a combination of [BFR+13] and [BSCTV14b] to build an efficient arithmetization of the random access memory.

3.3.13 Using templates from CIRCOMLIB

As we have explained in the previous sections, the use of templates allows CIRCOM developers to build large circuits from smaller individual subcircuits. In this regard, CIRCOM users can create their own custom templates, but in addition to the language and the compiler, we also provide an open-source library of CIRCOM templates called CIRCOMLIB [Ide20b], with hundreds of different circuits. On the one side, CIRCOMLIB has the implementation of basic operations, such as binary logic gates, comparators, conversions between field elements and their binary representations, and multiplexers. On the other side, the library contains more complex circuit structures that are used in the context of distributed ledgers and cryptocurrencies, such as digital signatures, elliptic curve-based cryptographic schemes, hash functions, and Merkle tree structures. We would like to remark that apart from CIRCOMLIB, there is a community actively using CIRCOM for building their own custom templates. Remarkable examples are an elliptic-curve pairing implementation from 0xParc [0xpb] and a CIRCOM-based library from Electron Labs that allows to generate proofs for a batch of Ed25519 signatures [Laba]. In the following Section 3.4, we show how to make use of CIRCOMLIB templates and present some practical applications of CIRCOM.

3.4 Applications

Most of the applications that use ZK proofs need to prove the correctness of computations such as hash functions, public key derivations, and digital signatures. In this section, we present some implementations of these circuits, which illustrate the expressiveness and potential of the CIRCOM language. In Section 3.4.1, we give an example of a circuit that allows us to prove that we know the

preimage of a hash value using templates from CIRCOMLIB. In Section 3.4.2, we introduce templates that implement the arithmetic operations on the elliptic curve called Baby Jubjub [BWB+21]. In Section 3.4.3, we explain how to use the previous curve operations to verify that a private key corresponds to a public key without revealing the private key. Finally, in Section 3.4.4, we explain how to verify a signature with templates from CIRCOMLIB and give an example of a circuit that verifies that a given message has been signed by a public key from a pair of authorized public keys, but without revealing which of the two was used.

3.4.1 Hash functions

A *cryptographic hash function* is a deterministic one-way function that maps data of an arbitrary size to a bit array of a fixed size. Hash functions are widely used in authentication systems to avoid storing plaintext passwords in databases, but are also used to identify and validate the integrity of files, documents, and other types of data. One of the main uses of hash functions is in digital signatures, where the hash is used to create a cryptographic digest of the data being signed (see Section 3.4.4).

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

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

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

```
include "sha256.circom";

template Main() {
  signal input in[300];
  signal output out[256];

  component sha256 = Sha256(300);

  for (var i=0; i<300; i++){
    sha256.in[i] <== in[i];
  }
  for (var i=0; i<256; i++){
    out[i] <== sha256.out[i];
  }
}

component main = Main();
```

In line 7, we instantiate the template `Sha256(nBits)` with `nBits` = 300. In this case, we have to assign the values of the signal array bit by bit (line 10). Finally, we set each bit of `out` to each bit of the output of the `sha256` component (line 13).

Classical hash functions, such as the family of SHA functions [Han05], are heavy on bit operations, which makes them very inefficient to implement inside arithmetic circuits. For example, the previous template `sha256` from CIRCOMLIB for an input of 300 bits is described by 29,450 constraints. Recently, there have been efforts to develop new hash functions that optimize their representation inside arithmetic circuits. In this regard, CIRCOMLIB also contains the implementation the Pedersen hash [LMS17] (`pedersen`), two hash functions from the MiMC family [AGR+16] (`mimc, mimc_sponge`), and Poseidon [GKR+21] (`poseidon`).

3.4.2 Elliptic-curve arithmetic

A classical use of ZK protocols is to prove ownership of a public key without revealing the secret key. For that, we need to be able to write the logic of verifying that a given secret key corresponds to a given public key inside an $\mathbb{F}_p$-arithmetic circuit. This logic is usually implemented by means of arithmetic operations of an elliptic curve. In this section, we show how to implement the arithmetic operations of an elliptic curve called *Baby Jubjub* [BWB+21], used in the Ethereum blockchain to implement elliptic-curve cryptography inside circuits [WBB20]. We describe this curve in great detail later in Chapter 4.

Defining the parameters of the curve

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

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

and described by equation

$$ax^2 + y^2 = 1 + dx^2y^2, \tag{3.1}$$

with $a = 168700$ and $d = 168696$. Baby Jubjub consists of the set of points (x, y) with $x, y \in \mathbb{F}_p$ that satisfy Equation (3.1), together with the point at infinity, which is usually represented by $(1, 0)$.

To avoid replicating the values of a and d from Equation (3.1) in every template to the curve, it is useful to define them only once. As we explained in Section 3.3.10, CIRCOM does not admit the definition of global constants and, instead, we have to to define two functions that always return these values.

```
1 function baby_const_a(){
2   return 168700;
3 }
4
5 function baby_const_d(){
6   return 168696;
7 }
```

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

Checking if a point belongs to the curve

We start by checking if a pair of coordinates (x, y) correspond to a point on the curve that safisfies Equation (3.1). For that, we create a template called `BabyCheck()`, that verifies if a pair of x and y are a solution to the equation.

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

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

$$\texttt{a*x*x + y*y === 1 + d*x*x*y*y},$$

as in Equation (3.1), since it is not a quadratic expression. Instead, we use two new intermediate signals, `x2` and `y2`, to represent x^2 and y^2 (lines 8- 9). Once these signals are defined, we can check if the point `(x,y)` belongs to the curve using the quadratic constraint

$$\texttt{a*x2 + y2 === 1 + d*x2*y2}.$$

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

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

Adding two points in the curve

Now, we define how to operate in the elliptic-curve group. For that, we use that the addition of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on Baby Jubjub is defined [BWB+21] as a third point $P_3 = (x_3, y_3)$ with coordinates

$$x_3 = \frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2} \quad \text{and} \quad y_3 = \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2}. \tag{3.2}$$

The following piece of code consists of a template, called `BabyAdd`, that takes two points and outputs their addition using the formula from Equation (3.2).

```
template BabyAdd() {
  signal input p1[2];
  signal input p2[2];
  signal output pout[2];

  signal beta;
  signal gamma;
  signal delta;
  signal tau;
  var a = baby_const_a();
  var d = baby_const_d();

  beta <== p1[0] * p2[1];
  gamma <== p1[1] * p2[0];
  delta <== (-a * p1[0]+p1[1]) * (p2[0] + p2[1]);
  tau <== beta  *  gamma;

  pout[0] <-- (beta + gamma) / (1+ d * tau);
  (1+ d * tau)  *  pout[0] === (beta + gamma);

  pout[1] <-- (delta + a * beta - gamma) / (1-d * tau);
  (1-d * tau) * pout[1] === (delta + a * beta - gamma);
}
```

In this template, we define points using 2-dimensional arrays of signals. In particular, we have two points as input signals (`p1` and `p2`), and a third point as output signal (`pout`). We also have four intermediate signals (`beta`, `gamma`, `delta`, and `tau`), and two variables (`a` and `d`) with the coefficients from Equation (3.1). Since both expressions from Equation (3.2) involve a division by signals, we cannot write the formulas directly using the dual operator `<==`. Instead, we first use the individual computational operator `<--` to compute the

denominators, and then, use the individual symbolic operator `===` to enforce a multiplicative relation between the numerator and the denominator.

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

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

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

3.4.3 Public-key cryptography

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

```
template BabyPbk() {
  signal input in;
  var GEN[2] = [
  5299619240641551281634865583518297030 2
  8287447219077289408652114448272100155 3,
  1695015079846065771795862556782183455 0
  3016631616247077872228159361826389682 03
  ];
  signal output Ax;
  signal output Ay;

  component pvkBits = Num2Bits(253);
  pvkBits.in <== in;
  component mulFix = ScalarMulFix(253, BASE8);
```

```
15    var i;
16
17    for (i=0; i<253; i++) {
18      mulFix.e[i] <== pvkBits.out[i];
19    }
20
21    Ax  <== mulFix.out[0];
22    Ay  <== mulFix.out[1];
23  }
24  component main = BabyPbk();
```

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

After these definitions, we declare a component (`Num2Bits`) to transform the scalar `in` to its 253-bit representation, and assign `in` as its input signal. Right after, we declare a new component (`ScalarMulFix`) to perform the multiplication of the generator by the scalar. Details about how this multiplication is performed can be found in [Ide20b]. In line 16, each of the bits from the representation of the scalar are set to its corresponding input of the component. Finally, we assign the output signals of this component to the final output signals `Ax` and `Ay`.

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

3.4.4 Digital signatures

A popular elliptic curve-based signature scheme is the *Edwards-curve digital signature algorithm* (EdDSA) [JL17], which is a digital signature scheme based on twisted Edwards curves, as is Baby Jubjub. Given a public key as defined in Section 3.4.3, and a message, the EdDSA protocol uses a public cryptographic hash function to bind the signature to a given message and public key.

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

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

```
1  include "eddsa-simplified.circom";
2
3  template VerifyAuthorizedSignature() {
4    signal input pk1[2]; // public key 1
5    signal input pk2[2]; // public key 2
6    signal input msg; // message
7    signal input sig; // signature
8
9    signal out1;
10   signal out2;
11
12   component verify1 = EdDSAVerifier();
13   component verify2 = EdDSAVerifier();
14
15   // verify signature with pk1
16   verify1.pk <== pk1;
17   verify1.msg <== msg;
18   verify1.sig <== sig;
19   out1 <== verify1.out;
20
21   // verify signature with pk2
22   verify2.pk <== pk2;
23   verify2.msg <== msg;
24   verify2.sig <== sig;
25   out2 <== verify2.out;
26
27   out1 + out2 === 1;
28 }
29
30 component main {public [pk1[0],pk1[1],pk2[0],pk2[1],msg]}
31 = VerifyAuthorizedSignature();
```

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

3.5 CIRCOM performance on large circuits

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

To reduce the amount of constraints describing a circuit, the CIRCOM compiler simplifies the linear constraints. More specifically, the compiler divides the set of constraints into clusters of related linear constraints and then applies the classical Gauss-Jordan elimination to each of them. These optimizations are iterated until it is no longer possible to optimize more linear constraints. The compiler treats clusters independently which allows to parallelize the optimization subprocesses. The compiler runs these optimizations by default but the user can choose to turn them off. Currently, there is also an ongoing work on non-trivial optimization techniques applied to R1CS [ABI+22].

3.5.1 ZK-rollup circuits

To evaluate the performance of CIRCOM with large circuits, the language and the compiler have been analyzed with the ZK-rollup circuits of the Hermez [Her20] project. As we explained in Section 3.1, a ZK-rollup [GBFS16] is a construction intended to increase the scalability of Ethereum by performing calculations off-chain, rolling many transactions up into a single batch, and sending it to the main Ethereum chain for processing in one action. In more detail, a ZK proof is generated off-chain for every batch of transactions and it proves the validity of every transaction in the batch. This means that it is not necessary to rely on the Ethereum main chain to verify each signed transaction.

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

3.5.2 Performance results

In Table 3.4, we show the number of generated constraints, the size of the R1CS file, and the compilation time for different instances of ZK-rollup circuits. We also show their corresponding gains and losses before and after applying the simplification of linear constraints. The results have been obtained from an AMD Ryzen Threadripper 3990X 64-Core Processor with 270GB of RAM (Linux Kernel 5.4.0-80-generic).

Number of constraints			
Circuit	*without simplification*	*after simplification*	*gain*
ZK-Rollup-256	134,267,317	24,301,347	81.9%
ZK-Rollup-512	197,926,325	37,792,099	80.9%
ZK-Rollup-1024	325,244,341	64,773,603	80.1%
ZK-Rollup-2048	579,880,373	118,736,611	79.5%
ZK-Rollup-2341	652,925,030	134,203,765	79.4%

Size of the `.r1cs` file			
Circuit	*without simplification*	*after simplification*	*gain*
ZK-Rollup-256	15.7GB	8.5GB	45.9%
ZK-Rollup-512	23.4GB	13.7GB	41.5%
ZK-Rollup-1024	38.8GB	24.1GB	37.9%
ZK-Rollup-2048	69.5GB	44.6GB	35.8%
ZK-Rollup-2341	78.4GB	51.1GB	34.8%

Compilation time			
Circuit	*without simplification*	*after simplification*	*overhead*
ZK-Rollup-256	12.1min	38.9min	×3.22
ZK-Rollup-512	17.5min	58.3min	×3.33
ZK-Rollup-1024	28.4min	111.2min	×3.92
ZK-Rollup-2048	50.6min	512.5min	×10.14
ZK-Rollup-2341	56.5min	618.9min	×10.95

Table 3.4: Comparison of different Hermez ZK rollup circuits before and after the CIRCOM compiler applies optimization techniques to reduce the number of constraints describing the circuits.

As the experimental evaluation shows, in circuits this large, the compiler's optimizations are crucial to handle the huge amount of constraints. For instance, for ZK-Rollup-256, CIRCOM without simplification generates 134,267,317

constraints whose file size is 15.7GB and the time needed for the compilation is 12.05min. On the other hand, the simplification allows us to reduce the number of constraints up to 24,301,347, whose file size is 8.5GB and the compilation time is increased up to 38.92min. Note that in this case, the reduction on the number of constraints is close to 82%, whereas the size of the `.r1cs` file is not reduced in the same proportion. This is due to the fact that constraint simplification often implies the addition of new variables in the remaining constraints.

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

Note that with circuits from Table 3.4, CIRCOM produces around 100 million constraints (500 million without simplification). With these numbers, ZK-rollups can handle around 2,000 transactions. Take into account that the software used afterwards to generate and validate ZK-SNARK proofs also have bounds inherited from the ZK protocol. In fact, thanks to simplification, we can handle up to 2,341 transactions without exceeding the limit of 2^{27} constraints that SNARKJS can handle [Ide20c]. Processing a batch of this size needs less than 2.1 million gas, so with this amount of transactions per batch and the current Ethereum gas limit per block of 30 millions, we have that we can process 32,774 transfer transactions per block, which is around 23 times more transfers than if they were executed directly in the Ethereum blockchain.

3.6 Analysis

Let $\mathbb{C}^{n\times t\times m}$ be the set of all circuits that can be programmed in CIRCOM with n input signals, t intermediate signals, and m output signals. Given a circuit $C \in \mathbb{C}^{n\times t\times m}$, we denote by $\mathcal{C}(C)$ the set of constraints generated by CIRCOM after compiling C. Let $W : \mathbb{C}^{n\times t\times m} \times \mathbb{F}_p^n \to \mathbb{F}_p^t \times \mathbb{F}_p^m$ be a partial function that takes a circuit $C \in \mathbb{C}^{n\times t\times m}$ and n values for the input signals, such that it returns the t values of the intermediate signals and the m values of the output

signals. The function W describes the computation made by the executable code obtained from compiling C after the input values are given. Note that W is a partial function, since not every input of a circuit produces a valid output.

Definition 3.1 (Correct CIRCOM *program).* A CIRCOM program $C \in \mathbb{C}^{n \times t \times m}$ is said to be *correct* if for every given values $\vec{i} \in \mathbb{F}_p^n$ for the n input signals of C, we have that: if $\mathcal{C}(C)$ replacing the inputs signals by $\vec{i}$ is satisfiable, then $W(C, \vec{i}) = (\vec{t}, \vec{o}) \in \mathbb{F}_p^t \times \mathbb{F}_p^m$ and $(\vec{i}, \vec{t}, \vec{o})$ is a solution to the system $\mathcal{C}(C)$. Otherwise, we say that C is *incorrect*.

A CIRCOM program is called *strongly safe* when the values computed by the executable code are the unique solution to the R1CS constraint system. However, sometimes this notion of safety involving all signals including the intermediate ones could be too strong for some components, as it happens with the `IsZero` circuit from Section 3.3.9. In that case, when the value of signal `in` is 0, the computation sets the intermediate signal `inv` to be also 0, but `inv` could have taking any other value and still satisfy the constraints from the template. For these reasons, there is an alternative weaker notion of safety, which only requires the constraints and the code to meet on inputs and outputs, but not necessarily on the intermediate signals.

Definition 3.2 (Safe CIRCOM *program).* A CIRCOM program $C \in \mathbb{C}^{n \times t \times m}$ is said to be *strongly safe* if for every given values $\vec{i} \in \mathbb{F}_p^n$ for the n input signals of c, we have that: if $W(C, \vec{i}) = (\vec{t}, \vec{o}) \in \mathbb{F}_p^t \times \mathbb{F}_p^m$, then $(\vec{i}, \vec{t}, \vec{o})$ is the only solution of $\mathcal{C}(C)$, and it is said to be *safe* if all solutions of $\mathcal{C}(C)$ are of the form $(\vec{i}, \vec{t'}, \vec{o})$. Otherwise, the program is called *unsafe*.

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

Lemma 3.1. A strongly safe CIRCOM program is deterministic.

Proof. From definition 3.2, we deduce that, given a safe CIRCOM program C and an input $\vec{i}$ for this program, if there exists an output $\vec{o}$ and an intermediate $\vec{t}$ for C, then it must be unique.

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

Lemma 3.2. A CIRCOM program is strongly safe if it is written without using `<--` and `-->` and all its intermediate and output signals are the target of an assignment operation.

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

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

1. If the expression involves signals whose computation has been already defined, then it is the backend's job to ensure that the set of constraints is satisfied by the set of computed inputs. That is, given a set of inputs, if the witness-generator program produces a set of signals which do not satisfy the set of constraints, the backend will not be able to produce a valid proof. In other words, the security is guaranteed by the backend.

2. If the `===` involves a new signal that should be computed (using the `<--` operator) but this computation is not defined in the CIRCOM description, the compiler will detect this fact and throw an error.

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

1. If the `<--` operator is backed up by an associated constraint or constraints (using `===`), then the CIRCOM description is as safe as using the double operator `<==`.

2. If the `<--` operator is not backed up by its associated constraints, then this means that the CIRCOM description lacks constraints. In this case, the backend cannot help the user to detect missing constraints, it just generates proofs that can be verified but that have fewer constraints than might be necessary.

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

3.7 Conclusions

In this chapter, we presented CIRCOM, a constraint-based DSL for describing ZK circuits. CIRCOM is in a level of abstraction between a program DSL and a library and, to the best of our knowledge and according to the available literature [OBW22], it is the only implemented DSL of this type. The CIRCOM compiler is responsible for generating all the necessary material to, later, generate and verify ZK-SNARK proofs. The philosophy of CIRCOM is that programmers have full control over the exact construction of arithmetic circuits and the resulting set of constraints which, at the end, are the ones used to build ZK proofs. CIRCOM is modular at many levels, and to deal with extra constraints introduced by the interconnection of templates, we implement several rounds of optimizations of linear constraints in the compiler, which are crucial for the use of CIRCOM in industrial circuits describing real-world problems.

Chapter 4

Twisted Edwards elliptic curves for arithmetic circuits

Design is where art and science break even.

– Robin Mathew

4.1 Introduction

In 2008, blockchain was added to the list of practical uses of ECC. In Satoshi's seminal Bitcoin paper, ECC was used for securing the various transactions occurring on the network, for controlling the generation of new currency units, and verifying the transfer of digital assets and tokens. A few years later, privacy-oriented cryptocurrencies incorporated new cryptographic techniques to ensure user anonymity and obfuscate payment details. For instance, Monero started using ring signatures and Pedersen commitments [NML16], while Zcash used ZK-SNARKs [MGGR13, BSCG+14]. As we have seen in the previous chapters, before proving computational statements with ZK-SNARKs, statements have to be expressed as an $\mathbb{F}_p$-arithmetic circuit, also called *ZK circuit* or *ZK-SNARK circuit*. The specific instantiation of the ZK protocol is what determines p, but typically p is a large prime number of approximately 254 bits that is determined by the order of a pairing-friendly elliptic curve [WBB20]. For instance,

in Ethereum, p is the order of the BN256 elliptic curve, and in Zcash, p is the large prime order subgroup of BLS12-381.

Classical cryptographic schemes consisted mostly of Boolean operations, which makes them inefficient when evaluated inside a ZK-SNARK circuit. As an example, the Zcash circuit relied on the SHA256 hash function to create a message-authentication code to prevent malleability, for generating pseudo-random strings, and for commitments. However, each invocation of SHA256 added tens of thousands of multiplication gates to ZK-SNARK circuits, making this hash the primary cost when generating ZK-SNARK proofs [zcaa]. These issues motivated the search for algebraic primitives to replace SHA256 and other inefficient functions. So, instead of hash functions such as SHA256 inside ZK-SNARK circuits, the idea was to use ECC that works in large prime fields, which is the natural representation of circuits. For this reason, new schemes that relied on elliptic curves were gradually adopted in ZK-SNARK constructions.

In this context, two prominent examples are Pedersen hashes [LMS17] and EdDSA [JL17]. These schemes can be built efficiently by using elliptic curves that can be represented in twisted Edwards form, using the fact that they have a single formula for doubling and adding points of the curve [BBJ+08]. There is another form of representing elliptic curves called Montgomery, that makes computations faster but has different formulas for adding and doubling points [LM87]. A twisted Edwards curve is generally birationally equivalent to a Montgomery curve, so curves can be easily converted from one form to another [BBJ+08]. Inside a circuit, we can use the Montgomery form when we know for sure that either we are adding different points or we are adding the same point, and use twisted Edwards when, depending on the inputs of the circuit, this cannot be assured. Combining the two forms in this way makes the implementation of the group law as an arithmetic circuit very efficient.

In order to implement the Pedersen hash and EdDSA as circuits, we need curves that are defined over $\mathbb{F}_p$, where p is determined by the particular choice of pairing-friendly elliptic curve used to generate ZK-SNARK proofs. It is crucial to choose an appropriate twisted Edwards elliptic curve with optimal parameters for the cryptographic schemes, since the choice of the curve has great impact on their security and efficiency. Moreover, curves need to be generated in a transparent and deterministic way, so that anyone can audit and recreate the procedure. Transparency is paramount, as it significantly reduces the possibility of a backdoor being present, thus leading to better security. It is also crucial that the new curves are also tested for resilience against best known attacks, such as the rho method, or additive and multiplicative transfers, which attack the discrete logarithm problem over elliptic curve groups [BSS99].

4.1.1 Contributions and organization

In this chapter, we present a set of deterministic algorithms that, given a field $\mathbb{F}_p$, allows us to generate secure twisted Edwards elliptic curves that are suitable for $\mathbb{F}_p$-arithmetic circuits and allow the efficient computation of ZK-SNARK proofs that prove ECC statements. There have already been two curves that have been generated using these methods. On the one side, Jubjub curve for Zcash, defined over the scalar field of BLS12-381 and on the other side, Baby Jubjub for Ethereum, defined over the scalar field of BN256. The present work is a formalization and generalization of the common efforts to generate suitable twisted Edwards curves for ZK-SNARK circuits.

The chapter is organized as follows. In Section 4.2, we introduce related work on generation of elliptic curves in the ZK context. In Section 4.3, we extend the background from Section 2.2 to twisted Edwards and Montgomery elliptic curves. In Section 4.4, we present a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We complement the work in Section 4.5, with the security checks that a twisted Edwards elliptic curve should pass. We base the latter section on the work from Bernstein and Lange gathered in [BL19]. In Section 4.6, we use our algorithms to generate Baby Jubjub. As we explained in Section 3.4, this curve can be used to implement ECC inside $\mathbb{F}_p$-arithmetic circuits for Ethereum. In fact, Baby Jubjub was accepted as an Ethereum improvement proposal [WBB20] and has already been used in practical applications such as Hermez and Tornado Cash.

In Section 4.7, we include some discussion of the work and future research directions. In Appendix A.1 there is a SAGE implementation of the security checks presented in Section 4.5, which was used to prove that Baby Jubjub was safe under best-known security attacks.

4.2 Related work

The enhance privacy in the blockchain space, there has been an intensive use of cryptographic schemes that make use of elliptic curves [AHG22], which has motivated the appearance of new problems that have been tackled by both industry and academia. For instance, the need for efficient pairing-friendly curves in ZK-SNARK schemes resurfaced the work from Barreto and Naehrig [BN05], and Barreto, Lynn, and Scott [BLS02], who developed techniques to generate pairing-friendly elliptic curves that had an optimal Ate pairing. The vast application of curves derived from their work, which are usually called

BN and BLS curves, resulted in undergoing processes from the IRTF Crypto Forum Research Group to standardize particular instantiations of these curves [KKKK14, YCKS19], such as the BN-256 and the BLS12-381, which are used in digital signature schemes and ZK-SNARK protocols all over the Internet.

The order of these curves is what determines the type of statements we can prove using ZK. More precisely, the largest prime p dividing the curve's order fixes the field in which we can do modular arithmetic. As a result, computational statements that involve elliptic-curve operations can only be proved efficiently with curves that are defined over $\mathbb{F}_p$. Hence, the implementation of ECC schemes that make use of twisted Edwards curves require new curves defined over $\mathbb{F}_p$. In Table 4.1, we summarize the related work.

System	Outer curve	Inner curve
Zcash [zcaa]	BLS12-381	Jubjub
Masson et al. [MSZ21]	BLS12-381	Bandersnatch
Our proposal	BN-256	Baby Jubjub
Ben-Sasson et al. [BSCTV14a]	MNT4	MNT6
Zexe [BCG+20]	CP6-782	BLS12-377
Housni et al. [HG20]	BW6-761	BLS12-377

Table 4.1: Use of elliptic curves in different ZK constructions.

The Zcash team was the first to generate a suitable curve for $\mathbb{F}_p$-arithmetic circuits. Since Zcash ZK-SNARK constructions are based on BLS12-381, their Jubjub elliptic curve was expressly built over the BLS12-381 scalar field. Recently, a new elliptic curve built over the BLS12-381 scalar field was introduced in [MSZ21], but although this curve allows a faster scalar multiplication algorithm than Jubjub, it does not provide any performance improvement in multi-scalar multiplications or in the ZK circuit representations [MSZ21].

We have used a similar approach to generate Baby Jubjub, which is an embedded elliptic curve designed to operate on the field produced by the BN256 elliptic curve. The work presented in this chapter covers the case in which we want to use circuits that can verify public key cryptography primitives such as digital signatures and encryptions in Ethereum. Then, these proofs can be verified by an Ethereum smart contract. We presented and discussed the techniques used to generate Jubjub and Baby Jubjub at the second annual *ZKProof standardization workshop* [WBB19]. This chapter is a reviewed, formalized, and generalized version of the efforts towards the standardization of the generation of suitable twisted Edwards curves for proving arithmetic circuit satisfiability.

The generation of other type of curves for ZK-SNARK circuits has also appeared in other lines of research. For instance, the authors of [BSCTV14a] presented the first practical setting of recursive proof composition with a cycle of two Miyaji–Nakabayashi–Takano (MNT) pairing-friendly elliptic curves [MNT01]. The idea of their proposal is that proofs generated from one curve can feasibly reason about proofs generated from the other curve. To achieve this, one curve's order is the other curve's base field order and vice versa. Although current MNT cycles of curves are quite expensive at the 128-bit security, the work opens the door to the possibility of having succinct blockchains that are verifiable with one single proof.We explore these ideas in further detail in Chapter 5. Bowe et al. [BCG+20] proposed ZEXE, a construction that follows a relatively relaxed approach to find a suitable pair of curves that form a chain rather than a cycle. A later work from El Housni and Guillevic [HG20] improved ZEXE with a new curve that makes the verification of composed ZK-SNARK proofs significantly faster.

4.3 Elliptic curves

Both the affine and the projective short Weierstrass forms presented in Section 2.2 are the most general ways to describe elliptic curves over finite fields $\mathbb{F}_p$ with $p > 3$. However, in certain situations it might be advantageous to consider more specialized representations of elliptic curves for example to get faster algorithms for the group law or the scalar multiplication. In this section, we review Montgomery and twisted Edwards elliptic curves and over finite fields. We assume $p > 3$ and denote $\mathbb{F}_p$ the finite field of order p. We follow [BBJ+08,OKS00].

4.3.1 Montgomery curves

Montgomery curves are those whose equation can be written in Montgomery form. This curves allow for constant time algorithms for the elliptic curve scalar multiplication [OKS00].

Definition 4.1 (Montgomery curve). Let $A \in \mathbb{F}_p \backslash \{-2, 2\}$ and $B \in \mathbb{F}_p \backslash \{0\}$. An elliptic curve defined by

$$E^M : By^2 = x^3 + Ax^2 + x$$

is called a *Montgomery (elliptic) curve.*

Although Montgomery curves look different from short Weierstrass curves, they are just a special way to describe certain short Weierstrass curves. In fact, every curve in affine Montgomery form can be transformed into an elliptic curve in short Weierstrass form. The set of ($\mathbb{F}_p$-rational) points of E^M forms a group with the point at infinity O as the identity element. The rules of the following theorem are a complete description of the elliptic curve group law [LM87].

Definition 4.2 (Montgomery group law). Let E^M be a Montgomery curve. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E^M with $P_1, P_2 \neq \mathcal{O}$. We define $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

- If $x_1 \neq x_2$, then

$$\begin{aligned} \Lambda &= (y_2 - y_1)/(x_2 - x_1), \\ x_3 &= B\Lambda^2 - A - x_1 - x_2, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{4.1}$$

- If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = O$.
- If $P_1 = P_2$ and $y_1 \neq 0$, then

$$\begin{aligned} \Lambda &= (3x_1^2 + 2Ax_1 + 1)/(2By_1), \\ x_3 &= B\Lambda^2 - A - 2x_1, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{4.2}$$

- If $P_1 = P_2$ and $y_1 = 0$, then $P_1 + P_2 = O$.

Moreover, we define $P + O = P$ for all points P on E^M. Note that the inverse of O is O, and the inverse of any other point $P = (x, y)$ is $(x, -y)$.

A result due to [LM87], is that the order of a Montgomery curve is always divisible by 4.

4.3.2 Twisted Edwards curves

Edwards in [Edw07] introduced an addition law for the curves $x^2 + y^2 = c^2(1 + x^2y^2)$ over a non-binary field, and later, Bernstein and Lange [BBJ+08] generalized the addition law to the curves $x^2 + y^2 = 1 + dx^2y^2$. This form covers considerably more elliptic curves over a finite field than the Edwards form.

Definition 4.3 (Twisted Edwards curve). Let $a, b \in \mathbb{F}_p \backslash \{0\}$ with $a \neq b$. An elliptic curve defined by

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

is called a *twisted Edwards (elliptic) curve.*

In contrast to Montgomery and short Weierstrass curves, twisted Edwards curves have complete addition formulas. This is very convenient when writing the arithmetic as a circuit, because we can express it as a single computation without the need of branching the operation to each case.

Definition 4.4 (Twisted Edwards group law). Let E be a twisted Edwards curve. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be any points on E. We define the addition $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

$$\begin{aligned} \lambda &= dx_1x_2y_1y_2, \\ x_3 &= (x_1y_2 + y_1x_2)/(1 + \lambda), \\ y_3 &= (y_1y_2 - x_1x_2)/(1 - \lambda). \end{aligned}$$

Note that the inverse of any point $P = (x, y)$ is $(-x, y)$.

Montgomery and twisted Edwards curves are birationally equivalent. The following theorem gives the explicit transformation from one form to the other.

Theorem 4.1 ([BBJ+08, Theorem 3.2]). Every twisted Edwards curve $E/\mathbb{F}_p$ is birationally equivalent over $\mathbb{F}_p$ to a Montgomery curve $E^M/\mathbb{F}_p$ with parameters

$$A = 2\,\frac{a+d}{a-d} \quad \text{and} \quad B = \frac{4}{a-d}.$$

The birational equivalence from E to E^M is the map

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

with inverse

$$(u, v) \to (x, y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right). \tag{4.3}$$

Conversely, every Montgomery curve $E^M/\mathbb{F}_p$ is birationally equivalent over $\mathbb{F}_p$ to a twisted Edwards curve $E/\mathbb{F}_p$ with parameters

$$a = \frac{A+2}{B} \quad \text{and} \quad d = \frac{A-2}{B}.$$

4.4 Generation of twisted Edwards curves

In this section, we present a deterministic method that, given a prime number p, it outputs a twisted Edwards curve E defined over $\mathbb{F}_p$.

4.4.1 General overview

Our algorithm takes a prime number p and returns a twisted Edwards curve defined over $\mathbb{F}_p$. More precisely, the specific outputs of the algorithm are:

- The prime order of the finite field the curve is defined over, which is the input p.
- Parameters a and d of the equation that defines the twisted Edwards curve.
- The order of the curve and its decomposition into the product of a cofactor h and a large prime q.
- A generator and a base point for the curve.

Since the finite field is defined by the input p, no specification of this parameter is required. The order of the curve and its decomposition are also determined once the parameters of the equation describing the curve are fixed. Hence, the only remaining specifications are parameters a and d, and the choice of generator and base point. We have divided the procedure in four steps:

1. *Choice of Montgomery equation* (Section 4.4.2): we start by deterministically generating a Montgomery elliptic curve E^M over $\mathbb{F}_p$.
2. *Choice of generator and base points* (Section 4.4.3): we set the generator and base points of E^M.
3. *Transformation to twisted Edwards* (Section 4.4.5): we convert the curve E^M to its birationally equivalent twisted Edwards form and the generator and base points using the maps from Theorem 4.1.
4. *Optimization of parameters* (Section 4.4.4): if possible, we rescale all parameters so that the arithmetic in the curve can be sped up [HWCD08].

All algorithms presented in this section have been implemented in SAGE programming language, and are presented in Section 4.6.

4.4.2 Choice of Montgomery equation

We start by finding a Montgomery curve defined over $\mathbb{F}_p$, where p is a given prime number. The assumptions and algorithm presented are based on the work of [LHT16] and Zcash team [Bow19]. Algorithm 1 takes a prime p, fixes $B = 1$ and returns the Montgomery elliptic curve defined over $\mathbb{F}_p$ with the smallest coefficient A such that $A - 2$ is a multiple of 4. This approach comes from the fact that, when defining a Montgomery curve, the smaller A is, the faster the group operation becomes. More precisely, as pointed out in [BL17], for the best performance, we need $(A - 2)/4$ to be small. As with $A = 1$ and $A = 2$, a twisted Edwards equation does not describe a smooth curve, so the algorithm starts with $A = 3$.

Algorithm 1 Generation of E^M

Input: prime number p
Output: coefficients A, B, order n, cofactor h, prime q
fix $B = 1$
start with $A = 3$
if $(A - 2) = 0 \bmod 4$ **then**
    continue
else
    increment A by 1 and go back to line 3
if equation $y^2 = x^3 + Ax^2 + x$ defines an elliptic curve over $\mathbb{F}_p$ **then**
    continue
else
    increment A by 1 and go back to line 3
compute the group order n and cofactor h
if $p = 1 \bmod 4$ **then**
    if (cofactor is 8 **and** cofactor of twist is 4) **then**
        set $h = 8$
    else
        increment A by 1 and go back to line 3
if $p = 3 \bmod 4$ **then**
    if (cofactor **and** cofactor of twist is 4) **then**
        set $h = 4$
    else
        increment A by 1 and go back to line 3
compute $q = n/h$
return A, B, n, h, and q

For primes congruent to 1 modulo 4, the minimal cofactors of the curve and its twist are either $\{4, 8\}$ or $\{8, 4\}$. We choose a curve with the latter cofactors, so that any algorithms that take the cofactor into account do not have to worry about checking for points on the twist, because the twist cofactor will be the smaller of the two [LHT16]. For a prime congruent to 3 modulo 4, both the curve and twist cofactors can be 4, and this is minimal.

4.4.3 Choice of generator and base points

To pick a generator G_0^M of the curve, we choose the smallest element of $\mathbb{F}_p$ that corresponds to an x-coordinate of a point in the curve of order n. Then, as a base point, we define $G_1^M = 8G_0^M$, which has order q. The steps are written down in Algorithm 2.

Algorithm 2 Generator and base points of E^M

Input: Montgomery curve E^M, order n, cofactor h
Output: generator G_0^M, base point G_1^M
start with $u = 1$
find v such that (u, v) is a point of E^M. **else**, increment u by 1 and repeat
check that (u, v) has order n. **else**, increment u by 1 and go back to step 2
set $G_0^M = (u, v)$ and $G_1^M = hG_0$
return G_0^M and G_1^M

4.4.4 Transformation to twisted Edwards

In Algorithm 3, we use the birational map from Equation (4.3) to get the coefficients, generator, and base points in the twisted Edwards form.

Algorithm 3 Conversion of E^M to E

Input: Montgomery coefficients A, B, generator $G_0^M = (x_0^M, y_0^M)$, base point $G_1^M = (x_1^M, y_1^M)$
Output: twisted Edwards coefficients a, d, generator G_0, base point G_1
compute $a = (A + 2)/B$ and $d = (A - 2)/B$
compute $x_0 = x_0^M/y_0^M$ and $y_0 = (x_0^M - 1)/(x_0^M + 1)$
compute $x_1 = x_1^M/y_1^M$ and $y_1 = (x_1^M - 1)/(x_1^M + 1)$
set $G_0 = (x_0, y_0)$ and $G_1 = (x_1, y_1)$
return a, d, G_0 and G_1

4.4.5 Optimization of parameters

As pointed out in [HWCD08, Section 3.1], if $-a$ is a square in $\mathbb{F}_p$, it is possible to optimize the number of operations in a twisted Edwards curve by scaling it. The result follows directly from the map's definition.

Theorem 4.2. Consider a twisted Edwards elliptic curve defined over $\mathbb{F}_p$ given by equation $ax^2 + y^2 = 1 + dx^2y^2$. If $-a$ is a square in $\mathbb{F}_p$, then the map $(x, y) \to (x/\sqrt{-a}, y)$ defines the curve $-x^2 + y^2 = 1 + (-d/a)x^2y^2$. We call $f = \sqrt{-a}$ the *scaling factor*.

Algorithm 4 rescales, if possible, the twisted Edwards curve found in the previous step as described in Theorem 4.2. It also converts the generator and base points to the new coordinates. After applying the algorithm, the map that transforms E^M to E is the composition of maps from Theorems 4.1 and 4.2.

Algorithm 4 Rescaling of E with $a = -1$ (if possible)

Input: coefficients a, d, generator $G_0 = (x_0, y_0)$, base point $G_1 = (x_1, y_1)$
Output: scaling factor f, coefficients $a' = a/f^2$, $d' = -d/a$, generator $G'_0 = (x_0/f, y_0)$, base point $G'_1 = (x_1/f, y_1)$
if $-a$ is a square in $\mathbb{F}_p$ **then**
    take $f = \sqrt{-a}$
    set $a' = -1$ and $d' = -d/a$
    compute $x'_0 = x_0/f$ and $x'_1 = x_1/f$
    set $G'_0 = (x'_0, y_0)$ and $G'_1 = (x'_1, y_1)$
    return f, a', d', G'_0 and G'_1
else
    set $f = 1$
    return f, a, d, G_0 and G_1

4.5 Security tests

This section specifies the safety criteria that an elliptic curve found in the previous section should satisfy. The choices of security parameters are based on the joint work of Bernstein and Lange summarized in [BL19]. In Appendix A.1, we provide an implementation of the algorithm that should be run after finding the elliptic curve as proposed in the previous section. The algorithm is based on the code from [Hop17], which is an extension of the original SAGE code from [BL19], to general twisted Edwards curves.

Curve parameters

We check that all given parameters describe a well-defined elliptic curve over a prime finite field:

- The given number p is prime.
- The given parameters define an equation that corresponds to an elliptic curve.
- The product of h and q results into the order of the curve and the point G_0 is a generator.
- The given number q is prime and the point G_1 is a base point.

Elliptic-curve discrete logarithm problem

We check that the discrete logarithm problem remains difficult in the given curve. For that, we check the curve is resistant to the following known attacks:

- *Rho method* (Section V.1, [BSS99]): we require the cost for the rho method, which takes on average around $0.886\sqrt{q}$ additions, to be above 2^{100}.
- *Additive and multiplicative transfers* (Section V.2, [BSS99]): we require the embedding degree to be at least $(q-1)/100$.
- *High discriminant* (Section IX.3, [BSS99]): we require the complex-multiplication field discriminant D to be larger than 2^{100}.

Elliptic-curve cryptography

We check if the curve is suitable for ECC:

- *Ladders* [LM87]: check the curve supports the Montgomery ladder.
- *Twists* (twist, [BL19]): check if it is secure against the small-subgroup attack, invalid-curve attacks and twisted-attacks.
- *Completeness* (complete, [BL19]): check if the curve has complete single-scalar and multiple-scalar formulas. It is enough to check that there is only one point of order 2 and two points of order 4.
- *Indistinguishability* [BHKL13]: check availability of maps that turn elliptic-curve points indistinguishable from uniform random strings.

4.6 Baby Jubjub: a suitable curve for Ethereum

Ethereum, the second-largest blockchain, uses BN256 to generate and verify ZK-SNARK proofs. BN256 is a pairing-friendly elliptic curve of prime order

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.$$

In order to prove ECC statements with ZK-SNARKs, Ethereum needed a new curve defined over $\mathbb{F}_p$. In this section, we present a SAGE implementation of the algorithms presented in the previous sections, and we use them to generate *Baby Jubjub*, a twisted Edwards elliptic curve suitable for ZK-SNARK circuits in Ethereum.

In Listing 4.1, we implemented Algorithm 1, which generates a Montgomery curve E^M with the smallest A satisfying the conditions from Section 4.4.2.

```
def findCurve(prime, curveCofactor, twistCofactor, _A):
  Fp = GF(prime)
  A = _A
  while A < _A + 200000:
    if (A-2.) % 4 != 0:
      A+=1.
      continue
    try:
      E = EllipticCurve(Fp, [0, A, 0, 1, 0])
    except:
      A+=1.
      continue

    groupOrder = E.order()
    if (groupOrder % curveCofactor != 0 or not is_prime(groupOrder //
     curveCofactor)):
      A+=1
      continue

    twistOrder = 2 * (prime+1)-groupOrder
    if (twistOrder % twistCofactor != 0 or not is_prime(twistOrder //
     twistCofactor)):
      A+=1
      continue

    return E, A, 1, groupOrder, curveCofactor, groupOrder //
     curveCofactor

def find1Mod4(prime, curveCofactor, twistCofactor, A):
  assert((prime % 4) == 1)
  return findCurve(prime, curveCofactor, twistCofactor, A)
```

Listing 4.1: Generation of E^M.

In the following lines of code, we instantiate the functions from Listing 4.1 using the prime number p, which is the order of BN256 curve, and enforce the resulting curve to have cofactor $h = 8$.

```
# Baby Jubjub in Montgomery form
prime = 21888242871839275222246405745257275088548364400416034343698204186575808495617
Fp = GF(prime)
h = 8
A = 1.
EC, A, B, n, h, q = find1Mod4(prime, h, 4, A)
```

The result is that the smallest A satisfying the conditions is $A = 168698$. As a result, the Montgomery form $E^M/\mathbb{F}_p$ of Baby Jubjub is defined by equation:

$$y^2 = x^3 + 168698x^2 + x.$$

The function `findCurve` also returns the order of the curve, which is

$$n = 21888242871839275222246405745257275088614511777268538073601725287587578984328,$$

where $n = h \times q$, with $h = 8$, and q is the large prime number

$$q = 2736030358979909402780800718157159386076813972158567259200215660948447373041.$$

To get a generator and a base point for E^M deterministically, we implemented Algorithm 2 in Listing 4.2, which returns as generator G_0^M the point of the curve of order n with smallest x-coefficient, and $G_1^M = hG_0^M$ as base point.

```
def findGenPoint(prime, A, EC, N):
  Fp = GF(prime)
  for uInt in range(1, 1e3):
    u = Fp(uInt)
    v2 = u^3 + A * u^2 + u
    if not v2.is_square():
      continue
    v = v2.sqrt()

    point = EC(u, v)
    pointOrder = point.order()
    if pointOrder == N:
      return point

def findBasePoint(EC, h, u, v):
  return h * EC(u, v)
```

Listing 4.2: Generator G_0 and base point G_1 of E^M.

The next couple of lines of code are used to find a valid generator and base point for Baby Jubjub in Montgomery form.

```
# Generator and base points of Baby Jubjub in Montgomery form
gen_u, gen_v, gen_w = findGenPoint(prime, A, EC, n)
base_u , base_v, base_w = findBasePoint(EC, h, gen_u, gen_v)
```

The outputs are the generator $G_0^M = (x_0^M, y_0^M)$ with coordinates

$$\begin{aligned} x_0^M &= 7, \\ y_0^M &= 42587277738759406903626075504983045981 \\ &\quad 01071202821725296872974770776423442226, \end{aligned}$$

and the base point $G_1^M = (x_1^M, y_1^M)$ with coordinates

$$\begin{aligned} x_1^M &= 71179280504075836181111764215552147566 \\ &\quad 75765419608405867398403713213306743542, \\ y_1^M &= 14577268218881899420966779687690205425 2 \\ &\quad 27431577728659819975198491127179315626. \end{aligned}$$

Algorithm 3 maps a Montgomery curve to its twisted Edwards form. We divided the algorithm in three different functions. The first function `mont_to_ted` converts a Montgomery point to a twisted Edwards point, the second function `ted_to_mont` does the opposite, and finally `is_on_ted` checks if a point is a solution to a given twisted Edwards equation. Although the last two functions are not needed in the original algorithm, we implemented them in order to have sanity checks after the conversion from Montgomery to twisted Edwards form.

```
def mont_to_ted(u, v , prime):
  Fp = GF(prime)
  x = Fp(u / v)
  y = Fp((u-1)/(u+1))
  return(x, y)

def ted_to_mont(x, y , prime):
  Fp = GF(prime)
  u = Fp((1 + y )/ ( 1 - y))
  v = Fp((1 + y ) / ( (1 - y)  * x))
  return(u, v)

def is_on_ted(x, y, prime, a, d):
  Fp = GF(prime)
  return Fp(a * (x * * 2) + y * * 2 - 1 - d * (x * * 2) * (y * * 2)) == 0
```

Listing 4.3: Conversion of E^M to E.

We run these functions to get the twisted Edwards form E of Baby Jubjub.

```
# Conversion of Baby Jubjub to twisted Edwards
a = Fp((A + 2) / B)
d = Fp((A - 2) / B)

# Check we have a safe twist and discriminant != 0
assert(not d.is_square())
assert(a * d * (a-d)!=0)

# Conversion of generator to twisted Edwards
gen_x, gen_y = mont_to_ted(gen_u, gen_v, prime)
assert(is_on_ted(gen_x, gen_y, prime, a , d))

# Sanity check: the inverse map returns the original point in Montgomery
u , v = ted_to_mont(gen_x, gen_y, prime)
assert (u == gen_u)
assert (v == gen_v)

# Conversion of base point to twisted Edwards
base_x , base_y = mont_to_ted(base_u, base_v, prime)
assert(is_on_ted(base_x,base_y, prime , a , d))
```

After the conversion maps, we get that E is described by equation

$$168700x^2 + y^2 = 1 + 168696x^2y^2.$$

The code from Listing 4.3 also outputs the generator $G_0 = (x_0, y_0)$ and base point $G_1 = (x_1, y_1)$ in twisted Edwards form. The specific outputs are that G_0 has coordinates

$$\begin{aligned} x_0 = {} & 995203441582195749578291179787384436505546430278305826713579947235728471134, \\ y_0 = {} & 547206071795981880556160143631431877213709110010400858592455104664395212390 5, \end{aligned}$$

and G_1 has coordinates

$$\begin{aligned} x_1 = {} & 529961924064155128163486558351829703028287447219077289408652114448272100155 3, \\ y_1 = {} & 169501507984606577179586255678218345503016631616247077872228159361826389682 03. \end{aligned}$$

Finally, the last Algorithm 4 tries to escalate the twisted Edwards form of the curve, so that the equation has parameter $a = -1$. This last step is implemented in Listing 4.4.

```
def scaling(a, d, prime):
  Fp = GF(prime)
  if Fp(-a).is_square():
    f = sqrt(Fp(-a));
    a_ = Fp(a / (f * f));
    d_ = Fp(d / (-a));
    if a_ == Fp(-1):
      a_ = -1
    else:
      a_ = a;
      d_ = a;
  return a_, d_, f

def ted_to_tedprime(x, y, prime, scaling_factor):
  Fp = GF(prime)
  x_ = Fp(x  *  (-scaling_factor))
  y_ = y;
  return(x_, y_)

def tedprime_to_ted(x_, y_, prime, scaling_factor):
  Fp = GF(prime)
  x = Fp(x_ / (-scaling_factor))
  y = y_
  return(x, y)

def is_on_ted_prime(x, y, prime, a_, d_):
  Fp = GF(prime)
  return Fp(a_ * (x * * 2) + y * * 2 - 1 - d_ * (x * * 2) * (y * * 2)) == 0
```

Listing 4.4: Scaling of E to E'.

We call these functions to get E escalated to E'.

```
# Conversion of E to E'
a_, d_, f = scaling (a, d, prime)

# Conversion of generator to E'
gen_x_, gen_y_ = ted_to_tedprime(gen_x, gen_y, prime, f);
assert(is_on_ted_prime(gen_x_, gen_y_, prime, a_ , d_))

# Sanity check: the inverse map returns the original point in E
u , v = tedprime_to_ted(gen_x_prime, gen_y_prime, prime, f)
assert (u == gen_x)
assert (v == gen_y)

# Conversion of base point to E'
base_x_, base_y_ = ted_to_tedprime(base_x, base_y, prime, f);
assert(is_on_ted_prime(base_x_,base_y_, prime , a_ , d_))

# Sanity check: the inverse map returns the original point in E
u , v = tedprime_to_ted(base_x_prime, base_y_prime, prime, f)
assert (u == base_x)
assert (v == base_y)
```

The resulting scaling factor is

$$f = 19119828543052250743812513441033299316$$
$$37610209014896889891168275855466657090.$$

This way, the optimal version of Baby Jubjub in twisted Edwards form is given by equation

$$-x^2 + y^2 = 1 + d'x^2y^2,$$

where

$$d' = 12181644023421730124874158521699555681 7$$
$$64249180949974110617291017600649128846.$$

After generating Baby Jubjub, we checked that the curve passed all safety checks described in Section 4.5. The security evidence is shown in [Whi18]. The determinism and transparency of the procedure allows any party to reproduce the generation of the curve and ensure its resilience against best-known security attacks [ST16].

4.6.1 Elliptic-curve arithmetic

In this section, we present a circuit description of the scalar multiplication on Baby Jubjub. Before, we recall from Section 4.3, that Montgomery curves have group laws where many cases have to be distinguished. Cases are translated to branches in a circuit, and as we pointed out in Section 3.2.4, circuit branches are undesirably costly. Twisted Edwards curves offer an advantage in this regard, as doubling can be carried out using the same formula as addition, which eliminates the need for such case distinctions. In contrast, operating in Montgomery curves offers cost efficiency. For a detailed breakdown of the precise number of operations required in different forms of elliptic curves, see [BBJ+08].

In this section, we describe a circuit that performs a scalar multiplication on a point on Baby Jubjub. The circuit combines both forms Montgomery and twisted Edwards of the curve to take advantage of the closed formula of the former and the efficiency of the latter.

Scalar multiplication

Let $P \neq O$ be a point of the curve E of order strictly greater than 8, and let k a binary number representing an element of $\mathbb{F}_p$. We describe the circuit used to compute the point kP.

First, we divide k into chunks of 248 bits. If k is not a multiple of 248, we take j segments of 248 bits and leave a last chunk with the remaining bits. More precisely, we write

$$k = k_0 k_1 \dots k_j$$

with

$$\begin{cases} k_i = b_0^i b_1^i \dots b_{247}^i & \text{for } i = 0, \dots, j-1, \\ k_j = b_0^j b_1^j \dots b_s^j & \text{with } s \leq 247. \end{cases}$$

Then,

$$kP = k_0 P + k_1 2^{248} P + \dots + k_j 2^{248j} P. \tag{4.4}$$

In Figure 4.1 we describe de circuit that performs this sum. The idea is that each term of Equation (4.4) is calculated separately inside an SEQ box and then all terms added together. The computations performed in the SEQ boxes described in Figures 4.2–4.4.

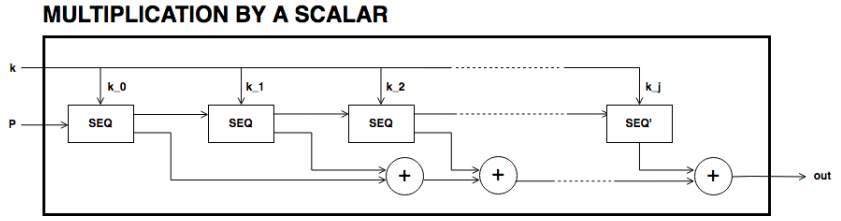

Figure 4.1: Description of the arithmetic circuit that performs the multiplication of a scalar k by a point P on Baby Jubjub elliptic curve. The SEQ and SEQ' boxes are described in more detail in Figures 4.2–4.4.

First note that if we denote $P_i = 2^{248i} P$ for $i = 0, \dots, j-1$, then Equation (4.4) can be written as

$$kP = k_0 P_0 + k_1 P_1 + \dots + k_j P_j. \tag{4.5}$$

Now, the SEQ box takes as inputs a point P_i of E and a segment k_i, and outputs two points:

$$P_{i+1} = 2^{248i} P_i \quad \text{and} \quad \sum_{n=0}^{247} b_n^i 2^n P_i.$$

The first output point is the input of the next $(i+1)$-th SEQ box, and the second output is the computation of the i-th term in Equation (4.5). The idea of the circuit, which is depicted in Figure 4.2 is to first compute

$$Q_i = P_i + b_1(2P_i) + b_2(4P_i) + b_3(8P_i) + \cdots + b_{247}(2^{247}P_i),$$

and output the point $Q_i - b_0P_i$. This trick allows the computation of Q_i using the Montgomery form of Baby Jubjub and only use the twisted Edwards form for the second calculation.

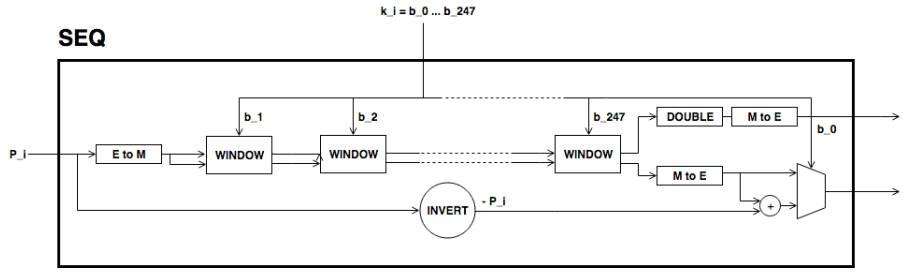

Figure 4.2: Description of the SEQ circuit that is part of the circuit from Figure 4.1. The WINDOW box is described in more detail in Figure 4.3.

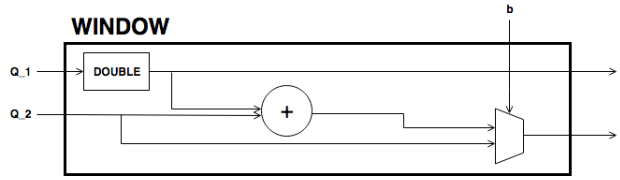

Figure 4.3: Description of the WINDOW circuit that is part of Figures 4.2 and 4.4.

Finally, the last term of Equation (4.5) is computed in a very similar manner. The difference is that the number of bits composing k_j may be shorter and that there is no need to compute P_{j+1}, as there is no other SEQ box after this one. So, there is only output, the point $k_jP_j = k_j2^{248j}P$. This circuit is named SEQ' and is depicted in Figure 4.4.

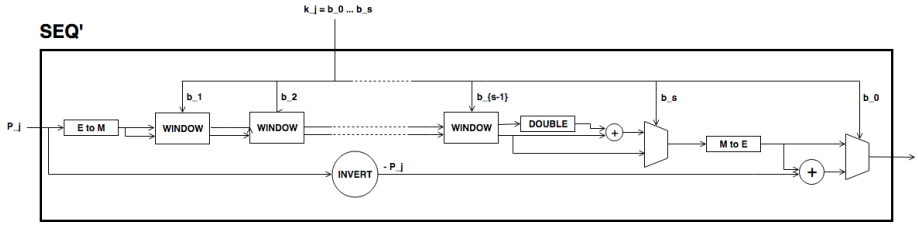

Figure 4.4: Description of the SEQ' circuit that is part of the circuit from Figure 4.1. The WINDOW box is described in more detail in Figure 4.3.

Security and efficiency considerations

The reason to change from Montgomery to twisted Edwards form in the SEQ and SEQ' circuits in the calculation of the output, is that if $b_0 = 0$, then the computation may become a sum with input the point at infinity. For this reason, by switch back to twisted Edwards and use a single formula for computing $b_0 P_i$.

For the rest, we use the addition and doubling formulas in Montgomery form because are more efficient, but we have to ensure that none of the points being doubled or added when working with E^M is the point at infinity and, also that we never add a point to itself. Let's show that this is not the case:

- By assumption, $P \neq O$ and $\text{ord}(P) > 8$. Hence, by Lagrange theorem [BC68, Corollary 4.12], P must have order q, $2q$, $4q$ or $8q$. For this reason, none of the points in E_M being doubled or added in the circuit is the point at infinity, because for any integer m, 2^m is never a multiple of r, even when 2^m is larger than q, as q is a prime number. Hence, $2^m P \neq O$ for any $m \in \mathbb{Z}$.

- Looking closely at the two inputs of the sum, it is easy to realize that they have different parity, one is an even multiple of P_i and the other an odd multiple of P_i, so they must be different points. Hence, the sum in E^M is done as expected.

4.6.2 The Bowe–Hopewood–Pedersen hash

The Bowe–Hopwood–Pedersen hash is al algebraic hash function that was introduced by Zcash in the Sapling network upgrade. The precise definition was first introduced in [HBHW19, Section 5.4.1.7] and it is based on the works

[CDG87, CvHP92, BGG94] with optimizations for efficient instantiation in ZK-SNARK circuits by Sean Bowe and Daira Hopwood. The Bowe–Hopwood–Pedersen hash maps a sequence of bits to a compressed point on a twisted Edwards elliptic curve [LMS17]. We adapt the original definition from Jubjub to Baby Jubjub using 4-bit windows.

Definition 4.5 (Bowe–Hopwood–Pedersen hash function). Let E by Baby Jubjub elliptic curve in twisted Edwards form and let $\mathbb{G}$ denote the large subgroup of E of prime order q. For a fixed $k \in \mathbb{Z}$, let $P_0, P_1, \ldots, P_k$ be uniformly sampled generators of $\mathbb{G}$. We define the *Bowe–Hopwood–Pedersen hash function* $H : \{0,1\}^* \to \mathbb{G}$ for a binary input of arbitrary length M the following way. Split M into sequences of at most 200 bits and each of those into chunks of 4 bits[1]. More precisely, write $M = M_0 M_1 \ldots M_l$ where

$$M_i = m_0 m_1 \ldots m_{k_i} \quad \text{with} \quad \begin{cases} k_i = 49 & \text{for } i = 0, \ldots, l-1, \\ k_i \leq 49 & \text{for } i = l, \end{cases}$$

where the m_j terms are chunks of 4 bits $[b_0\ b_1\ b_2\ b_3]$. Define

$$enc(m_j) = (1 - 2b_3)(1 + b_0 + 2b_1 + 4b_2)$$

and let

$$\langle M_i \rangle = \sum_{j=0}^{k_i - 1} enc(m_j) 2^{5j}.$$

Then, $H(M)$ is defined as

$$H(M) = \langle M_0 \rangle P_0 + \langle M_1 \rangle P_1 + \langle M_2 \rangle P_2 + \cdots + \langle M_l \rangle P_l. \tag{4.6}$$

Note that Equation (4.6) is linear combination of elements of $\mathbb{G}$, so $H(M)$ is a point of E of order q.

Circuit description

We assume that for a fixed k, the points $P_0, \ldots, P_k$ are known uniformly sampled generators of $\mathbb{G}$. As suggested in [HBHW19], this can be done by taking a string `D = "string_seed"` followed by a byte `S` holding that smallest number such that `H = Keccak256(D || S)` results in a point in the elliptic curve E.

[1] If M is not a multiple of 4, pad M to a multiple of 4 bits by appending zero bits.

In Figure 4.5 we depicted the circuit used to compute the Pedersen hash of a message M as described in Definition 4.5. Each MULTIPLICATION box in the circuit, returns a term of the sum from Equation (4.6). The logic of MULTIPLICATION is depicted in Figure 4.6,

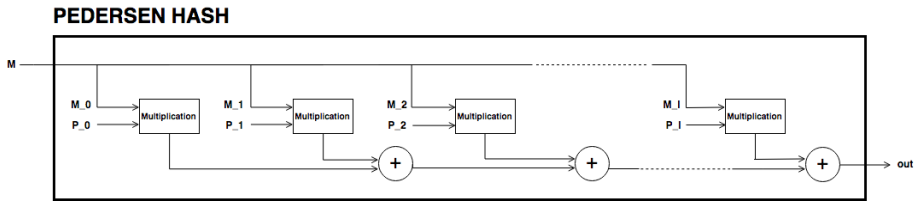

Figure 4.5: Description of the arithmetic circuit that performs the Bowe–Hopwood–Pedersen hash on a binary input message M. The MULTIPLICATION box is described in more detail in Figure 4.6.

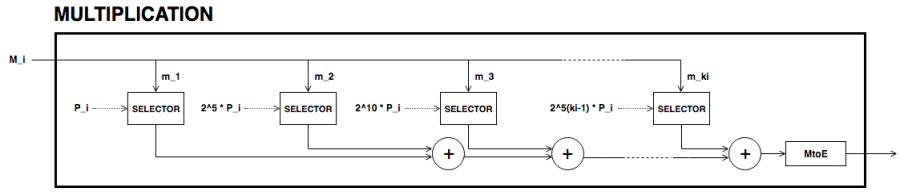

Figure 4.6: Description of the MULTIPLICATION circuit that is part of the circuit from Figure 4.5. The SELECTOR box is described in more detail in Figure 4.7.

As the set of generators $P_0, \ldots, P_k$ are fixed, we can precompute its multiples and use 4-bit lookup windows to select the right points. This is done as depicted in Figure 4.7. This circuit receives a 4-bit chunk input and returns a point. The first three bits are used to select the right multiple of the point and last bit decides the sign of the point. The sign determines if the x-coordinate should be taken positive or negative. As we pointed out in Definition 4.4, negating a point in twisted Edwards form corresponds to the negation of its first coordinate.

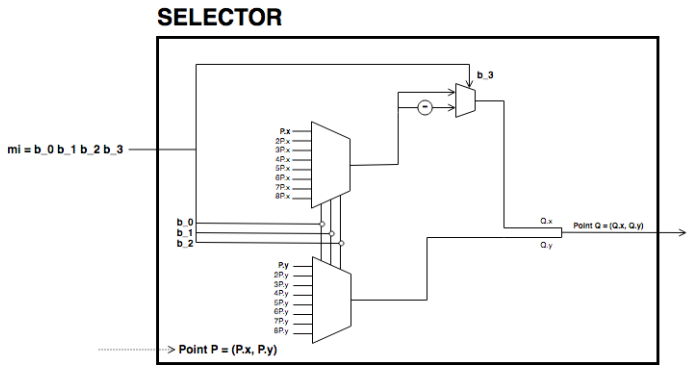

Figure 4.7: Description of the SELECTOR circuit that is part of Figure 4.6.

Security considerations

In our circuit description, we use a windowed scalar multiplication algorithm with signed digits. Each 4-bit message chunk corresponds to a window called SELECTOR and each chunk is encoded as an integer from the set $\{-8..8\}\backslash\{0\}$. This allows a more efficient lookup of the window entry for each chunk than if the set $\{1..16\}$ had been used, because a point can be conditionally negated using only a single constraint [HBHW19]. As there are up to 50 segments per each generator P_i, the largest multiple of the generator P_i is nP_i with

$$n = 2^0 \times 8 + 2^5 \times 8 + \left(2^5\right)^2 \times 8 \cdots + 2^{245} \times 8.$$

To avoid overflow, this number should be smaller than $(q-1)/2$. Indeed,

$$\begin{aligned} n &= 8 \times \sum_{k=0}^{49} 2^{5k} = 8 \times \frac{2^{250}-1}{2^5-1} \\ &= 466903585634339497675689455680193176827701551071131306610716064 \\ &\quad 548036813064, \end{aligned}$$

and,

$$\begin{aligned} \frac{q-1}{2} &= 136801517948995470139040035907857969303840698607928362960010 \\ &\quad 07830474223686520 \\ &> n. \end{aligned}$$

Efficiency discussion

We look at the number of constraints per bit of the input in our circuit implementation. We are using 4-bit windows in the Montgomery form of the curve, so we have 1 constraint for the sign, and 3 constraints for the sum. Now, let's look at the constraints required in the multiplexers from the circuit in Figure 4.7. Standard 4-bit windows require 2 constraints, one for the output and another to compute the product $s_0 s_1$. So, without optimizations, we would need a total of 4 constraints, 2 per multiplexer. However, we can reduce it to 3 by reusing the computation $s_0 s_1$, which is the same in both multiplexers. Let the multiplexers of coordinates x and y be represented by the following look-up tables:

multiplexer for x			
s_2	s_1	s_0	out
0	0	0	a_0
0	0	1	a_1
0	1	0	a_2
0	1	1	a_3
1	0	0	a_4
1	0	1	a_5
1	1	0	a_6
1	1	1	a_7

multiplexer for y			
s_2	s_1	s_0	out
0	0	0	b_0
0	0	1	b_1
0	1	0	b_2
0	1	1	b_3
1	0	0	b_4
1	0	1	b_5
1	1	0	b_6
1	1	1	b_7

We can express these tables using these 3 constraints:

$$\textit{(1)}\ aux = s_0 s_1,$$

$$\textit{(2)}\ out = \Big[(a_7 - a_6 - a_5 + a_4 - a_3 + a_2 + a_1 - a_0)aux + (a_6 - a_4 - a_2 + a_0)s_1 + (a_5 - a_4 - a_1 + a_0)s_0 + (a_4 - a_0)\Big] z + (a_3 - a_2 - a_1 + a_0)aux + (a_2 - a_0)s_1 + (a_1 - a_0)s_0 + a_0,$$

$$\textit{(3)}\ out = \Big[(b_7 - b_6 - b_5 + b_4 - b_3 + b_2 + b_1 - b_0)aux + (b_6 - b_4 - b_2 + b_0)s_1 + (b_5 - b_4 - b_1 + b_0)s_0 + (b_4 - b_0)\Big] z + (b_3 - b_2 - b_1 + b_0)aux + (b_2 - b_0)s_1 + (b_1 - b_0)s_0 + b_0.$$

As a result, the amount of constraints per bit using 4-lookup windows is $(1 + 3 + 3)/4 = 1.75$. Indeed, our implementation of the Bowe–Hopwood–Pedersen hash in CIRCOMLIB [Ide20b] for a binary input of 256 bits consists of 452 constraints, roughly 256×1.75. The extra 4 constraints come from CIRCOM templates connections (see Section 3.5).

4.7 Conclusions

In the recent years, ZK proofs arose as a potential solution to blockchain privacy and scalability issues and, today, we can see many zero-knowledge protocols integrated and deployed in various blockchain projects. The use of cryptography in the blockchain space arises new efficiency needs that motivate novel lines of research that combine theoretical and practical aspects. In particular, the recent implementation of ZK protocols has had a huge impact in the interest for generating types of curves with special properties. The correct and transparent generation of new elliptic curves is paramount to the success of cryptographic primitives that can help blockchains improve their privacy and scalability guarantees. In this chapter, we presented a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We also provided an algorithm for checking the safety of a curve against best known security attacks. Additionally, we gave an example that puts theory into practice: we detailed the generation of the twisted Edwards curve Baby Jubjub and present specific circuits for performing elliptic arithmetic and a hash function based on this curve.

Chapter 5

Revisiting cycles of pairing-friendly elliptic curves

It is possible to write endlessly on elliptic curves.

(This is not a threat.)

– Serge Lang

5.1 Introduction

A recent area of cryptographic interest is *recursive composition* of proof systems [Val08, BSCTV17], since it leads to *proof-carrying data (PCD)* [CT10], a cryptographic primitive that allows multiple untrusted parties to collaborate on a computation that runs indefinitely, and has found multiple applications [CTV15, NT16, KB20, BMRS20]. In recursive composition of proof systems, each prover in a sequence of provers takes the previous proof and verifies it, and performs some computations on their own, finally producing a proof that guarantees that (a) the previous proof verifies correctly, and (b) the new computation has been performed correctly. This way, the verifier, who simply verifies the last proof produced in the sequence, can be sure of the correct computation of every step.

We require two things from the proof system for recursive composition to work. First, that it is expressive enough to be able to accept its own verification algorithm as something to prove statements about, and second, that the verification algorithm is small enough so that the prover algorithm does not grow on each step. SNARKs are of particular interest, since they provide a computationally sound proof of small size compared to the size of the statement [BCCT12]. In particular, pairing-based SNARKs [PHGR13, Gro16, GWC19] make use of elliptic-curve pairings for verification of proofs, achieving verification time that does not depend on the size of the statement being proven. In this chapter, we focus on this type of SNARKs, due to the appeal of constant verification time.

A pairing-based SNARK relies on an elliptic curve $E/\mathbb{F}_q$ for some prime q, and such that $E(\mathbb{F}_q)$ has a large subgroup of prime order p. With this setting, the SNARK is able to prove satisfiability of arithmetic circuits over $\mathbb{F}_p$. However, the proof will be composed of elements in $\mathbb{F}_p$ and, crucially, elements in $E(\mathbb{F}_q)$. Each of these latter elements, although they belong to a group of order p, are represented as a pair of elements in $\mathbb{F}_q$. Moreover, the verification involves operations on the curve, which have formulas that use $\mathbb{F}_q$-arithmetic. Therefore, recursive composition of SNARK proofs requires to write the $\mathbb{F}_q$-arithmetic, derived from the verification algorithm, with an $\mathbb{F}_p$-circuit. Since $\mathbb{F}_p$-circuit satisfiability is an NP complete problem, it is possible to simulate $\mathbb{F}_q$-arithmetic via $\mathbb{F}_p$-operations, but this solution incurs into an efficiency blowup of $O(\log q)$ compared to native arithmetic [BSCTV17, Section 3.1].

Ideally, we would like $q = p$. However, there is a linear-time algorithm for solving the discrete logarithm problem on curves of this kind [Sma99]. Therefore, we shall assume that $p \neq q$. In this case, one approach is to instantiate a new copy of the SNARK with another elliptic curve E' to deal with $\mathbb{F}_q$-circuits. In [CFH+15], the authors propose to use a *2–chain* of pairing-friendly elliptic curves to achieve bounded recursive proof composition. A 2-chain of (pairing-friendly) elliptic curves is a tuple of pairing-friendly elliptic curves (E_1, E_2), defined over $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$, where $p_1 \mid \#E_2(\mathbb{F}_{p_2})$.

A more ambitious approach, proposed in [BSCTV17], is to use pairs of curves that also satisfy that $p_2 \mid \#E_1(\mathbb{F}_{p_1})$. In this case, the pair of curves is called a *2–cycle*. By alternating the instantiation of the SNARK with the two curves of the cycle, it is possible to allow unbounded recursive composition of the SNARK without incurring into non-native arithmetic simulation. Although this idea can also be used with longer cycles, 2-cycles are the optimal choice for recursive SNARKs, because they only require the generation and maintenance of two CRS.

5.1.1 Contributions and organization

In this chapter, we continue with the line of research of [CCW19] and tackle some of the open problems suggested by the authors. In Section 5.3, we review the background material on families of pairing-friendly curves with prime order. In Section 5.4, we recall the notion of cycles of elliptic curves, and what is known about them. We also present some new results, in particular a lower bound on the trace of curves involved in a 2-cycle, when both curves have the same (small) embedding degree. In Section 5.5 we study whether a combination of curves from different families can form a 2-cycle. This answers one of the open questions from [CCW19], for the case of 2-cycles.

Theorem 5.21 (informal). Parametric families either form 2-cycles as polynomials or only form finitely many pairing-friendly 2-cycles, and these can be explicitly bounded.

Moreover, we show that no curve from any of the known families can be in a 2-cycle in which the other curve has embedding degree $\ell \leq 22$, even going a bit further in some cases. This is achieved by combining the previous theorem with explicit computations for each of the families. These results shed some light over the difficulty of finding new cycles of elliptic curves, considering the fact that polynomial families are the only known way to produce pairing-friendly elliptic curves with prime order. In Section 5.6 we estimate the density of pairing-friendly cycles among all cycles. In [BK98], Balasubramanian and Koblitz estimated the density of pairing-friendly curves. We generalize their result to cycles of pairing-friendly curves. In Section 5.7, we discuss some future research directions. In Appendix A.2, we include SageMath code that we used to derive some of the results of this chapter.

5.2 Related work

Silverman and Stange [SS11] introduced and did a systematic study on 2-cycles of elliptic curves. As they show in their paper, in general, cycles of elliptic curves are easy to find. However, for recursive composition of pairing-based SNARKs, we need to be able to compute a pairing operation on the curves of the cycle. For this reason, curves need to have a *low* embedding degree, so that the pairing can be computed in a reasonable amount of time. Such curves are called *pairing-friendly* curves.

In [CCW19], Chiesa, Chua, and Weidner focused on cycles of pairing-friendly curves. In particular, they showed that only prime-order curves can form cycles. The only known method to produce prime-order curves is via families of curves parameterized by polynomials, and currently there are only five that are known. The first three of these families were introduced by Miyaji, Nakabayashi, and Takano [MNT01], who characterized all prime-order curves with embedding degrees 3, 4, and 6. These are called MNT curves. Based on the work from [GMV07], Barreto and Naehrig [BN05] found a new family of curves with embedding degree 12, and later Freeman [Fre06] found another one with embedding degree 10. The only known cycles are formed by alternating MNT curves of embedding degrees 4 and 6 [KT08, CCW19]. As proposed in [BSCTV17], these cycles can be used to instantiate recursive composition of SNARKs, but due to their very low embedding degree, the parameter sizes need to be very large to avoid classical discrete-logarithm attacks [MOV93], making the whole construction slow. Furthermore, the fact that the embedding degrees are different leads to an unbalance in the parameters, making one curve larger than necessary. Therefore, it would be desirable to have 2-cycles in which both curves have the same embedding degree k, for k a bit larger than in MNT curves. For instance, [CCW19] suggests embedding degrees 12 or 20. This would allow for more efficient instantiations of protocols that make use of recursive composition of pairing-friendly SNARKs.

A characterization of all the possible cycles consisting of MNT curves is given in [CCW19]. They also showed that there are no cycles consisting of curves from only the Freeman or Barreto–Naehrig (BN) families. They also gave some properties and impossibility results about pairing-friendly cycles, suggesting that adding the condition of pairing-friendliness to the curves of a cycle is a strong requirement: while cycles of curves are easy to find, cycles of pairing-friendly curves are not.

Recent progress has focused on chains of elliptic curves [EHG22] but there are still some interesting problems in the direction of cycles. In particular, [CCW19] lists some open problems, such as studying 2-cycles where the two curves have the same embedding degree or finding a cycle by combining curves from different families.

5.3 Pairing-friendly elliptic curves

Notation. Throughout this chapter, we assume that $p, q, q_i > 3$ are prime numbers. We denote by $\mathbb{F}_q$ the finite field with q elements. For $n \in \mathbb{N}$, we

denote by $\varphi(n)$ the Euler's totient function on n, and by Φ_n the n-th cyclotomic polynomial, which has degree $\varphi(n)$. A polynomial $g \in \mathbb{Q}[X]$ is *integer-valued* if $g(x) \in \mathbb{Z}$ for all $x \in \mathbb{Z}$.

5.3.1 Elliptic curves

An *elliptic curve* E over $\mathbb{F}_q$ (denoted $E/\mathbb{F}_q$) is a smooth algebraic curve of genus 1, defined by the equation

$$Y^2 = X^3 + aX + b,$$

for some $a, b \in \mathbb{F}_q$ such that $4a^3 - 27b^2 \neq 0$. We denote the group of $\mathbb{F}_q$-rational points by $E(\mathbb{F}_q)$, and refer to $\#E(\mathbb{F}_q)$ as the *order* of the curve. The neutral point is denoted by O. Given $m \in \mathbb{N}$, the *m-torsion group* of E is $E[m] = \{P \in E(\overline{\mathbb{F}}_q) \mid mP = O\}$, where $\overline{\mathbb{F}}_q$ is the algebraic closure of $\mathbb{F}_q$. When $q \nmid m$, we have that $E[m] \cong \mathbb{Z}_m \times \mathbb{Z}_m$. The *trace of Frobenius* (often called just *trace*) of E is

$$t = q + 1 - \#E(\mathbb{F}_q).$$

Hasse's theorem [Sil94, Theorem V.1.1] states that $|t| \leq 2\sqrt{q}$, and Deuring's theorem [Cox89, Theorem 14.18] states that, for any $t \in \mathbb{Z}$ within the Hasse bound, there exists an elliptic curve $E/\mathbb{F}_q$ with trace t.

A curve is said to be *supersingular* when $q \mid t$, and *ordinary* otherwise. Since we work with $q > 3$ prime, the Hasse bound implies that the only supersingular curves are those with $t = 0$. In the case of ordinary curves, the endomorphism ring will be an order $\mathcal{O} \subseteq \mathbb{Q}(\sqrt{d})$, where d is the square-free part of $t^2 - 4q$. The value d is called the *discriminant* of the curve E, and we say that E has *complex multiplication* by $\mathcal{O}$. Note that the Hasse bound implies that $d < 0$.[1]

Pairings and the embedding degree

Let $E/\mathbb{F}_q$ be an elliptic curve. Then, for m such that $q \nmid m$, we can build a *pairing*

$$e : E[m] \times E[m] \to \mu_m,$$

where $E[m] \cong \mathbb{Z}_m \times \mathbb{Z}_m$ is the m-torsion group of the curve and μ_m is the group of mth roots of unity. The map e is bilinear, i.e. $e(aP, bQ) = e(P, Q)^{ab}$ for any $P, Q \in E[m]$. Various instantiations of this map exist, e.g. the Weil

[1] Other works take $|d|$ as the discriminant.

pairing [Sil94, §III.8]. Since $\mu_m \subset \mathbb{F}_{q^k}^*$ for some $k \in \mathbb{N}$ and is a multiplicative subgroup, it follows that $m \mid q^k - 1$. The smallest k satisfying this property is called the *embedding degree* of $E[m]$. When $m = \#E(\mathbb{F}_q)$, we refer to this k as the embedding degree of E.

Proposition 5.1. Let $E/\mathbb{F}_q$ be an elliptic curve of prime order p. The following conditions are equivalent:

- E has embedding degree k.
- k is minimal such that $p \mid \Phi_k(q)$ [MNT01, Remark 1].
- k is minimal such that $p \mid \Phi_k(t-1)$ [BLS02, Lemma 1].

Most curves have a very large embedding degree [BK98, Theorem 2]. This has a direct impact on the computational cost of computing the pairing. On the one hand, we want small embedding degrees to ensure efficient arithmetic. On the other hand, however, small embedding degrees open an avenue for attacks, more precisely the [MOV93] and [FR94] reductions. These translate the discrete logarithm problem on the curve to the discrete logarithm problem on the finite field $\mathbb{F}_{q^k}$, where faster (subexponential) algorithms are known. With a small embedding degree, we are forced to counteract the reduction to finite field discrete logarithms by increasing our parameter sizes. Therefore, a balanced embedding degree is preferred when using pairing-friendly curves.

We note the following result, useful for finding curves with small embedding degree.

Proposition 5.2. Let $E/\mathbb{F}_q$ be an elliptic curve with prime order p and embedding degree k such that $p \nmid k$. Then $p \equiv 1 \pmod{k}$.

Proof. The embedding degree condition is equivalent to k being minimal such that $q^k \equiv 1 \pmod{p}$. Since p is prime, by Lagrange's theorem we have that $k \mid p - 1$.

The complex multiplication (CM) method

Let $E/\mathbb{F}_q$ be an elliptic curve with prime order p and trace t. The embedding degree condition is determined by p and q alone, so the actual coefficients of the curve equation do not play any role. Because of this, the main approach to finding pairing-friendly curves tries to find (t, p, q) first, and then curve coefficients that are compatible with these values.

Given (t, p, q) such that $p = q + 1 - t$ and $t \leq 2\sqrt{q}$, Deuring's theorem ensures that a curve exists, but that does not mean that it is easy to find. The algorithm that takes (t, p, q) and produces the curve coefficients is known as the *complex multiplication (CM) method*, and its complexity strongly depends on the discriminant d of the curve. Currently, this is considered feasible up to $|d| \approx 10^{16}$ [Sut12].

Because of our focus on finding *good* triples (t, p, q), we will identify curves with them. That is, we write $E \leftrightarrow (t, p, q)$ as shorthand for an elliptic curve $E/\mathbb{F}_q$ with order p and trace t. This curve might not be unique, but any of them will have the same embedding degree and discriminant, so they are indistinguishable for our purposes.

5.3.2 Pairing-friendly polynomial families

The idea of considering families of elliptic curves parameterized by low-degree polynomials is already present in [MNT01, BN05], but is studied in a more systematic way in [Fre06, FST10]. We will consider triples of polynomials $(t, p, q) \in \mathbb{Q}[X]^3$ such that, given $x \in \mathbb{Z}$, there is an elliptic curve $E \leftrightarrow (t(x), p(x), q(x))$.

We are interested in prime-order elliptic curves, so we require that the polynomials p, q represent primes.

Definition 5.1. Let $g \in \mathbb{Q}[X]$. We say that g *represents primes* if:

- $g(X)$ is irreducible, non-constant and has a positive leading coefficient,
- $g(x) \in \mathbb{Z}$ for some $x \in \mathbb{Z}$ (equivalently, for infinitely many such x), and
- $\gcd\{g(x) \mid x, g(x) \in \mathbb{Z}\} = 1$.

The Bunyakovsky conjecture [Peg] states that a polynomial in the conditions of the definition above takes prime values for infinitely many $x \in \mathbb{Z}$. We now formally define polynomial families of pairing-friendly elliptic curves.

Definition 5.2. Let $k, d \in \mathbb{Z}$ with $d < 0 < k$. We say that a triple of polynomials $(t, p, q) \in \mathbb{Q}[X]^3$ *parameterizes a family of elliptic curves* with embedding degree k and discriminant d if:

1. $p(X) = q(X) + 1 - t(X)$,
2. p is integer-valued (even if its coefficients are in $\mathbb{Q} \setminus \mathbb{Z}$),
3. p and q represent primes,

4. $p(X) \mid \Phi_k(t(X)-1)$, and
5. the equation $4q(X) = t(X)^2 + |d|Y^2$ has infinitely-many integer solutions (x, y).

We naturally extend the notation $E \leftrightarrow (t, p, q)$ to polynomial families.

Conditions 1-3 ensure that the polynomials represent infinitely many sets of parameters compatible with an elliptic curve. Condition 4 ensures that the embedding degree is at most k, where ideally k is small. Condition 5 ensures that there are infinitely many curves in the family with the same discriminant d. If this d is not too large, we will be able to use the CM method to find the curves corresponding to these parameters. If we ignore condition 5, such families are not too hard to find, as illustrated by the following lemma.[2]

Lemma 5.3. For any integer $k \geq 3$ there are infinitely many pairs (q, E_q) with embedding degree k, and such that $|E(\mathbb{F}_q)|$ is prime, under the Bunyakovsky conjecture.

Proof. Infinite families are known for $k = 3, 4, 6$, as detailed below in Table 5.1. We can then assume $\varphi(k) \geq 4$. We will construct a family represented by the polynomial tuple (t, p, q) as follows.

Let $p(X) = \Phi_{rk}(X)$, for some prime number r such that $r \nmid k$. Then, it holds that $\varphi(kr) \geq 4(r-1) \geq 2r$. We set $q(X) = p(X) + X^r$. Then

$$p(X) \mid X^{rk} - 1 = (X^r)^k - 1 = (q(X) - p(X))^k - 1,$$

so $p(X) \mid q(X)^k - 1$. In this case $p(X) = q(X) - X^r$, so the trace is given by $t(X) = 1 + X^r$, and $\deg(t) \leq \deg(p)/2$. Also, the cyclotomic polynomial is irreducible, so it represents infinitely many prime values.

Let $f(X) = 4q(X) - t(X)^2$. Freeman [Fre06] observed that condition 5 in Definition 5.2 is strongly related to the form of this polynomial.

Proposition 5.4. Fix $k \in \mathbb{N}$, and let $(t, p, q) \in \mathbb{Z}[X]^3$ satisfying conditions (1-4) in the previous definition. Assume that one of these holds:

- $f(X) = aX^2 + bX + c$, with $a, b, c \in \mathbb{Z}$, $a > 0$ and $b^2 - 4ac \neq 0$. There exists a discriminant d such that ad is not a square. Also, the CM equation has an integer solution.

[2] Furthermore, numerical experiments easily find many tuples (t, p, q) with low degree and small coefficients satisfying conditions 1-4, but unfortunately not condition 5.

- $f(X) = (\ell X + |d|)g(X)^2$ for some discriminant d, $\ell \in \mathbb{Z}$, and $g \in \mathbb{Z}[X]$.

Then, we have that (t, p, q) parameterizes a family of elliptic curves with embedding degree k and discriminant d.

On the other hand, if $\deg f \geq 3$, it is unlikely to produce a family of curves, as highlighted by the following result, which is a direct consequence of Siegel's theorem [Sil94, Corollary IX.3.2.2].

Proposition 5.5. Fix $k \in \mathbb{N}$, and let (t, p, q) as above, and satisfying conditions (1-4) in the previous definition. Assume that $f(X)$ is square-free and $\deg f \geq 3$. Then (t, p, q) cannot represent a family of elliptic curves with embedding degree k.

Finally, [Fre06] also proves some results on the relations between the degrees of the polynomials involved in representing a family of curves.

Proposition 5.6. Let $t \in \mathbb{Q}[X]$. Then, for any k and any irreducible factor $p \mid \Phi_k(t-1)$, we have that $\varphi(k) \mid \deg p$.

Proposition 5.7. Let (t, p, q) represent a family of curves with embedding degree k, with $\varphi(k) \geq 4$. If $f = 4q - t^2$ is square-free, then:

- $\deg p = \deg q = 2\deg t$.
- If a is the leading coefficient of $t(X)$, then $a^2/4$ is the leading coefficient of $p(X), q(X)$.

Known pairing-friendly families with prime order

Only a few polynomial families of elliptic curves with prime order and low embedding degree are known. The first work in this direction is due to Miyaji, Nakabayashi, and Takano, [MNT01], who characterized all prime-order curves with embedding degrees $k = 3, 4, 6$ (these correspond to $\varphi(k) = 2$). Based on the work of Galbraith, McKee and Valença [GMV07], two additional families were found: Barreto and Naehrig [BN05] found a family with $k = 12$, and Freeman [Fre06] found another one with $k = 10$ (both cases have $\varphi(k) = 4$). Note, however, that their results are not exhaustive, meaning that there could still be other families with these embedding degrees that have not been found, unlike in the MNT case. We summarize the polynomial descriptions of these families in Table 5.1.

Family	k	$t(X)$	$p(X)$	$q(X)$
MNT3	3	$6X-1$	$12X^2-6X+1$	$12X^2-1$
MNT4	4	$-X$	X^2+2X+2	X^2+X+1
MNT6	6	$2X+1$	$4X^2-2X+1$	$4X^2+1$
Freeman	10	$10X^2+5X+3$	$25X^4+25X^3+15X^2+5X+1$	$25X^4+25X^3+25X^2+10X+3$
BN	12	$6X^2+1$	$36X^4+36X^3+18X^2+6X+1$	$36X^4+36X^3+24X^2+6X+1$

Table 5.1: Polynomial descriptions of MNT, Freeman, and BN curves, where k corresponds to the embedding degree, $t(X)$ is the trace, $p(X)$ is the order, and $q(X)$ is the order of the base field.

For completeness, we note that there are no elliptic curves with prime order and embedding degree $k \leq 2$, except for a few cases of no cryptographic interest.

Proposition 5.8. Let $p, q \in \mathbb{Z}$ be prime numbers. If $q \geq 14$, then there is no elliptic curve $E/\mathbb{F}_q$ with $\#E(\mathbb{F}_q) = p$ and embedding degree $k \leq 2$.

Proof. Suppose that such a curve exists.

- If $k = 1$, then $p \mid q-1$. Clearly $p \neq q-1$, since otherwise p, q cannot both be prime. Then $p \leq \frac{q-1}{2}$, and then $q - p \geq \frac{q+1}{2}$. But, at the same time, $q - p = t - 1 \leq 2\sqrt{q} - 1$, due to the Hasse bound. These two conditions are only compatible when $q \leq 9$, which is already ruled out by hypothesis.
- If $k = 2$, then $p \mid q^2 - 1 = (q-1)(q+1)$. We have that $p \nmid q-1$ (otherwise $k = 1$), and thus $p \mid q+1$ because p is prime. Again, $p \neq q+1$, because otherwise p, q cannot both be prime. Then $p \leq \frac{q+1}{2}$, and thus $q - p \geq \frac{q-1}{2}$. By the Hasse bound, $q - p \leq 2\sqrt{q} - 1$, and these are only compatible for $q < 14$.

5.4 Cycles of elliptic curves

5.4.1 Definition and known results

The notion of cycles of elliptic curves was introduced in [SS11].

Definition 5.3. Let $s \in \mathbb{N}$. An *s-cycle of elliptic curves* is a tuple $(E_1, \ldots, E_s)$ of elliptic curves, defined over fields $\mathbb{F}_{q_1}, \ldots, \mathbb{F}_{q_s}$, respectively, and such that

$$\#E_i(\mathbb{F}_{q_i}) = q_{i+1 \bmod s},$$

for all $i = 1, \ldots, s$.

Remark 5.1. Cycles of length 2 have some particular properties that are worth noting. Let E, E' be two curves forming a 2-cycle.

- If $E \leftrightarrow (t, p, q)$, then Definition 5.3 implies that $E' \leftrightarrow (2 - t, q, p)$.
- We have that $p = \#E(\mathbb{F}_q)$ is in the Hasse interval of $q = \#E'(\mathbb{F}_p)$ if and only if q is in the Hasse interval of p. Indeed, if the former holds, then

$$\sqrt{p} - 1 \leq \sqrt{q} \leq \sqrt{p} + 1,$$

which is equivalent to

$$\sqrt{q} - 1 \leq \sqrt{p} \leq \sqrt{q} + 1.$$

It is known that cycles of any length exist [SS11, Theorem 11]. We summarize in the following two propositions some facts about cycles. These results are due to [CCW19].

Proposition 5.9. Let $E_1, \ldots, E_s$ be an s-cycle of elliptic curves, defined over prime fields $\mathbb{F}_{q_1}, \ldots, \mathbb{F}_{q_s}$. Then:

(i) $E_1, \ldots, E_s$ are ordinary curves.

(ii) If $q_1, \ldots, q_s > 12s^2$, then $E_1, \ldots E_s$ have prime order.

(iii) Let $t_1, \ldots, t_s$ be the traces of $E_1, \ldots, E_s$, respectively. Then

$$\sum_{i=1}^{s} t_i = s.$$

(iv) If $s = 2$, then the curves in the cycle have the same discriminant d.

(v) If the curves in the cycle have the same discriminant $|d| > 3$, then $s = 2$.

(vi) If $s > 2$ and $E_1, \ldots, E_s$ have the same discriminant d, then necessarily $s = 6$ and $|d| = 3$.

There are also some impossibility results.

Proposition 5.10. We have the following.

(i) There is no 2-cycle with embedding degree pairs $(5, 10)$, $(8, 8)$ or $(12, 12)$.

(ii) There is no cycle formed only by Freeman curves.

(iii) There is no cycle formed only by BN curves.

5.4.2 Some properties of cycles

In this section, we show some results about cycles, most of them about 2-cycles in which both curves have the same embedding degree. We start with a small result that rules out safe primes in 2-cycles with the same embedding degree.

Proposition 5.11. Safe primes are not part of any 2-cycle in which both curves have the same embedding degree k.

Proof. Let p, q be the orders of the curves in the cycle. Assume that p is a safe prime, i.e. $p = 1 + 2r$, with r prime. Since p, q are in a cycle, $q = p + 1 - t$ for some $|t| \leq 2\sqrt{p}$. Now, since $k \mid p - 1$ by Proposition 5.2, we have $k = 1, 2, r, 2r$. We already know that $k \neq 1, 2$, hence $k \in \{r, 2r\}$. Since q also has embedding degree k, again by Proposition 5.2 we have that $k \mid q - p$, and thus $r \mid q - p$. Therefore

$$|q - p| \geq r = \frac{p-1}{2} > 2\sqrt{p} + 1$$

for any $p > 3$, which contradicts the fact that $|q - p| = |1 - t| < 2\sqrt{p} + 1$.

Proposition 5.12. Let $s \in \mathbb{Z}$, and let $(t, p, q) \in \mathbb{Q}[X]^3$ parameterize a family of pairing-friendly elliptic curves, with $\deg t$ even. Then, there are only finitely many s-cycles such that all s curves in the cycle belong to the family.

Proof. If s curves with traces $t_1, \ldots, t_s$, respectively, form a cycle, by Proposition 5.9.(iii) we have that $\sum_{i=1}^{s} t_i = s$. Since $\deg t \geq 2$ and s is fixed, necessarily there exist $a, b \in \{1, \ldots, s\}$ such that t_a, t_b have different signs. However, since $\deg t$ is even, there exists a lower bound b such that, for all $|x| > b$, we have that $t(x)$ has the same sign. Therefore, only finitely many cases can occur in which the traces have opposing sign.

Given an elliptic curve $E \leftrightarrow (t, p, q)$, Hasse's theorem gives us the bound $|t| \leq 2\sqrt{q}$, which in the polynomial case implies that $\deg t \leq \frac{1}{2}\deg q$. We now derive a lower bound for t in the case of 2-cycles in which both curves have the same small embedding degree. We require first the following technical lemma.

Lemma 5.13. Let $k \in \mathbb{N}$ and $3 \leq k \leq 104$. We have that:

(i) For any $|x| > 1$,

$$\Phi_k(x) \leq \frac{|x|}{|x| - 1} x^{\varphi(k)}.$$

(ii) For any $\varepsilon > 0$, there exists $B > 0$ such that, for all x with $|x| > B$,

$$\Phi_k(x-1) \leq (1+\varepsilon)\frac{|x|}{|x|-1}x^{\varphi(k)}.$$

Proof. Clearly such bound exists for $|x|$ large enough, since $\Phi_k(x) = x^{\varphi(k)} + o\left(x^{\varphi(k)}\right)$. More precisely, for $k \leq 104$, the k-th cyclotomic polynomial has only 0 and ± 1 as coefficients [Mig83]. Therefore

$$\begin{aligned}\Phi_k(x) &\leq x^{\varphi(k)} + \sum_{i=0}^{\varphi(k)-1} |x|^i = x^{\varphi(k)}\left(1 + \sum_{i=1}^{\varphi(k)} \frac{1}{|x|^i}\right) \\ &\leq x^{\varphi(k)}\left(1 + \frac{1}{|x|-1}\right) = \frac{|x|}{|x|-1}x^{\varphi(k)},\end{aligned}$$

using the fact that the geometric series converges when $|x| > 1$.

Part (ii) is now trivial when $x > 0$. For $x < 0$, we note that, since Φ_k is a polynomial with positive leading coefficient, for any $\varepsilon > 0$ there exists $B > 0$ such that, for all x with $|x| > B$,

$$\Phi_k(x-1) \leq (1+\varepsilon)\Phi_k(x),$$

since otherwise the function would grow exponentially fast when $x \to -\infty$. The result follows directly from applying part (i) to $\Phi_k(x)$.

Remark 5.2. More precisely, for k such that $3 \leq k \leq 104$, we do not need to choose B too large to achieve a small constant. The following values have been obtained computationally.

$1+\varepsilon$	2	1.1	1.01
B	146	1069	10250

Proposition 5.14. Let $E \leftrightarrow (t, p, q)$ be an elliptic curve with embedding degree k, with $|t| > 1$ and $3 \leq k \leq 104$. Then, for any $\varepsilon > 0$ there exists $B > 0$ such that, for all t with $|t| > B$, we have

$$|t| > \left(\frac{1}{1+\varepsilon}\frac{|t|-1}{|t|}q\right)^{\frac{1}{\varphi(k)}}.$$

Proof. We have that $p \mid \Phi_k(t-1)$, so $p \leq \Phi_k(t-1)$. Also, we have that $|t| < |\Phi_k(t) - \Phi_k(t-1)|$ for $|t|$ large enough, since Φ_k is at least quadratic in t. Assume first that $t > 1$. Then, combining these upper bounds on p and t, and using part (i) of the previous lemma, we obtain

$$q = p - 1 + t \leq p + t < \Phi_k(t) \leq \frac{t}{t-1} t^{\varphi(k)}.$$

Taking $\varphi(k)$-th roots,

$$t > \left(\frac{t-1}{t} q\right)^{\frac{1}{\varphi(k)}}.$$

The case $t < -1$ is completely analogous, using part (ii) of Lemma 5.13.

The result above deals with a single curve, but actually it can be strengthened for some 2-cycles.

Proposition 5.15. Let $E \leftrightarrow (t, p, q)$ and $E' \leftrightarrow (2-t, q, p)$ be two elliptic curves with $|t| > 1$ and the same embedding degree $k \equiv 0 \pmod 4$, such that $3 \leq k \leq 104$. Then, for any $\varepsilon > 0$ there exists $B > 0$ such that, for all t with $|t| > B$, we have

$$|t| > \left(\frac{1}{1+\varepsilon} \frac{|t|-1}{|t|} q\right)^{\frac{2}{\varphi(k)}}.$$

Proof. The case $k \equiv 0 \pmod 4$ corresponds to those cyclotomic polynomials such that $\Phi_k(x) = \Phi_k(-x)$ for all x. From the embedding degree conditions, we have

$$p \mid \Phi_k(t-1),$$
$$q \mid \Phi_k(1-t),$$

and therefore $pq \mid \Phi_k(t-1)$, since p, q are different primes. Assume, without loss of generality, that $q < p$. Then $q^2 \leq pq \leq \Phi_k(t-1)$, and proceeding as the proof of Proposition 5.14, we obtain

$$q^2 \leq (1+\varepsilon) \frac{|t|}{|t|-1} t^{\varphi(k)},$$

from which we obtain the desired bound.

Corollary 5.16. Let $E \leftrightarrow (t, p, q)$ and $E' \leftrightarrow (2-t, q, p)$ be two elliptic curves with the same embedding degree $k \equiv 0 \pmod 4$, such that $3 \leq k \leq 104$. There exists B such that, if $|t| > B$, then

$$\frac{1}{2} q^{\frac{2}{\varphi(k)}} < |t| \leq 2q^{\frac{1}{2}}.$$

Remark 5.3. The result above is particularly interesting in two cases:

- When $\varphi(k) = 2$, i.e. $k = 4$. In this case,

$$\frac{1}{2}q < |t| \leq 2q^{\frac{1}{2}},$$

  which cannot happen for $q > 15$. This shows that there are no $(4, 4)$-cycles (which was already known from [CCW19]).

- When $\varphi(k) = 4$, i.e. $k \in \{8, 12\}$. In this case,

$$\frac{1}{2}q^{\frac{1}{2}} < |t| \leq 2q^{\frac{1}{2}},$$

  which shows that t asymptotically behaves like $\sqrt{q}$, and therefore is on the outermost part of the Hasse interval. In particular, for polynomial families this means that $\deg t = \frac{1}{2}\deg p$, which improves on the inequality known before.

5.5 Cycles from known families

In this section, we prove our main result about 2-cycles of elliptic curves: given a family $(t, p, q) \in \mathbb{Q}[X]^3$ with embedding degree k, and $\ell \in \mathbb{N}$, one of two things can happen:

(a) $q \mid p^\ell - 1$, as polynomials. In this case, any curve in the family forms a 2-cycle with the corresponding curve in the family $(2 - t, q, p)$, which has embedding degree ℓ (see Proposition 5.1). Observe that, due to Proposition 5.9, both families have the same discriminant.

(b) Only finitely many curves from the family form a 2-cycle with curves of embedding degree ℓ.

Furthermore, when we are in the second case we can explicitly find these cycles. For all known families (Table 5.1), we prove that no curve from them (except for a few anecdotal cases) is part of a 2-cycle with any curve with embedding degree $\ell \leq L$. The bound L depends on the family, and in all cases at least $L \geq 22$.

5.5.1 Cycles from parametric-families

First, we show a technique that will help us rule out many cases from our main results, by performing a very simple check.

Proposition 5.17. Let $(t, p, q) \in \mathbb{Q}[X]^3$ parameterize a family of pairing-friendly elliptic curves. Let a curve E from the family be in a cycle, and assume that the previous curve in the cycle has embedding degree ℓ. Then there exists $i \in \{0, \ldots, \ell - 1\}$ such that

$$q(i) \equiv 1 \pmod{\ell}.$$

Proof. Let $x \in \mathbb{Z}$ such that $E \leftrightarrow (t(x), p(x), q(x))$, and let $E' \leftrightarrow (t', p', q')$ be the previous curve in the cycle with embedding degree ℓ. From the definition of cycle, $p' = q(x)$. Then, applying Proposition 5.2 to curve E', we deduce that

$$q(x \bmod \ell) \equiv q(x) \equiv p' \equiv 1 \pmod{\ell}.$$

By testing the condition given by Proposition 5.17 for known families and $3 \leq \ell \leq 100$, we obtain the following results. Note that this not help for BN curves, since in that case $q(0) = 1$, and thus the condition holds for any ℓ.

Corollary 5.18. An MNT3 curve cannot be preceded in a cycle by a curve with embedding degree ℓ, where

$$\begin{aligned}\ell \in \{&3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 24, 26, 27, 28, 30, 31, 32,\\ &33, 34, 35, 36, 37, 39, 40, 41, 42, 44, 45, 48, 49, 51, 52, 54, 55, 56, 57, 59, 60, 61,\\ &62, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 87,\\ &88, 89, 90, 91, 92, 93, 96, 98, 99, 100\}.\end{aligned}$$

Corollary 5.19. A Freeman curve cannot be preceded in a cycle by a curve with embedding degree ℓ, where

$$\begin{aligned}\ell \in \{&4, 5, 8, 10, 11, 12, 15, 16, 20, 22, 24, 25, 28, 30, 32, 33, 35, 36, 40, 44, 45, 48, 50,\\ &52, 53, 55, 56, 59, 60, 61, 64, 65, 66, 68, 70, 72, 75, 76, 77, 79, 80, 83, 84, 85, 88,\\ &90, 92, 95, 96, 97, 99, 100\}.\end{aligned}$$

Furthermore, even when we cannot rule out a certain ℓ, we obtain a condition on $x \bmod \ell$, which will help us later when we check by brute force all x in an interval. Also note that, despite the fact that we will use these corollaries to simplify our work in the next section, which deals with 2-cycles, these results work for cycles of any length.

5.5.2 2-cycles from parametric families

The goal here will be to start from a known family of pairing-friendly elliptic curves, and argue that they form no 2-cycles with other pairing-friendly curves. To do so, let (t, p, q) represent such family. For any curve $E \leftrightarrow (t(x), p(x), q(x))$, there is another curve $E' \leftrightarrow (2 - t(x), q(x), p(x))$ such that the two of them form a 2-cycle. Furthermore, if E' has a small embedding degree $\ell \in \mathbb{Z}$, then $q(x) \mid p(x)^\ell - 1$. Note that this is for a particular $x \in \mathbb{Z}$, not as polynomials.

Informally, our strategy will be the following. The embedding degree condition on E' can be reformulated in terms of integer division: the division of $p(x)^\ell$ by $q(x)$ has remainder 1. We will compare integer division and polynomial division, and show that, outside of a finite interval $[N_{\mathsf{left}}, N_{\mathsf{right}}]$, the remainders in both cases essentially agree. Therefore, by showing that the polynomial remainder $r(x)$ never takes the value 1, we will rule out any possibility of cycles outside of $[N_{\mathsf{left}}, N_{\mathsf{right}}]$. For known families of curves, we will deal with the cases $x \in [N_{\mathsf{left}}, N_{\mathsf{right}}]$ manually, as there are only a finite number of them, and show that none of them leads to a partner curve with small embedding degree.

Lemma 5.20. Let $x \in \mathbb{Z}$, and let $a, b \in \mathbb{Q}[X]$ be two integer-valued polynomials. Assume that b has even degree and positive leading coefficient.

- Let $h, r \in \mathbb{Q}[X]$ be the quotient and remainder, respectively, of the polynomial division of a by b. Let $c > 0$ be the smallest integer such that $ch, cr \in \mathbb{Z}[X]$.
- Let $h_x, r_x \in \mathbb{Z}$ be the quotient and remainder, respectively, of the integer division of $ca(x)$ by $b(x)$.

Then either $\deg r = 0$, or there exist $N_{\mathsf{left}}, N_{\mathsf{right}} \in \mathbb{Z}$ and $\delta_{\mathsf{left}}, \delta_{\mathsf{right}} \in \{0, 1\}$ such that:

- For all $x < N_{\mathsf{left}}$, we have that $\operatorname{sign}(r(x))$ is constant, and $r_x = cr(x) + \delta_{\mathsf{left}} b(x)$.
- For all $x > N_{\mathsf{right}}$, we have that $\operatorname{sign}(r(x))$ is constant, and $r_x = cr(x) + \delta_{\mathsf{right}} b(x)$.

Furthermore, let us denote $\sigma_{\mathsf{left}} = \operatorname{sign}\{r(x) \mid x < N_{\mathsf{left}}\}$ and $\sigma_{\mathsf{right}} = \operatorname{sign}\{r(x) \mid x > N_{\mathsf{right}}\}$. Then

$$\delta_{\mathsf{left}} = \frac{1 - \sigma_{\mathsf{left}}}{2}, \qquad \delta_{\mathsf{right}} = \frac{1 - \sigma_{\mathsf{right}}}{2}.$$

Proof. We observe that c is well-defined, as it can be taken as the least common multiple of all denominators occurring in the coefficients of h, r. Likewise, $\sigma_{\mathsf{left}}, \sigma_{\mathsf{right}}$ are well-defined, since r is a polynomial, and thus at most it changes sign $\deg r$ times. For the second part, we have that

$$\begin{aligned} ca(x) &= b(x)h_x + r_x, \\ ca(x) &= b(x)(ch(x)) + cr(x), \end{aligned}$$

where $0 \leq r_x < b(x)$, and $\deg r < \deg b$, and all these values are integer. Subtracting, we obtain

$$r_x - cr(x) = b(x)(ch(x) - h_x),$$

and thus $r_x \equiv cr(x) \pmod{b(x)}$. Since $0 \leq r_x < b(x)$, we just need to find $cr(x) \bmod b(x)$, as this will necessarily be the same as r_x.

We illustrate the technique for the case $\sigma_{\mathsf{left}} = -1, \sigma_{\mathsf{right}} = 1$ (the other cases are completely analogous). Note that, if $\deg r > 0$, then r is not a constant polynomial.

- Let $N_{\mathsf{left}} \in \mathbb{Z}$ be the largest integer such that $0 < -cr(x) \leq b(x)$ for all $x < N_{\mathsf{left}}$. Such N_{left} exists because both $b(x), -cr(x) \to \infty$ when $x \to -\infty$, and $\deg b > \deg(-cr)$. If $x < N_{\mathsf{left}}$, then $0 < -cr(x) \leq b(x)$. Multiplying by (-1), we get that $-b(x) \leq cr(x) < 0$, and adding $b(x)$, we get $0 \leq cr(x) + b(x) < b(x)$. Therefore, $r_x = cr(x) + b(x)$.
- Let $N_{\mathsf{right}} \in \mathbb{Z}$ be the smallest integer such that $0 \leq cr(x) < b(x)$ for all $x > N_{\mathsf{right}}$. Such N_{right} exists because both $b(x), cr(x) \to \infty$ when $x \to \infty$, and $\deg b > \deg(cr)$. If $x > N_{\mathsf{right}}$, then $0 \leq cr(x) < b(x)$. Therefore, necessarily $r_x = cr(x)$.

We can now prove the main theorem of this section, from which the desired results will directly follow.

Theorem 5.21. Let $k, \ell \in \mathbb{N}$. Let (t, p, q) be a triple of polynomials parameterizing a family of elliptic curves with embedding degree k. Then either $q \mid p^\ell - 1$ as polynomials, or there are at most finitely many 2-cycles formed by a curve from the family and a curve with embedding degree ℓ. Furthemore, in the second case, there exist efficiently computable bounds such that all cycles considered above are within those bounds.

Proof. Due to Proposition 5.8, we can safely assume that $k, \ell \geq 3$. Assume that there exists a 2-cycle involving a curve E from the family and another curve E'

with embedding degree ℓ. That is, assume that there exists $x \in \mathbb{Z}$ such that $E \leftrightarrow (t(x), p(x), q(x))$ is in a 2-cycle. Then $E' \leftrightarrow (2 - t(x), q(x), p(x))$. By the condition of the embedding degree, we have that

$$q(x) \mid p(x)^\ell - 1,$$

and thus there exists $h \in \mathbb{Z}$ such that

$$p(x)^\ell = q(x)h + 1.$$

We now wish to apply Lemma 5.20, with $a = p^\ell$ and $b = q$, so we must argue that q has even degree and positive leading coefficient. We distinguish two cases:

- For $k \in \{3, 4, 6\}$, all the prime-order families are the MNT families, which have $\deg q = 2$ and positive leading coefficient.
- For k with $\varphi(k) \geq 4$, we have from Proposition 5.6 that $\varphi(k) \mid \deg p$, and in this case $\varphi(k)$ is always even. Furthermore, since $p = q + 1 - t$ and $t = O(\sqrt{q})$ (due to the Hasse bound), necessarily $\deg q = \deg p$. Now, since q has even degree, it necessarily has positive leading coefficient, otherwise it could not represent infinitely many curves.

Let $h, r \in \mathbb{Q}[X]$ be the quotient and remainder, respectively, of the polynomial division of p^ℓ by q. If $q \nmid p^\ell - 1$ as polynomials, then $r \neq 1$. If r is another constant polynomial, then the embedding degree condition does not hold for any $x \in \mathbb{Z}$. If $\deg r > 0$, Lemma 5.20 gives us $c, N_{\mathsf{left}}, N_{\mathsf{right}} \in \mathbb{Z}, \delta_{\mathsf{left}}, \delta_{\mathsf{right}} \in \{0, 1\}$ such that, if $x < N_{\mathsf{left}}$,

$$cr(x) + \delta_{\mathsf{left}} b(x) = 1,$$

and, if $x > N_{\mathsf{right}}$, then

$$cr(x) + \delta_{\mathsf{right}} b(x) = 1.$$

The polynomials $cr(X)$ and $cr(X) + b(X)$ can only take the value 1 finitely many times. By enlarging $[N_{\mathsf{left}}, N_{\mathsf{right}}]$ if necessary, we can ensure that this only happens inside of $[N_{\mathsf{left}}, N_{\mathsf{right}}]$. Therefore, there are no cycles for $x \notin [N_{\mathsf{left}}, N_{\mathsf{right}}]$.

This result immediately yields the following consequences for concrete families of curves. Let (t, p, q) parametrize a family of curves. Given a certain value of ℓ, it is immediate to check whether $q \nmid p^\ell - 1$ as polynomials. If that is not the case (which happens most of the time), Theorem 5.21 ensures that there are at most finitely-many cycles formed by a curve from the family and a curve with

embedding degree ℓ. For each candidate ℓ, we compute the values $c, N_{\mathsf{left}}, N_{\mathsf{right}}$ from Theorem 5.21 corresponding to the division of p^ℓ by q. Interestingly, $c = 1$ for all known families of pairing-friendly curves with prime order. The resulting values of $N_{\mathsf{left}}, N_{\mathsf{right}}$ are summarized in Table 5.2 for the MNT3, Freeman, and BN families. No tables are included for MNT4 and MNT6 families because, in these cases, we have $N_{\mathsf{left}} = -1, N_{\mathsf{right}} = 0$ and $N_{\mathsf{left}} = N_{\mathsf{right}} = 0$, respectively, regardless of ℓ.

Remark 5.4. Given arbitrary integer-valued polynomials $p, q \in \mathbb{Q}[X]$ and $\ell \in \mathbb{N}$, there is no guarantee that the polynomial remainder of p^ℓ by q will have integer coefficients, i.e. $c = 1$, or even be integer-valued. Nevertheless, this does happen for MNT, Freeman, and BN curves.

Freeman curves. We proceed by induction on ℓ. For $\ell = 1$, we have that

$$p(X) \bmod q(X) = -10X^2 - 5X - 2.$$

This polynomial is of the form $25aX^3 + 5bX^2 + 5cX + d$, for some $a, b, c, d \in \mathbb{Z}$. We will now show that, if $p^\ell \bmod q$ is of this form, then $p^{\ell+1} \bmod q$ is also of this form. This will prove that all the remainder is actually in $\mathbb{Z}[X]$ for any $\ell \in \mathbb{N}$.
Hence, suppose that there exist $a, b, c, d \in \mathbb{N}$ such that

$$p(X)^\ell \bmod q(X) = 25aX^3 + 5bX^2 + 5cX + d.$$

Then

$$\begin{aligned} p(X)^{\ell+1} &\equiv p(X)^\ell p(X) \equiv \left(25aX^3 + 5bX^2 + 5cX + d\right)\left(-10X^2 - 5X - 2\right) \\ &\equiv -250aX^5 - (125a + 50b)X^4 - (50a + 25b + 50c)X^3 \\ &\quad -(10b + 25c + 10d)X^2 - (10c + 5d)X - 2d \\ &\equiv (75a + 25b - 50c)X^3 + (-25a + 40b - 25c - 10d)X^2 \\ &\quad +(-20a + 20b - 10c - 5d)X + (-15a + 6b - 2d) \pmod{q(X)}. \end{aligned}$$

Since the coefficient of degree 3 is divisible by 25, and the coefficients of degree 2 and 1 are divisible by 5, the induction step works.

MNT3 curves. In this case, $q(X) = 12X^2 - 1$. We proceed by induction on ℓ. For $\ell = 1$, we have that

$$p(X) \bmod q(X) = -6X + 2,$$

which is of the form $6aX + b$, for some $a, b \in \mathbb{Z}$. We show that, if $p^\ell \bmod q$ is of this form, then so is $p^{\ell+1} \bmod q$. Then all the remainders will actually be in

Bounds for MNT3

ℓ	N_{left}	N_{right}
5	-104	104
10	-75658	75657
19	-10626317415	10626317415

Bounds for Freeman

ℓ	N_{left}	N_{right}
3	-2	4
6	-164	161
7	-686	685
9	-10608	10607
13	-1805067	1805066
14	-6158596	6158595
17	-210958904	210958905
18	-643610018	643610019
19	-1875810507	1875810508
21	-12522961240	12522961243
23	-15125575810	15125575853

Bounds for BN

ℓ	N_{left}	N_{right}
3	-1	0
4	-3	4
5	-12	11
6	-15	4
7	-65	64
8	-104	103
9	-167	168
10	-831	830
11	-513	508
12	-3523	3524
13	-8620	8619
14	-4092	4097
15	-52351	52350
16	-66417	66414
17	-164463	164464
18	-626817	626816
19	-186373	186364
20	-2992820	2992819
21	-6014684	6014683
22	-5673471	5673474
23	-41263041	41263040
24	-39448697	39448694
25	-151319223	151319224
26	-462478015	462478014
27	-20593636	20593693
28	-2473968276	2473968275
29	-4050737756	4050737755
30	-6238668798	6238668799
31	-31854421247	31854421246
32	-20649322466	20649322461

Table 5.2: Bounds $N_{\mathsf{left}}, N_{\mathsf{right}}$ from Lemma 5.20 for different embedding degrees ℓ of the potential partner curve of MNT3, Freeman, and BN curves. The remaining intermediate values of ℓ are covered by Corollaries 5.18 and 5.19 for MNT3 and Freeman curves, respectively.

$\mathbb{Z}[X]$. Assume that there exist $a, b, c, d \in \mathbb{N}$ such that

$$p(X)^\ell \bmod q(X) = 6aX + b.$$

Then

$$\begin{aligned} p(X)^{\ell+1} &\equiv p(X)^\ell p(X) \equiv (6aX + b)(-6X + 2) \\ &\equiv -36aX^2 + (12a - 6b)X + 2b \\ &\equiv (12a - 6b)X + (-3a + 2b) \pmod{q(X)}. \end{aligned}$$

Since the coefficient of degree 1 is divisible by 6, the induction step works.

BN curves. In this case, $q(X) = 36X^4 + 36X^3 + 24X^2 + 6X + 1$. Assume that there exist $a, b, c, d \in \mathbb{N}$ such that

$$p(X)^\ell \bmod q(X) = 36aX^3 + 6bX^2 + 6cX + d,$$

for some $a, b, c, d \in \mathbb{Z}$. Then

$$\begin{aligned} p(X)^{\ell+1} &\equiv p(X)^\ell p(X) \equiv \left(36aX^3 + 6bX^2 + 6cX + d\right)\left(-6X^2\right) \\ &\equiv -216aX^5 - 36bX^4 - -36cX^3 - 6dX^2 \\ &\equiv (-72a + 36b - 36c)X^3 + (-108a + 24b - 6d)X^2 \\ &\quad +(-30a + 6b)X + (-6a + b) \pmod{q(X)}. \end{aligned}$$

Since the coefficient of degree 3 is divisible by 36, and the coefficients of degree 2 and 1 are divisible by 6, the induction step works.

Remark 5.5. The values of $N_{\mathsf{left}}, N_{\mathsf{right}}$ in MNT4 and MNT6 families are in stark contrast with the other families (shown in Appendix **??**), but can be easily explained. In MNT3, Freeman, and BN curves, the remainder r of the polynomial division q^k by p has coefficients that mostly increase with k. Because of this, we need to get further away from zero before the asymptotic behavior kicks in.

On the contrary, only a small number of remainders are possible in MNT4 and MNT6 curves. Let $(t, p, q) \in \mathbb{Q}[X]^3$ parameterize MNT4 curves. We have that $q \mid p^6 - 1$ (they form infinitely many cycles with MNT6 curves). That is, p has order 6 modulo q, and thus $p^k \bmod q$ can only take 6 possible values. Concretely, $p(X)^k \bmod q(X) \in \{\pm 1, \pm X, \pm(X + 1)\}$ for any $k \in \mathbb{N}$, and all of these yield the bounds $N_{\mathsf{left}} = -1, N_{\mathsf{right}} = 0$. Similarly, in the case of MNT6 curves, the remainder of p^k by q can only take 4 values. Concretely $p(X)^k \bmod q(X) \in \{\pm 1, \pm 2X\}$ for any $k \in \mathbb{N}$, which yield the bounds $N_{\mathsf{left}} = N_{\mathsf{right}} = 0$.

An exhaustive search in $[N_{\mathsf{left}}, N_{\mathsf{right}}]$ reveals no curves with embedding degree ℓ, for any of the values of ℓ considered, except for a few examples with no cryptographic interest (see Table 5.3). We consider MNT3, Freeman, and BN curves, since it is already known [CCW19] that MNT4 and MNT6 curves are only in cycles with each other.

Corollary 5.22. Let (E, E') be a 2-cycle of elliptic curves, and assume that E is not one of the curves described in Table 5.3.

(i) If E is an MNT3 curve, then E' has embedding degree $\ell \geq 23$.

(ii) If E is a Freeman curve, then E' has embedding degree $\ell \geq 26$.

(iii) If E is a BN curve, then E' has embedding degree $\ell \geq 33$.

Family	k	ℓ	x	t	p	q
MNT3	3	10	-1	-7	19	11
MNT3	3	10	1	5	7	11
BN	12	18	-1	7	13	19

Table 5.3: Instances of curves $E \leftrightarrow (t, p, q)$, with embedding degree k, from known cycles that form a pairing-friendly 2-cycle with another curve E' with embedding degree ℓ.

The computational check took a few hours on a standard computer, using the SageMath code from Appendix A.2. Theoretically, there is no reason to stop at a given embedding degree ℓ. However, the interval $[N_{\mathsf{left}}, N_{\mathsf{right}}]$ grow rapidly, making the brute force check inside of the interval a much more serious computing effort, requiring a more polished implementation. Still, the most interesting cases are those with smaller embedding degree, as the ideal cycles for recursive composition would be those in which the embedding degrees of both curves are as close as possible.

5.6 Density of pairing-friendly cycles

The previous sections have been mostly an algebraic treatment of cycles. In this section, we look at cycles from a different angle, concerning ourselves with their density. The goal is to quantify in concrete terms the folklore notion that pairing-friendly cycles are hard to find. Our starting point is the following

result of [BK98]. It proves an upper bound on the probability of a random elliptic curve being pairing-friendly.[3]

Theorem 5.23 ([BK98], Theorem 2). Let $M \in \mathbb{Z}$. Let $\mathfrak{p}$ be the probability of finding an elliptic curve $E/\mathbb{F}_q$ with prime order $p \in [M, 2M]$ and embedding degree $k \leq (\log q)^2$, by sampling uniformly from all the curves with orders in the interval $[M, 2M]$. Then

$$\mathfrak{p} < c\frac{(\log M)^9(\log\log M)^2}{M},$$

for some constant $c > 0$.

We generalize the result above to s-cycles of elliptic curves. In particular, an s-cycle is a collection of s primes $q_1, \ldots, q_s$ and s elliptic curves $E_1/\mathbb{F}_{q_1}, \ldots, E_s/\mathbb{F}_{q_s}$, such that $\#E_i(\mathbb{F}_{q_i}) = q_{i+1 \bmod s}$. Among these, we are interested in finding those with small embedding degrees. As s increases, the number of cycles also increases. However, since the embedding degree condition is imposed on every step of the cycle, the probability decreases dramatically with s, as this is a very strong requirement. We start by stating the main result of this section.

Theorem 5.24. Let $s \geq 2$, $K > 0$, and $M \in \mathbb{Z}$. Let $\mathfrak{p}$ be the probability of finding an s-cycle of elliptic curves $E_1/\mathbb{F}_{q_1}, \ldots, E_s/\mathbb{F}_{q_s}$ with $q_i \in [M, 2M]$ and embedding degrees $k_i \leq K$ for all $i = 1, \ldots, s$, by sampling uniformly from all the s-cycles of elliptic curves with orders in the interval $[M, 2M]$. Then

$$\mathfrak{p} < cK(K+1)\frac{(\log M)^{3s}(\log\log M)^{2s}}{M^{s/2}},$$

for some constant $c > 0$ depending on s.

We will prove our result above through a sequence of lemmas. The overall strategy is as follows: in Lemma 5.25, we count the number of s-tuples of primes within the interval $[M, 2M]$ that are compatible with the Hasse condition. In Lemma 5.26, we impose an upper bound K on the embedding degree. Finally, in Lemmas 5.28 and 5.29, we count the curves that are compatible with the primes counted in the previous two results.

[3] In [BK98], the authors define pairing friendliness as having an embedding degree $k \leq (\log q)^2$. We will keep the bound as an unspecified parameter K.

We start by disregarding the curves and just looking at the primes. In order to get a cycle, we need an s-tuple of primes $q_1, \ldots, q_s$ that fit in the Hasse interval of each other, i.e. $|q_{i+1} - q_i - 1| \leq 2\sqrt{q_i}$. Thus, we first count the s-tuples of possible primes $q_1, \ldots, q_s$ that are not too far apart.

Lemma 5.25. Let $s \geq 2$ be a fixed positive integer and $C > 0$ a constant depending on s. For any $M \geq 2$ we denote by $T_s(M)$ the number of s-tuples of primes in the interval $[M, 2M]$ with $|q_i - q_j| \leq C\sqrt{M}$. Then, there exist constants c_5, c_9 depending on s, such that

$$c_5 \frac{M^{(s+1)/2}}{(\log M)^s} \leq T_s(M) \leq c_9 \frac{M^{(s+1)/2}}{(\log M)^s}.$$

Proof. We split the interval $[M, 2M]$ in subintervals $I_k = [M+(k-1)C\sqrt{M}, M+kC\sqrt{M})$ for $1 \leq k \leq \lfloor \sqrt{M}/C \rfloor$ and call π_k the number of primes on the interval I_k. We denote $M_C = M + C\left\lfloor \frac{\sqrt{M}}{C} \right\rfloor \sqrt{M}$. Observe that $2M - M_C \leq C\sqrt{M}$ and, hence, the prime number theorem gives

$$\sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k = \pi(M_C) - \pi(M) = \frac{M}{\log M} + e,$$

where $|e| < \varepsilon \frac{M}{\log M}$ for any $\varepsilon > 0$ and $M > M_\varepsilon$ sufficiently large, depending on ε. Then, a simple application of Hölder's inequality [BB61, Chapter 1, Theorem 2] for $p = s$ and $q = \frac{s}{s-1}$ gives us that, for $M > M_\varepsilon$,

$$\begin{aligned}(1-\varepsilon)\frac{M}{\log M} \leq \sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k &\leq \left(\sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} 1 \right)^{(s-1)/s} \left(\sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k^s \right)^{1/s} \\ &\leq c_1 M^{(s-1)/2s} \left(\sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k^s \right)^{1/s}.\end{aligned} \tag{5.1}$$

Hence,

$$\frac{c_2 M^{(s+1)/2}}{(\log M)^s} \leq \sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k^s.$$

Finally, observe that every s-tuple of primes on each interval I_k is counted in $T_s(M)$, so we can use the above expression to get a lower bound on $T_s(M)$. Let

A be the set of indices k such that the interval I_k has more than $(s+1)^2$ primes. Now, since for any $N_1 > 0$ and $N_2 > 1$ we have the following inequality [MV73, Corollary 2],

$$\pi(N_1 + N_2) - \pi(N_1) \leq \frac{2N_2}{\log N_2}, \tag{5.2}$$

we get that $\pi_k \leq c_3 \frac{\sqrt{M}}{\log M}$ for any k. Therefore,

$$\begin{aligned} \frac{M}{\log M} &\sim \sum_{k\in A} \pi_k + \sum_{k\in \overline{A}} \pi_k < c_3 \frac{\sqrt{M}}{\log M} \#A + (s+1)^2(\sqrt{M} - \#A) \\ &< c_3 \frac{\sqrt{M}}{\log M} \#A + (s+1)^2 \sqrt{M}, \end{aligned}$$

which gives us the bound

$$\#A > c_4 \sqrt{M}$$

for any M sufficiently large. Now, to get the lower bound, we look at the variations of s-tuples of primes in each interval I_k for $1 \leq k \leq \lfloor \sqrt{M}/C \rfloor$.

$$\begin{aligned} T_s(M) &\geq \sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \frac{\pi_k!}{(\pi_k - s)!} \geq \sum_{k\in A} \frac{\pi_k!}{(\pi_k - s)!} = \sum_{k\in A} \pi_k^s \prod_{j=0}^{s-1} \left(1 - \frac{j}{\pi_k}\right) \\ &> \sum_{k\in A} \pi_k^s e^{-s(s+1)/\pi_k} > \frac{1}{e} \sum_{k\in A} \pi_k^s \\ &= \frac{1}{e} \left(\sum_{k=1}^{\lfloor \sqrt{M}/C \rfloor} \pi_k^s - \sum_{k\in \overline{A}} \pi_k^s \right) \geq c_5 \frac{M^{(s+1)/2}}{(\log M)^s} - \frac{1}{e}(s+1)^{2s}\sqrt{M} \\ &> c_6 \frac{M^{(s+1)/2}}{(\log M)^s}. \end{aligned}$$

In order to prove the second inequality, we denote the primes in the interval $[M, 2M]$, in increasing order, as $q_1, \ldots, q_N$. If we have an s-tuple starting with q_i, then the rest of the $s-1$ primes on the s-tuple will be in the interval $I_i = (q_i, q_i + C\sqrt{M}]$. Hence, letting $\pi_i = \sum_{q\in I_i} 1$, we can apply the inequality of Equation (5.2) to obtain

$$T_s(M) \leq \sum_{i=1}^{N} \binom{\pi_i}{s-1} \leq c_7 \sum_{i=1}^{N} \pi_i^{s-1} \leq c_8 \frac{M^{\frac{s-1}{2}}}{(\log M)^{s-1}} N \leq c_9 \frac{M^{\frac{s+1}{2}}}{(\log M)^s}.$$

Remark 5.6. For $s = 2$ and $C = 1$ we can get any constant $c_5 < 1/2$. Note that, when $C = 1$, we have $c_1 = 1$ in Equation (5.1), and thus it yields the inequality

$$\sum_{k=1}^{\lfloor\sqrt{M}\rfloor} \pi_k^s \geq (1-\varepsilon)\frac{M^{(s+1)/2}}{(\log M)^s},$$

for any $\varepsilon > 0$. Then, we observe that, for M large enough,

$$\begin{aligned} T_2(M) &\geq \frac{1}{2}\sum_{k=1}^{\sqrt{M}} \pi_k(\pi_k - 1) = \frac{1}{2}\sum_{k=1}^{\sqrt{M}} \pi_k^2 - \frac{1}{2}\sum_{k=1}^{\sqrt{M}} \pi_k \\ &\geq \frac{1}{2}\frac{M^{3/2}}{(\log M)^2} - \frac{1}{2}\frac{M}{\log M} \geq \left(\frac{1}{2} - \varepsilon'\right)\frac{M^{3/2}}{(\log M)^2}, \end{aligned}$$

where in the last inequality we have used that the first term is asymptotically dominant. A different proof of the lower bound for the case $s = 2$ and $C = 1$, with a slightly worse constant, is given in [Kob91, Lemma 1].

Now, let us impose the condition of having very small embedding degree.

Lemma 5.26. For any $M > 0$ and $K > 0$, let $T_{s,K}(M)$ be the number of s-tuples of primes in the interval $[M, 2M]$, with $|q_i - q_j| \leq C\sqrt{M}$, for some constant $C > 0$ and such that $q_{i+1} \mid q_i^{k_i} - 1$ for some $k_i \leq K$. Then

$$T_{s,K}(M) \leq c_2 K(K+1)\sqrt{M},$$

for some constant $c_2 > 0$.

Proof. We proceed similarly to [BK98]. First note that if $q_{i+1} \mid q_i^{k_i} - 1$, then $q_{i+1} \mid (q_i - q_{i+1})^{k_i} - 1$ and, since $|q_i - q_j| \leq C\sqrt{M}$, we have that for any i there exists an integer $|h_i| \leq C\sqrt{M}$ such that $q_{i+1} \mid h_i^{k_i} - 1$ for some $k_i \leq K$. Now, since $q_{i+1} > M \geq (Ch_i)^2$, we see that ${h_i}^{k_i} - 1$ has at most $c_1 \frac{k_i}{2}$ prime divisors on the interval $[M, 2M]$, for some constant $c_1 > 0$. Summing over the possible k and h we get

$$T_{s,K}(M) \leq \sum_{k \leq K} \sum_{|h| \leq C\sqrt{M}} \sum_{q \mid h^k - 1} 1 \leq c_2 K(K+1)\sqrt{M}.$$

Finally, we bring curves back into the equation. Given an interval $[M, 2M]$, we will count the tuples of curves with orders in the intervals, and the subset

of those such that every curve in the tuple is pairing-friendly. Theorem 5.24 will follow directly from these. We introduce the following result from [Len87], which we will require for the proof.

Lemma 5.27 ([Len87], Propositon 1.9). Let $q > 3$ be a prime number, let $P \subset \mathbb{N}$ and let $N_{q,P}$ be the number of isomorphism classes of elliptic curves over $\mathbb{F}_q$ and order $\#E(\mathbb{F}_q) \in P$. Then:

- If $P \subset [q+1-2\sqrt{q}, q+1+2\sqrt{q}]$, then $N_{q,P} \leq c\#P(\log q)(\log\log q)^2\sqrt{q}$ for some constant $c > 0$.
- If $P \subset [q-\sqrt{q}, q+\sqrt{q}]$ and $\#P \geq 3$, then $N_{q,P} \geq c(\#P-2)\frac{\sqrt{q}}{\log q}$ for some constant $c > 0$.

Lemma 5.28. Let $M \geq 2$, and let $C_s(M)$ be the number of s-tuples of elliptic curves $E_1/\mathbb{F}_{q_1}, \ldots, E_s/\mathbb{F}_{q_s}$ forming a cycle of length s, where $q_i \in [M, 2M]$ for all $i = 1, \ldots, s$. Then there exist constants c_7, c_2, depending on s, such that

$$c_7 \frac{M^{(2s+1)/2}}{(\log M)^{2s}} \leq C_s(M) \leq c_2(\log\log M)^{2s} M^{(2s+1)/2}.$$

Proof. First, note that, if we have an s-cycle of curves, then the corresponding primes are as in Lemma 5.25 for any $C > s$. Without loss of generality, let us assume that cycles start at the smallest prime. Now, if we have an s-tuple in which the smallest prime is q_1, then the rest of the $s-1$ primes on the s-tuple will be in the interval $I_i = (q_1, q_1 + s\sqrt{q}_1 + (s/2)^2]$. To see this, we first prove a result by induction. Let $q_\ell, q_{\ell+1}$ be the ℓ-th and $(\ell+1)$-th primes in the cycle, respectively. The induction hypothesis is that $q_\ell \leq (\sqrt{q_1} + \ell)^2$ (the base case is true due to the Hasse bound). Then,

$$\begin{aligned} q_{\ell+1} &\leq q_\ell + 2\sqrt{q}_\ell + 1 \leq q_1 + 2\ell\sqrt{q}_1 + \ell^2 + 2\sqrt{q_1 + 2\ell\sqrt{q}_1 + \ell^2} + 1 \\ &= q_1 + 2(\ell+1)\sqrt{q}_1 + (\ell+1)^2 = (\sqrt{q_1} + (\ell+1))^2, \end{aligned}$$

concluding the induction step. From here, we deduce that, for any $\ell = 1, \ldots, s$, we have that

$$\sqrt{q_\ell} - \sqrt{q_1} \leq \ell.$$

Since they form a cycle, then it must be the case that $q_\ell \in I_i$ for all ℓ (note that there are at most $s/2$ primes between the largest and the smallest prime of a cycle).

Now, let us start by proving the upper bound for $C_s(M)$. Let P be a subset of primes p satisfying that $|p-(q+1)| \leq 2q$. By the first part of Lemma 5.27, we know that there are at most $c_1\sqrt{q}\log q(\log\log q)^2\#P$ isomorphism classes over $\mathbb{F}_q$ of elliptic curves with $\#E(\mathbb{F}_q) \in P$ for some constant c_1. Taking P with $\#P = s$ and multiplying over each prime of an s-tuple we get that, on each s-tuple, there will be less than

$$c_2(\log M)^s(\log\log M)^{2s}M^{s/2}$$

isomorphism classes of elliptic curves with points on the s-tuple and, in particular, forming a cycle of length at most s. Note that the constant c_2 depends on s. Applying the second inequality of Lemma 5.25, we get the expected upper bound for cycles of length at most s, and in particular for $C_s(M)$.

To prove the lower bound for $C_s(M)$ we use the second part of Lemma 5.27. In this case, for any q and any subset of primes $P \subset [q-\sqrt{q}, q+\sqrt{q}]$ with $\#P \geq 3$ there are more than $c_3(\#P-2)\frac{\sqrt{q}}{\log q}$ isomorphism classes over $\mathbb{F}_q$ of elliptic curves with $\#E(\mathbb{F}_q) \in P$ for some constant c_3. Hence, on each s-tuple with $s \geq 3$ we have more than $c_4\frac{M^{s/2}}{(\log M)^s}$ isomorphism classes of elliptic curves with points on the s-tuple and, in particular, forming a cycle of length at most s. Note that c_4 is a constant that depends on s. Observe that, in particular, all those primes lie on the Hasse interval for q, since $P \subset [q-\sqrt{q}, q+\sqrt{q}] \subset [q+1-2\sqrt{q}, q+1+2\sqrt{q}]$. Combining this with the first inequality of Lemma 5.25, we get the lower bound

$$c_5\frac{M^{(2s+1)/2}}{(\log M)^{2s}}.$$

Then, $C_s(M)$ will be cycles of isomorphism classes of elliptic curves of length at most s minus cycles of isomorphism classes of elliptic curves of length at most $s-1$. In order to bound the number of cycles of length at most $s-1$, we use the previous upper bound for $C_i(M)$, for $i = 1, \ldots, s-1$, so we get

$$\sum_{i=1}^{s-1}(\log\log M)^{2i}M^{(2i+1)/2} \leq c_6(\log\log M)^{2s-2}M^{(2s-1)/2}$$

for some constant c_6. Hence,

$$C_s(M) \geq c_5\frac{M^{(2s+1)/2}}{(\log M)^{2s}} - c_6(\log\log M)^{2s-2}M^{(2s-1)/2} \geq c_7\frac{M^{(2s+1)/2}}{(\log M)^{2s}},$$

for some constant c_7 and for M sufficiently large depending on s.

By mimicking the second part of the previous proof, but using Lemma 5.26 instead of Lemma 5.25, we obtain the following analogous result.

Lemma 5.29. Let $M \geq 2$. Let $C_{s,K}(M)$ be the number of s-tuples in the same conditions as in Lemma 5.28, which additionally satisfy that E_i has embedding $k_i \leq K$ for all $i = 1, \ldots, s$. Then there exists a constant c, depending on s, such that

$$C_{s,K}(M) \leq cK(K+1)(\log M)^s(\log\log M)^{2s}M^{(s+1)/2}.$$

Finally, by dividing $C_{s,K}(M)$ by $C_s(M)$ from Lemmas 5.28 and 5.29, we get Theorem 5.24 .

5.7 Conclusions

Cycles of elliptic curves require the curves involved to be of prime order, and families of elliptic curves parameterized by low-degree polynomials are the only known approach at generating pairing-friendly curves with prime order. In this chapter, we have shown that this approach is unlikely to yield new cycles, beyond the MNT4-MNT6 cycles that are already known. In particular, we have shown that no known families are involved in a 2-cycle with any pairing-friendly curve of cryptographic interest. While a lot is still unknown about pairing-friendly cycles, we highlight two avenues that we consider interesting for future research.

- Generalizing Theorem 5.21 and Corollary 5.22 to s-cycles, for $s > 2$. The case $s = 2$ is the most appealing from a practical perspective, due to the application to recursive composition of SNARKs, but it would be desirable to have the complete picture. The main hurdle here is that, whereas fixing a curve in a 2-cycle automatically determines the other, longer cycles have more degrees of freedom, so we do not have as much explicit information to work with in the proof.

- Consider a 2-cycle such that both curves $E \leftrightarrow (t, p, q)$ and $E' \leftrightarrow (2-t, q, p)$ have the same embedding degree k. If we restrict ourselves to the case $k \equiv 0 \pmod 4$, it is easy to argue (as in Proposition 5.15) that

  $$pq \mid \Phi_k(t-1).$$

  This approach allows [CCW19] to prove that said cycles cannot exist when $k \in \{8, 12\}$. However, the authors leave higher values of k as an open question. If we consider families of curves, Theorem 5.21 tells us that the

above relation must hold as polynomials, or else only a finite number of cycles will exist. Thus, we wonder if considering the above condition as a relation between polynomials, and applying polynomial machinery, could help in answering this question.

Chapter 6

Conclusions

When the trust is high, communication is easy, instant, and effective.

– Stephen Covey

The initial papers on interactive proofs and zero-knowledge from the 1980s primarily contributed to expanding the theoretical boundaries of these proof systems. This changed with the emergence of distributed ledgers, which became a killer use case for ZK proofs. These proof systems have become a subject of intense study for enhancing the privacy and scalability of blockchains, which has led to the exploration of more efficient and generic protocols to address practical requirements within the blockchain space.

This symbiosis between academia and the blockchain industry has fostered a diverse community working on ZK proofs that encompasses computer science theorists, cryptographers, mathematicians, engineers, programmers, and financial analysts. Each profile brings unique expertise and perspectives to the field, with individuals specializing in different aspects of ZK proofs while considering other elements as black boxes.

For instance, most popular proof systems assume statements in R1CS form, but they do not delve into the optimal and secure formulation of the statements. In this regard, in Chapter 3 of this thesis, we introduced CIRCOM, a novel programming language that empowers developers to describe circuits at the

constraint level. The modularity of CIRCOM allows each circuit to be treated independently and implement optimizations at the template level. Ultimately, the clarity of the language facilitates circuit analysis and auditing, and it opens the door to a more in-depth study of the security an efficiency of arithmetic circuits.

In the subsequent sections of this thesis we delved into elliptic curves, an aspect often overlooked by both programmers and cryptographers. Typically, ZK protocols consider elliptic curves as abstract groups, without dealing with the specific instantiation of these groups in a practical scenario. In this thesis, we demonstrated that comprehending these mathematical objects, rather than treating them as black boxes, opens a wide range of interesting research problems. In this thesis, we have addressed two topics. On the one hand, in Chapter 4, we proposed a method for generating elliptic curves with efficient arithmetic within circuits. On the other hand, in Chapter 5, we addressed the search for suitable cycles enabling the recursive composition of ZK systems—an idea that holds significant potential for enhancing the performance and adoption of blockchains.

In conclusion, this PhD thesis has contributed to the advancement of arithmetic circuits and elliptic curves for ZK proofs in the context of blockchain technology. By bridging the gap between theoretical foundations and practical applications, this research has provided valuable insights, novel methodologies, and potential solutions to enhance privacy, scalability, and efficiency in blockchain systems. The findings and innovations presented in this thesis pave the way for further exploration and development in the field, offering new possibilities for the future of secure and decentralized systems.

Bibliography

[0xpa] 0xparc. zk–ECDSA: zk–SNARKs for EcDSA. 23

[0xpb] 0xparc. zk–SNARKs for elliptic-curve pairings. 40

[AB07] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Princeton University, 2007. 2

[ABI+22] Elvira Albert, Marta Bellés-Muñoz, Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Distilling constraints in zero-knowledge protocols. In Sharon Shoham and Yakir Vizel, editors, *Computer aided verification (CAV)*, pages 430–443, Cham, 2022. Springer International Publishing. 48

[AGR+16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 42

[AHG22] Diego Aranha, Youssef Housni, and Aurore Guillevic. A survey of elliptic curves for proof systems. *Designs, Codes and Cryptography*, 12 2022. 57

[Ale] Aleo. The Leo programming language. GitHub. 19

[BB61] Edwin F Beckenbach and Richard Bellman. *Inequalities*. Ergebnisse der Mathematik und ihrer Grenzgebiete. 2. Folge. Springer Berlin, Heidelberg, 1961. 105

[BBB+17] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. 3

[BBDM22] Marta Bellés–Muñoz, Jordi Baylina, Vanesa Daza, and Jose Luis Muñoz-Tapia. New privacy practices for blockchain software. *IEEE Software*, 39(03):43–49, May 2022. 4, 16, 19

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *Progress in cryptology – AFRICACRYPT 2008*, pages 389–405, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 56, 59, 60, 61, 72

[BC68] Benjamin Baumslag and Bruce Chandler. *Schaum's outline of theory and problems of group theory.* Schaum's outline series. McGraw-Hill Book Company, New York, 1968. 75

[BC94] Daniel P. Bovet and Pierluigi Crescenzi. *Introduction to the theory of complexity.* Prentice Hall international series in computer science. Prentice Hall, 1994. 1, 2

[BCC+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 *Proceedings, part II, of the 35th annual international conference on advances in cryptology — EUROCRYPT 2016*, volume 9666, pages 327–357, Berlin, Heidelberg, 2016. Springer-Verlag. 10, 16

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pages 326–349, 2012. 82

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on security and privacy (SP)*, pages 947–964, Los Alamitos, CA, USA, may 2020. IEEE Computer Society. 58, 59

[BFM88] 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*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery. 3

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, SOSP '13, page 341–357, New York, NY, USA, 2013. Association for computing machinery. 40

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 216–233, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. 76

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC Conference on computer and communications security*, CCS '13, page 967–980, New York, NY, USA, 2013. Association for computing machinery. 66

[BK98] R. Balasubramanian and N. Koblitz. The improbability that an elliptic curve has subexponential discrete log problem under the Menezes-Okamoto-Vanstone algorithm. *Journal of Cryptology*, 11(2):141–145, 1998. 83, 86, 104, 107

[BL17] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the Montgomery ladder. Cryptology ePrint Archive, paper 2017/293, 2017. 63

[BL19] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. Available online: `https://safecurves.cr.yp.to`, 2019. 57, 65, 66

[BLS02] Paulo S.L.M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International conference on security in communication networks*, pages 257–267. Springer, 2002. 57, 86

[BMIMT+22] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2022. 4, 16

[BMRS20] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Mina: Decentralized cryptocurrency at scale. *New York University, O(1) Labs, New York, NY, USA, Whitepaper*, pages 1–47, 2020. 81

[BMUS22] Marta Bellés-Muñoz, Jorge Jiménez Urroz, and Javier Silva. Revisiting cycles of pairing-friendly elliptic curves. Cryptology ePrint Archive, Paper 2022/1662, 2022. https://eprint.iacr.org/2022/1662. 5

[BN05] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 12th International conference on selected areas in cryptography*, SAC'05, page 319–331, Berlin, Heidelberg, 2005. Springer-Verlag. 57, 84, 87, 89

[Bow19] Sean Bowe. Derivation of jubjub elliptic curve. GitHub, 2019. Available online: https://github.com/zkcrypto/jubjub/blob/master/doc/derive/derive.sage (accessed on 30 October 2021). 63

[BS84] László Babai and Endre Szemerédi. On the complexity of matrix group problems i. In *IEEE Annual Symposium on Foundations of Computer Science*, 1984. 3

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014. 15, 55

[BSCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in cryptology – CRYPTO 2014*, pages 276–294, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 58, 59

[BSCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, USA, 2014. USENIX Association. 10, 16, 40

[BSCTV17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017. 81, 82, 84

[BSS99] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*, volume 256 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1999. 56, 66

[BWB+21] Marta Bellés-Muñoz, Barry Whitehat, Jordi Baylina, Vanesa Daza, and Jose Luis Muñoz Tapia. Twisted Edwards elliptic curves for zero-knowledge circuits. *Mathematics*, 9(23), 2021. 5, 41, 42, 44

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001. 8

[CC16] Paul-Jean Cahen and Jean-Luc Chabert. What you should know about integer-valued polynomials. *The American Mathematical Monthly*, 123(4):311–337, 2016. 140

[CCW19] Alessandro Chiesa, Lynn Chua, and Matthew Weidner. On cycles of pairing-friendly elliptic curves. *SIAM Journal on Applied Algebra and Geometry*, 3(2):175–192, 2019. 83, 84, 91, 95, 103, 110

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 87–119, Berlin, Heidelberg, 1987. Springer-Verlag. 76

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee

Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270, 2015. 82

[Cox89] David A Cox. *Primes of the form $x^2 + ny^2$: Fermat, class field theory, and complex multiplication*. John Wiley & Sons, 1989. 85

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, volume 10, pages 310–331, 2010. 81

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster computing in zero knowledge. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*, pages 371–403. Springer, 2015. 81

[CvHP92] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 470–484, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. 76

[CWC+21] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Paper 2021/651, 2021. 19, 20

[Dam92] Ivan Damgård. Non-interactive circuit based proofs and non-interactive perfect zero-knowledge with proprocessing. In *International Conference on the Theory and Application of Cryptographic Techniques*, 1992. 3

[DF] Dark Forest. zk–SNARK space warfare. 23

[Edw07] Harold Edwards. A normal form for elliptic curves. *Bulletin of The American Mathematical Society - BULL AMER MATH SOC*, 44:393–423, 07 2007. 60

[EHG22] Youssef El Housni and Aurore Guillevic. Families of SNARK-friendly 2-chains of elliptic curves. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 367–396. Springer, 2022. 84

[EMP] EMP. Efficient multi-party computation toolkit. GitHub. 18, 19

[ET18] Jacob Eberhardt and Stefan Tai. ZoKrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018. 19, 20

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple noninteractive zero knowledge proofs under general assumptions. *SIAM Journal on Computing*, 29(1):1–28, 1999. 3

[FR94] Gerhard Frey and Hans-Georg Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of computation*, 62(206):865–874, 1994. 86

[Fre06] David Freeman. Constructing pairing-friendly elliptic curves with embedding degree 10. In *International Algorithmic Number Theory Symposium*, pages 452–465. Springer, 2006. 84, 87, 88, 89

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of cryptology*, 23(2):224–280, 2010. 87

[Ful08] William Fulton. *Algebraic curves. An introduction to algebraic geometry*. 2008. 14

[GBFS16] Vincent Gramoli, Len Bass, Alan D. Fekete, and Daniel W. Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2711–2724, 2016. 48

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 3, 11

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *30th USENIX Security*

Symposium (USENIX Security 21), pages 519–535. USENIX Association, aug 2021. 42

[GKV+18] Jens Groth, Yael Kalai, Muthu Venkitasubramaniam, Nir Bitansky, Ran Canetti, Henry Corrigan-Gibbs, Shafi Goldwasser, Charanjit Jutla, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, Tal Rabin, Mariana Raykova, Ron Rothblum, Alessandra Scafuro, Eran Tromer, and Douglas Wikström. Security track proceeding. Technical report, ZKProof standards, Berkeley, CA, May 2018. https://zkproof.org/documents.html. 21

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on theory of computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. Association for Computing Machinery. 3, 7, 15

[GMV07] Steven D Galbraith, James F McKee, and Paula C Valença. Ordinary abelian varieties having small embedding degree. *Finite Fields and Their Applications*, 13(4):800–814, 2007. 84, 89

[GMW91] 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*, 38(3):690–728, jul 1991. 15

[GO94] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, dec 1994. 15

[Gro10] Jens Groth. Short non-interactive zero-knowledge proofs. In Masayuki Abe, editor, *Advances in cryptology - ASIACRYPT 2010*, pages 341–358, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 8

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *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*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016. 4, 8, 28, 82

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PlonK: Permutations over Lagrange-bases for oecumenical non-interactive arguments of knowledge. Cryptology ePrint Archive, article 2019/953, 2019. 4, 8, 28, 82

[Han05] Helena Handschuh. *SHA Family (Secure Hash Algorithm)*, pages 565–567. Springer US, Boston, MA, 2005. 42

[HBHW19] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification version 2019.0.0 [overwinter+sapling]. May 1, 2019. 3, 75, 76, 78

[Her20] Hermez Network. Hermez whitepaper, October, 2020. 23, 48

[HG20] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. Cryptology ePrint Archive, Report 2020/351, 2020. Available online: https://ia.cr/2020/351 (accessed on 30 October 2021). 58, 59

[Hop17] Daira Hopwood. Supporting evidence for security of the jubjub curve to be used in zcash. Available online: https://github.com/daira/jubjub/blob/master/verify.sage (accessed on 30 October 2021), 2017. 65

[HU79] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979. 1

[HWCD08] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *Advances in cryptology - ASIACRYPT 2008*, pages 326–343, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 62, 65

[Ide20a] Iden3. CIRCOM: Circuit compiler for zero-knowledge proofs. GitHub, 2020. 16, 19

[Ide20b] Iden3. CIRCOMLIB: Library of circom templates. GitHub, 2020. 16, 40, 46, 79

[Ide20c] Iden3. SNARKJS: JavaScript implementation of zk-SNARKs. GitHub, 2020. 17, 18, 19, 28, 50

[JL17] S. Josefsson and I. Liusvaara. Edwards-curve Digital Signature Algorithm (EdDSA). Internet Research Task Force (IRTF). Request for Comments: 8032, January, 2017. 46, 56

[KB20] Assimakis Kattis and Joseph Bonneau. Proof of necessary work: Succinct state verification with fairness guarantees. *Cryptology ePrint Archive*, 2020. 81

[KKKK14] K. Kasamatsu, S. Kanno, T. Kobayashi, and Y. Kawahara. Barreto-naehrig curves. Network Working Group. Internet-Draft, February, 2014. Available online: https://tools.ietf.org/html/draft-kasamatsu-bncurves-01 (accessed on 30 October 2021). 58

[Kob91] Neal Koblitz. Elliptic curve implementation of zero-knowledge blobs. *J. Cryptology*, 4(3):207–213, 1991. 107

[Kos] Ahmed Kosba. xJsnark. GitHub. 19

[KP98] Joe Kilian and Erez Petrank. An efficient noninteractive zero-knowledge proof system for np with general assumptions. *Journal of Cryptology*, 11:1–27, 1998. 3

[KPS18] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 5 2018. 19

[KT08] Koray Karabina and Edlyn Teske. On prime-order elliptic curves with embedding degrees $k = 3, 4$, and 6. In *International Algorithmic Number Theory Symposium*, pages 102–117. Springer, 2008. 84

[KV19] Dmitry Khovratovich and Mikhail Vladimirov. Tornado Privacy Solution. Cryptographic Review. Version 1.1. ABDK Consulting, November 29, 2019. 23

[KXV] Koninklijke Philips N.V., Glenn Xavier, and Meilof Veeningen. pysnark. GitHub. 18, 19

[Laba] Electron Labs. Bringing IBC to Ethereum using zk-SNARKs. EthResearch. 40

[Labb] Matter Labs. The Zinc language. GitHub. 19, 20

[Len87] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Ann. of Math. (2)*, 126(3):649–673, 1987. 108

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January, 2016. 63, 64

[LM87] Peter L. Montgomery. Montgomery, p.l.: Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation - Math. Comput.*, 48:243–243, 01 1987. 56, 60, 66

[LMS17] B. Libert, F. Mouhartem, and D. Stehlé. Tutorial 8, 1016-17. Notes from the Master Course Cryptology and Security at the École Normale Supérieure de Lyon. 42, 56, 76

[Mat19] Matter Labs. Zinc v0.2.3. Cryptology ePrint Archive, Report 2019/953, 2019. 19, 20

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society, 2013. 15, 55

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005. 19

[Mig83] Adolph Migotti. Zur Theorie der Kreisteilungsgleichung. *B. der Math.-Naturwiss, Classe der Kaiserlichen Akademie der Wissenschaften, Wien*, 87:7–14, 1883. 93

[MJ16] Nadia El Mrabet and Marc Joye. *Guide to Pairing-Based Cryptography*. Chapman & Hall/CRC, 2016. 13

[MNT01] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 84(5):1234–1243, 2001. 59, 84, 86, 87, 89

[MOV93] Alfred J Menezes, Tatsuaki Okamoto, and Scott A Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *iEEE Transactions on information Theory*, 39(5):1639–1646, 1993. 84, 86

[MSZ21] Simon Masson, Antonio Sanso, and Zhenfei Zhang. Bandersnatch: a fast elliptic curve built over the bls12-381 scalar field. Cryptology ePrint Archive, Report 2021/1152, 2021. Available online: https://ia.cr/2021/1152 (accessed on 30 October 2021). 58

[MV73] Hugh Lowell Montgomery and Robert Charles Vaughan. The large sieve. *Mathematika*, 20(2):119–134, 1973. 106

[NML16] Shen Noether, Adam Mackenzie, and The Lab. Ring confidential transactions. *Ledger*, 1:1–18, 12 2016. 55

[NT16] Assa Naveh and Eran Tromer. PhotoProof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 255–271. IEEE, 2016. 81

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2248–2266. IEEE, 2022. 16, 19, 20, 21, 22, 53

[OCWS] Alex Ozdemir, Edward Chen, Riad S. Wahby, and Northrim William Seo. CirC: The circuit compiler. GitHub. 20, 21

[OKS00] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. Elliptic curves with the montgomery-form and their cryptographic applications. In *Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '00, pages 238–257, London, UK, UK, 2000. Springer-Verlag. 59

[oL] o1 Labs. snarky. GitHub. 19, 20

[Peg] Ed Jr. Pegg. Bouniakowsky conjecture. *MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein.* 87

[Pepa] Pepper Project. Pequin: An end-to-end toolchain for verifiable computation, snarks, and probabilistic proofs. GitHub. 20

[Pepb] Pepper Project. tinyram. GitHub. 19, 20, 40

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013. Best Paper Award. 4, 8, 10, 12, 16, 28, 33, 82

[Qi] QED-it. zkInterface, a standard tool for zero-knowledge interoperability. GitHub. 21

[Sal] Xavier Salleras. ZPiE: Zero-knowledge proofs in embedded systems. GitHub. 18, 19

[Sch96] Bruce Schneier. *Applied Cryptography*. Wiley, 2nd edition, 1996. 2

[SD21] Xavier Salleras and Vanesa Daza. ZPiE: Zero-knowledge proofs in embedded systems. *Mathematics*, 9(20), 2021. 18, 19

[Sil94] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, New York, 1994. 85, 86, 89

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013. 2

[Sma99] Nigel P. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of cryptology*, 12:193–196, 1999. 82

[SS11] Joseph H Silverman and Katherine E Stange. Amicable pairs and aliquot cycles for elliptic curves. *Experimental Mathematics*, 20(3):329–357, 2011. 83, 90, 91

[ST16] Sushil Kumar Singh and Sudeep Tanwar. Analysis of software testing techniques: Theory to practical approach. *Indian Journal of Science and Technology*, 9, 08 2016. 72

[Sti] D. R. Stinson. On the connections between universal hashing, combinatorial designs and error-correcting codes. 45

[Suc] Succinct Computational Integrity and Privacy Research (SCIPR) Lab. libsnark: a C++ library for zk-SNARK proofs. GitHub. 17, 19

[Sut12] Andrew V Sutherland. Accelerating the CM method. *LMS Journal of Computation and Mathematics*, 15:172–204, 2012. 87

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. 1

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2008. 81

[WBB19] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Generation of twisted edwards elliptic curves for circuit use. Accepted Community Standard Proposal at ZK Proof Workshop 2, 2019. 58

[WBB20] Barry WhiteHat, Marta Bellés, and Jordi Baylina. Baby Jubjub elliptic curve. Ethereum Improvement Proposal, EIP-2494, January 29, 2020. 10, 42, 55, 57

[Whi18] Barry WhiteHat. Baby jubjub supporting evidence. GitHub, 2018. Available online: https://github.com/barryWhiteHat/baby_jubjub (accessed on 30 October 2021). 72

[WLG+] Barry WhiteHat, Chih Cheng Liang, Kobi Gurkan, Koh Wei Jie, and Harry Robert. Semaphore. 23

[WSR+15] Riad Wahby, Srinath Setty, Zuocheng Ren, Andrew Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium (NDSS 2015)*, 02 2015. 19, 20, 40

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. 18

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. Cryptology ePrint Archive, Report 2021/730, 2021. 18

[YCKS19] S. Yonezawa, S. Chikara, T. Kobayashi, and T. Saito. Pairing-friendly curves. Network Working Group. Internet-Draft, January, 2019. Available online: https://tools.ietf.org/html/draft-yonezawa-pairing-friendly-curves-00 (accessed on 30 October 2021). 58

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Report 2021/076, 2021. 18

[zcaa] Electric Coin Company. What is Jubjub? Available online: https://z.cash/technology/jubjub/ (accessed on 30 October 2021). 56, 58

[Zcab] Zcash. Bellman. GitHub. 17, 19

[Zer] ZeroPool. Privacy solution for blockchain. 23

[zkR] zkREPL. An online playground for zero–knowledge circuits. 23

[ZoKa] ZoKrates. Zokrates. GitHub. 19

[ZoKb] ZoKrates. ZoKrates documentation. GitHub Docs. 22, 23

Appendix A

Code

Talk is cheap. Show me the code.

– Linus Torvalds

A.1 Code from Chapter 4

A.1.1 Implementation of security tests from Section 4.5

```
# It outputs all results in the console

import os
import sys
from errno import ENOENT, EEXIST
from sortedcontainers import SortedSet

def readfile(fn):
  fd = open(fn,'r')
  r = fd.read()
  fd.close()
  return r

# Expresses n as sums or differences of sparse powers of 2 (if possible)
def expand2(n):
  s = ""

  while n != 0:
    j = 1
    while 2 * *j < abs(n): j += 1
    if 2 * *j - abs(n) > abs(n) - 2 * *(j-1): j -= 1
```

```
22
23      if abs(abs(n) - 2**j) > 2**(j - 1):
24        if n > 0:
25          if s != "": s += " + "
26          s += str(n)
27        else:
28          s += " - " + str(-n)
29        n = 0
30      elif n > 0:
31        if s != "": s += " + "
32        s += "2^" + str(j)
33        n -= 2**j
34      else:
35      s += " - 2^" + str(j)
36      n += 2**j
37
38    return s
39
40  def verify(curve):
41
42    p = Integer(readfile(curve+'/p')) # Prime p
43    k = GF(p) # Finite field F_p.
44    kz.<z> = k[] # Polynomial ring k[z].
45    l = Integer(readfile(curve+'/l')) # Large prime l dividing |E(F_p)|
46    x0 = Integer(readfile(curve+'/x0')) # (x0,y0): generating point of E
47    y0 = Integer(readfile(curve+'/y0'))
48    x1 = Integer(readfile(curve+'/x1')) # (x1,y1): base point of E[l]
49    y1 = Integer(readfile(curve+'/y1'))
50    shape = readfile(curve+'/shape').strip()
51    s = readfile(curve+'/primes')
52    rigid = readfile(curve+'/rigid').strip()
53
54    safefield = True
55    safeeq = True
56    safebase = True
57    saferho = True
58    safetransfer = True
59    safedisc = True
60    saferigid = True
61    safeladder = True
62    safetwist = True
63    safecomplete = True
64    safeind = True
65
66    V = [] # Distinct verified primes.
67    for line in s.split():
68      n = Integer(line)
69      if n.is_prime(): # Instead of generating the original Pocklington
       primality proofs
70        if not n in V: V += [n]
71
72    # Verify p is prime.
73    pstatus = 'Unverified'
74    if not p.is_prime(): pstatus = 'False'
75    if p in V: pstatus = 'True'
76    if pstatus != 'True': safefield = False
77    print('verify-pisprime: %s\n' %pstatus)
78
```

```
79  # Verify l is prime.
80  pstatus = 'Unverified'
81  if not l.is_prime(): pstatus = 'False'
82  if l in V: pstatus = 'True'
83  if pstatus != 'True': safebase = False
84  print('verify-lisprime: %s\n' %pstatus)
85
86  # Write l and p as sums or differences of sparse powers of 2 (if
      possible)
87  print('expand2-p: p = %s\n' % expand2(p))
88  print('expand2-l: l = %s\n' % expand2(l))
89
90  # Write the variables in base 16
91  print('hex-p:  %s' %hex(p))
92  print('hex-l:  %s' %hex(l))
93  print('hex-x0: %s' %hex(x0))
94  print('hex-x1: %s' %hex(x1))
95  print('hex-y0: %s' %hex(y0))
96  print('hex-y1: %s\n' %hex(y1))
97
98  # Verify gcd(l,p) = 1 (else, if l=p -> DL easy via additive transfers)
99  gcdlpis1 = gcd(l,p) == 1
100 print('verify-gcdlp1: %s\n' %gcdlpis1)
101
102 # Verify if embedding degree is large (else, multiplicative transfers (
      or MOV attacks) are easy)
103   # Embedding degree: smallest integer k such that l divides (p^k-1)
104     # It could also be computed (it takes longer): k = (Integers(l)(p))
      .multiplicative_order()
105   # Brainpool and SafeCurves require embedding degree > (l-1)/100
106     # [Balasubramin, Koblitz] MOV is subexponential if k < (log(p))^2
107 print('verify-movsafe: Unverified')
108 print('verify-embeddingdegree: Unverified')
109 if gcdlpis1 and l.is_prime():
110   u = Integers(l)(p)
111   d = l-1
112   for v in V:
113     while d % v == 0: d /= v
114   if d == 1:
115     d = l-1
116     for v in V:
117       while d % v == 0:
118         if u^(d/v) != 1: break
119         d /= v
120     print('verify-movsafe: %s' %((l-1)/d <= 100) )
121     print('verify-embeddingdegree: %s = (l-1)/%s\n' % (d,(l-1)/d))
122
123 # Compute the Frobenius trace t (it should satisfy |E(F_p)| = p + 1 - t
      )
124   # Hasse's theorem: |t|<2 * sqrt(p)
125 # Also compute the cofactor such that |E(F_p)| = cofactor  * l
126   # If E is Montgomery curve, the cofactor has to be a multiple of 4
127 t = p+1-l * round((p+1)/l)
128 if l^2 > 16 * p:
129   print('verify-trace: %s' % t)
130   f = factor(1)
131   d = (p+1-t)/l
132   for v in V:
```

```
133       while d % v == 0:
134         d //= v
135       f  *= factor(v)
136     print('verify-cofactor: %s\n' % f)
137   else:
138     print('verify-trace: Unverified')
139     print('verify-cofactor: Unverified\n')
140
141   # Compute the complex-multiplication field discriminant D:
142     # Let s^2 be the largest square dividing t^2-4p. Then (t^2-4p)/s^2 is
        a squarefree negative integer.
143     # If (t^2-4p)/s^2 mod 4 = 1, then D = (t^2-4p)/s^2.
144     # Otherwise, D = 4(t^2-4p)/s^2.
145   # Verify D is big: SafeCurves requires |D|>2^100.
146   D = t^2-4*p
147   for v in V:
148     while D % v^2 == 0: D /= v^2
149   if prod([v for v in V if D % v == 0]) != -D:
150     print('verify-disc: Unverified')
151     print('verify-discisbig: Unverified')
152     safedisc = False
153   else:
154     f = -prod([factor(v) for v in V if D % v == 0])
155     if D % 4 != 1:
156     D  *= 4
157     f = factor(4)  *  f
158     Dbits = (log(-D)/log(2)).numerical_approx()
159     print('verify-disc: %s = %s; -2^%.1f' % (D,f,Dbits))
160     print('verify-discisbig: %s\n' %(D < -2^100))
161
162   # Verify that the cost of the Pollard's rho attack is above 2^100
163   pi4 = 0.78539816339744830961566084581987572105
164   rho = log(pi4*l)/log(4)
165   print('verify-rho: %.1f' % rho)
166   print('verify-rhoabove100: %s\n' %(rho.numerical_approx() >= 100))
167
168   # Verify security against twist attacks
169   twistl = 'Unverified'
170   d = p+1+t
171   for v in V:
172     while d % v == 0: d /= v
173   if d ==1:
174     d = p+1+t
175     for v in V:
176       if d % v == 0:
177         if twistl == 'Unverified' or v > twistl: twistl = v
178
179   print('verify-twistl: %s\n' % twistl)
180   print('verify-twistmovsafe: Unverified')
181   print('verify-twistembeddingdegree: Unverified\n')
182   if twistl == 'Unverified':
183     print('hex-twistl: Unverified\n')
184     print('expand2-twistl: Unverified\n')
185     print('verify-twistcofactor: Unverified\n')
186     print('verify-gcdtwistlp1: Unverified\n')
187     print('verify-twistrho: Unverified\n')
188     safetwist = False
189   else:
```

```
190     print('hex-twistl: %s\n' %hex(twistl))
191     print('expand2-twistl: %s\n' % expand2(twistl))
192     f = factor(1)
193     d = (p+1+t)/twistl
194     for v in V:
195       while d % v == 0:
196         d //= v
197         f  *= factor(v)
198     print('verify-twistcofactor: %s\n' % f)
199     gcdtwistlpis1 = gcd(twistl,p) == 1
200     print('verify-gcdtwistlp1: %s\n' %gcdtwistlpis1)
201
202     movsafe = 'Unverified'
203     embeddingdegree = 'Unverified'
204     if gcdtwistlpis1 and twistl.is_prime():
205       u = Integers(twistl)(p)
206       d = twistl-1
207     for v in V:
208       while d % v == 0: d /= v
209     if d == 1:
210       d = twistl-1
211       for v in V:
212         while d % v == 0:
213           if u^(d/v) != 1: break
214           d /= v
215       print('verify-twistmovsafe: %s' %((twistl-1)/d <= 100))
216       print('verify-twistembeddingdegree: %s = (l^-1)/%s\n' % (d,(twistl
      -1)/d))
217
218     rho = log(pi4 * twistl)/log(4)
219     print('verify-twistrho %.1f' % rho)
220     print('verify-twistrhoabove100: %s\n' %(rho.numerical_approx() >=
      100))
221
222     precomp = 0
223     joint = 1
224     for v in V:
225       d1 = p+1-t
226       d2 = p+1+t
227       while d1 % v == 0 or d2 % v == 0:
228         if d1 % v == 0: d1 //= v
229         if d2 % v == 0: d2 //= v
230         # best case for attack: cyclic; each power is usable
231           # also assume that kangaroo is as efficient as rho
232         if v + sqrt(pi4 * joint/v) < sqrt(pi4 * joint):
233           precomp += v
234           joint /= v
235
236     rho = log(precomp + sqrt(pi4  *  joint))/log(2)
237     print('verify-jointrho: %.1f' % rho)
238     print('verify-jointrhoabove100: %s\n' %(rho.numerical_approx() >=
      100))
239
240   x0 = k(x0)
241   y0 = k(y0)
242   x1 = k(x1)
243   y1 = k(y1)
244
```

```
245   # Verify if the equation defines and elliptic curve
246   # Verify the shape of the elliptic curve
247   # Verify both points [x0,y0] and [x1,y1] are on the curve
248   if shape in ('edwards', 'tedwards'):
249     d = Integer(readfile(curve+'/d'))
250     a = 1
251     if shape == 'tedwards':
252       a = Integer(readfile(curve+'/a'))
253
254     print('verify-shape: Twisted Edwards')
255     print('verify-equation: %sx^2+y^2 = 1%+dx^2y^2\n' % (a, d))
256     if a == 1:
257       print('verify-shape: Edwards')
258       print('verify-equation: x^2+y^2 = 1%+dx^2y^2\n' % d)
259
260     a = k(a)
261     d = k(d)
262     elliptic = a * d * (a-d)
263     level0 = a * x0^2+y0^2-1-d * x0^2 * y0^2
264     level1 = a * x1^2+y1^2-1-d * x1^2 * y1^2
265
266   if shape == 'montgomery':
267     print('verify-shape: Montgomery')
268     A = Integer(readfile(curve+'/A'))
269     B = Integer(readfile(curve+'/B'))
270     if B == 1: print('verify-equation: y^2 = x^3%s+dx^2+x\n' %A)
271     else: print('verify-equation: %sy^2 = x^3%s+dx^2+x\n' %(B,A))
272
273     A = k(A)
274     B = k(B)
275     elliptic = B * (A^2-4)
276     level0 = B * y0^2-x0^3-A * x0^2-x0
277     level1 = B * y1^2-x1^3-A * x1^2-x1
278
279   if shape == 'shortw':
280     print('verify-shape: short Weierstrass')
281     a = Integer(readfile(curve+'/a'))
282     b = Integer(readfile(curve+'/b'))
283     print('verify-equation: y^2 = x^3%s+dx%s+d\n' % (a,b))
284
285     a = k(a)
286     b = k(b)
287     elliptic = 4 * a^3+27 * b^2
288     level0 = y0^2-x0^3-a * x0-b
289     level1 = y1^2-x1^3-a * x1-b
290
291   print('verify-elliptic: %s' %str(elliptic)) # discriminant
292   print('verify-iselliptic: %s' %(elliptic != 0)) # if eq. defines an
        elliptic curve
293   print('verify-isoncurve0: %s' %(level0 == 0)) # if generating point is
        on the curve
294   print('verify-isoncurve1: %s\n' %(level1 == 0)) # if base point is on
        the curve
295
296   # Transform an Edwards or a twised Edwards curve to a Montgomery curve
297   if shape in ('edwards', 'tedwards'):
298   A = 2 * (a+d)/(a-d)
299   B = 4/(a-d)
```

```
300   x0,y0 = (1+y0)/(1-y0),((1+y0)/(1-y0))/x0
301   x1,y1 = (1+y1)/(1-y1),((1+y1)/(1-y1))/x1
302   shape = 'montgomery'
303
304   # Transform a Montgomery curve to a short Weierstrass
305   if shape == 'montgomery':
306     a = (3-A^2)/(3*B^2)
307     b = (2*A^3-9*A)/(27*B^3)
308     x0,y0 = (x0+A/3)/B,y0/B
309     x1,y1 = (x1+A/3)/B,y1/B
310     shape = 'shortw'
311
312   try:
313     E = EllipticCurve([a,b])
314     numorder2 = 0
315     numorder4 = 0
316     for P in E(0).division_points(4):
317       if P != 0 and 2*P == 0:
318       numorder2 += 1
319       if 2*P != 0 and 4*P == 0:
320       numorder4 += 1
321     print('verify-numorder2: %s' %str(numorder2))
322     print('verify-numorder4: %s\n' %str(numorder4))
323
324     # Verify completeness
325     completesingle = False
326     completemulti = False
327     if numorder4 == 2 and numorder2 == 1:
328       # Complete Edwards and Montgomery with unique point of order 2
329       completesingle = True
330       completemulti = True
331     # Should extend this to allow complete twisted hessian
332     print('verify-completesingle: %s' %completesingle)
333     print('verify-completemulti: %s\n' %completemulti)
334
335     print('verify-ltimesbase1is0: %s' %(l * E([x1,y1]) == 0))
336     print('verify-ltimesbase1: %s\n' %(str(l * E([x1,y1]))) )
337
338     print("verify-cofactorbase01: it can not be done as I do not have z0.
        ")
339     print('verify-cofactorbase01: %s\n' %(str(((p+1-t)//l) * E([x0,y0])
        == E([x1,y1]))))
340   except:
341     print('verify-numorder2: Unverified')
342     print('verify-numorder4: Unverified\n')
343
344     print('verify-ltimesbase1: Unverified')
345     print('verify-cofactorbase01: Unverified\n')
346     safecomplete = False
347
348   # Verify monladder
349   montladder = False
350   for r,e in (z^3+a*z+b).roots():
351     if (3*r^2+a).is_square():
352       montladder = True
353     print('verify-montladder: %s' %montladder)
354
355   # Verify indistinguishability
```

```
356   indistinguishability = False
357   elligator2 = False
358   if (p+1-t) % 2 == 0:
359     if b != 0:
360       indistinguishability = True
361       elligator2 = True
362   print('verify-indistinguishability: %s' %indistinguishability)
363   print('verify-ind-notes: Elligator 2: %s\n' % (['No','Yes'][elligator2
        ]))
364
365   # Verify rigidity (by reading the file "rigid")
366   saferigid &= (rigid == 'fully rigid' or rigid == 'somewhat rigid')
367
368   safecurve = True
369   print('verify-safefield: %s' %safefield)
370   print('verify-safeeq: %s' %safeeq)
371   print('verify-safebase: %s' %safebase)
372   print('verify-saferho: %s' %saferho)
373   print('verify-safetransfer: %s' %safetransfer)
374   print('verify-safedisc: %s' %safedisc)
375   print('verify-saferigid: %s' %saferigid)
376   print('verify-safeladder: %s' %safeladder)
377   print('verify-safetwist: %s' %safetwist)
378   print('verify-safecomplete: %s' %safecomplete)
379   print('verify-safeind: %s' %safeind)
```

A.2 Code from Chapter 5

A.2.1 Setup

`MNT3()`, `MNT4()`, `MNT6()`, `Freeman()`, `BN()`

These functions return the set of polynomials that define the families of curves MNT3, MNT4, MNT6, Freeman, and BN, respectively.

The expected outputs are:

- `t`: polynomial $t(X) \in \mathbb{Q}[X]$ that parameterizes the trace.
- `p`: polynomial $p(X) \in \mathbb{Q}[X]$ that parameterizes the order of the curves.
- `q`: polynomial $q(X) \in \mathbb{Q}[X]$ that parameterizes the order of the finite field over which the curve is defined.

```
# SETUP

# Polynomial rings over the reals and rationals.
R.<X> = PolynomialRing(RR, 'X')
Q.<X> = PolynomialRing(QQ, 'X')

# Curve families.
def MNT3():
    t = Q(6 * X -1)
    q = Q(12 * X^2 - 1)
    p = q + 1 - t
    return(t, p, q)

def MNT4():
    t = Q(-X)
    q = Q(X^2 + X + 1)
    p = q + 1 - t
    return(t, p, q)

def MNT6():
    t = Q(2 * X + 1)
    q = Q(4 * X^2 + 1)
    p = q + 1 - t
    return(t, p, q)

def Freeman():
    t = Q(10 * X^2 + 5 * X + 3)
    q = Q(25 * X^4 + 25 * X^3 + 25 * X^2 + 10 * X + 3)
    p = q + 1 - t
    return(t, p, q)

def BN():
    t = Q(6 * X^2 + 1)
```

```
    q = Q(36 * X^4 + 36 * X^3 + 24 * X^2 + 6 * X + 1)
    p = q + 1 - t
    return(t, p, q)
```

A.2.2 Auxiliary functions

`is_integer_valued(g)`

This function checks whether a given polynomial `g` is integer-valued. It returns `True` if so, and `False` otherwise. The test is based on the fact that a polynomial $g \in \mathbb{Q}[X]$ is integer-valued if and only if $g(x) \in \mathbb{Z}$ for $\deg g + 1$ consecutive $x \in \mathbb{Z}$ [CC16, Corollary 2].

```
def is_integer_valued(g):

# Check if evaluation is integer in deg(g) + 1 consecutive points.
for x in range(g.degree()+1):
if (not g(x) in ZZ):
print(str(g) + " is not integer-valued.")
return False
return True
```

`find_relevant_root(w, b, side)`

This function finds the left-most or right-most root of a polynomial $b(X) \in \mathbb{Q}[X]$.

The expected inputs are:

- `w`: positive integer.
- `b`: polynomial $b(X) \in \mathbb{Q}[X]$.
- `side`: this parameter specifies which root to keep. If `side = -1`, then the function takes the left-most root, and if `side` $= 1$, it returns the right-most root.

The expected output is the relevant extremal root.

```
def find_relevant_root(w, b, side):
# Decide whether to keep the left-most or right-most root.
i = -(1 + side) / 2
# 0 <= w(x)
C_1 = 0
w_roots = R(w).roots()
```

```
    if (w_roots != []):
    C_1 = w_roots[i][0]
    # w(x) < b(x)
    C_2 = 0
    bw_roots = R(b - w).roots()
    if (bw_roots != []):
    C_2 = bw_roots[i][0]
    # Return the relevant extremal root.
    if (side == -1):
    return ceil(min(C_1, C_2))
    else:
    return floor(max(C_1, C_2))
```

`check_embedding_degree(px, qx, k)`

This function determines whether k is the smallest positive integer such that $(\mathtt{px}^k - 1) \pmod{\mathtt{qx}} = 1$, and outputs `True/False`.

```
def check_embedding_degree(px, qx, k):
# Checks divisibility condition
if ((px^k - 1) % qx != 0): return False
# Checks that divisibility conditions does not happen for smaller
   exponents
div = divisors(k)
div.remove(k)
for j in div:
if ((px^j - 1) % qx == 0):
return False
return True
```

A.2.3 Code for Proposition 5.17

`candidate_embedding_degrees(Family, K_low, K_high)`

Given a family of curves, this function computes the possible embedding degrees of curves that may form 2-cycles with a curve of the given family.

The expected inputs are:

- `Family`: a polynomial parameterization $(t(X), p(X), q(X))$ of a family of pairing-friendly elliptic curves with prime order.
- `K_low, K_high`: lower and upper bounds on the embedding degree to look for.

The expected outputs are:

- `embedding_degrees`: a list of potential embedding degrees k such that `K_low` $\leq k \leq$ `K_high` and a curve from the family *might* form a cycle with a curve with embedding degree k.

- `modular_conditions`: conditions on x mod k for each of these k.

```
def candidate_embedding_degrees(Family, K_low, K_high):

    (t, p, q) = Family()
    # Create an empty list to store the candidate embedding degrees
    embedding_degrees = []
    # Create an empty list to store the lists of modular conditions for
     each k
    modular_conditions = [None] * (K_high + 1)

    # Embedding degree k implies that q(x) = 1 (mod k).
    # We check this condition in 0, ..., k-1 and build a list of
     candidates
    # such that any x has to be congruent to one of them modulo k.
    for k in range(K_low, K_high + 1):

        candidate = False

        for i in range(k):
            if ((q(i) % k) == 1):
                # First time a candidate k is discovered, add it to the
     list and
                # create a list within modular_conditions to store the
     values i.
                if (not candidate):
                    candidate = True
                    embedding_degrees.append(k)
                    modular_conditions[k] = []
                modular_conditions[k].append(i)

    return embedding_degrees, modular_conditions
```

A.2.4 Code for Table 5.2

`compute_bounds(a, b)`

This function computes the bounds $N_{\mathsf{left}}, N_{\mathsf{right}}$ of Lemma 5.20. This function has been used to produce the results of tables from Table 5.2. It uses the auxiliary functions from Appendix A.2.2.

The expected inputs are:

- `a, b`: two integer-valued polynomials in $\mathbb{Q}[X]$.

The expected outputs are:

- `N_left, N_right`: integer bounds $N_{\mathsf{left}}, N_{\mathsf{right}}$ described in Lemma 5.20.

```
def compute_bounds(a, b):

    # Check that b has even degree and positive leading coefficient
    if (b.degree() % 2 == 1 or b.leading_coefficient() < 0):
        print("Invalid divisor.")
        return

    # Check that a, b are integer valued.
    if (not is_integer_valued(a) or not is_integer_valued(b)):
        return

    # Polynomial division
    (h, r) = a.quo_rem(b)

    # Compute c so that ch, cr are in Z[X]
    denominators = [i.denominator() for i in (h.coefficients() + r.
     coefficients())]
    c = lcm(denominators)

    # Compute signs
    sigma_right = sign(r.leading_coefficient())
    sigma_left = sigma_right * (-1)^(r.degree())

    # We compute the polynomials w_left, w_right such that
    # 0 <= w_left < b(x) for all x < N_left, and
    # 0 <= w_right < b(x) for all x > N_right.
    w_left = c * r + ((1 - sigma_left) / 2) * b
    w_right = c * r + ((1 - sigma_right) / 2) * b

    # Compute N_left, N_right
    N_left = find_relevant_root(w_left, b, -1)
    N_right = find_relevant_root(w_right, b, 1)

    return (N_left, N_right)
```

A.2.5 Code for Corollary 5.22

`exhaustive_search(Family, k, N_left, N_right, mod_cond)`

This function performs the exhaustive search from Corollary 5.22 within the intervals $[N_{\mathsf{left}}, N_{\mathsf{right}}]$.

The expected inputs are:

- `Family`: a polynomial parameterization $(t(X), p(X), q(X))$ of a family of pairing-friendly elliptic curves with prime order.
- `k`: an embedding degree.
- `N_left, N_right`: upper and lower integer bounds.
- `mod_cond`: conditions on x mod `k` for every x in the interval [`N_left, N_right`].

The expected output is:

- `curves`: a list of curve descriptions $(x, k, t(x), p(x), q(x))$ such that $x \in$ [`N_left, N_right`], and the curve parameterized by $(t(x), p(x), q(x))$ forms a cycle with a curve with embedding degree `k`.

```
def exhaustive_search(Family, k, N_left, N_right, mod_cond):

  (t, p, q) = Family()
  curves  = []

  for x in range(N_left, N_right+1):
      # We skip those values that will never yield q(x) = 1 (mod k), as
    precomputed above.
      if (not (x % k) in mod_cond): continue
      # Check the embedding degree condition
      if (check_embedding_degree(p(x), q(x), k)):
          curves.append((x, k, t(x), p(x), q(x)))

  return curves
```

A.2.6 Main function

`search_for_cycles(Family, K_low, K_high)`

This function looks for 2-cycles formed by a curve belonging to a given parameterized family of curves and a prime-order curve with an embedding degree between two given bounds.

The expected inputs are:

- `Family`: a polynomial parameterization $(t(X), p(X), q(X))$ of a family of pairing-friendly elliptic curves with prime order.
- `K_low, K_high`: integer lower and upper bounds on the embedding degree to look for.

The function prints to a file all 2-cycles involving a curve from the family and a prime-order curve with embedding degree $\texttt{K\_low} \leq k \leq \texttt{K\_high}$.

```
import time

def search_for_cycles(Family, K_low, K_high):

    file_name = 'output_' + Family.__name__ + '.txt'
    f = open(file_name, 'w')
    start = time.time()

    # Instantiate the family
    (t, p, q) = Family()
    print("Starting family: " + str(Family.__name__), file=f)
    print("t(X) = " + str(t), file=f)
    print("p(X) = " + str(p), file=f)
    print("q(X) = " + str(q), file=f)

    # Find the candidate embedding degrees up to K that are compatible with this family
    (embedding_degrees, modular_conditions) = candidate_embedding_degrees(Family, K_low, K_high)
    print("Candidate embedding degrees: " + str(embedding_degrees), file=f)
    for k in embedding_degrees:
        print(("For k = " + str(k) + ", necessarily x = " +str(modular_conditions[k])) + " (mod " + str(k) + ")", file=f)
    print("==================", file=f)

    # For each potential embedding degree, find the bounds N_left, N_right and perform exhaustive search within [N_left, N_right].
    for k in embedding_degrees:

        f.close()
        f = open(file_name, 'a')
        start_k = time.time()

        print("k = " + str(k), file=f)
        (N_left, N_right) = compute_bounds(p^k, q)
        print("N_left = " + str(N_left) + ", N_right = " + str(N_right), file=f)

        curves = exhaustive_search(Family, k, N_left, N_right, modular_conditions[k])
        print("Curves with embedding degree " + str(k) + " that form a cycle with a curve from the " + str(Family.__name__) + " family: " + str(len(curves)), file=f)

        for curve in curves:
            (x, k, tx, px, qx) = curve
            print("x = " + str(x), file=f)
            print("embedding degree = " + str(k), file=f)
            print("t(x) = " + str(tx), file=f)
            print("p(x) = " + str(px), file=f)
            print("q(x) = " + str(qx), file=f)
```

```
44              print("------------", file=f)
45
46          end_k = time.time()
47          print('Computations for embedding degree ' + str(k) + ' took',
        round(end_k - start_k, 2), 'seconds.', file=f)
48          print("------------------------", file=f)
49
50      end = time.time()
51      print("==================", file=f)
52      print('Overall computation took', round(end - start, 2), 'time', file
        =f)
53
54      f.close()
```

Appendix B

Publications

Journals

2020 R. Genés-Durán, D. Yarlequé-Ruesta, M. Bellés-Muñoz, A. Jimenez-Viguer, and J.L. Muñoz-Tapia, "An architecture for easy onboarding and key life-cycle management in blockchain applications", in *IEEE Access*, vol. 8, pp. 115005-115016, doi: 10.1109/ACCESS.2020.3003995.

Abstract. New manufacturing paradigms require a large number of business interactions between multiple cyber-physical systems with different owners. In this context, public distributed ledgers are disruptive because they make it possible to securely and publicly record proofs of agreements between parties that do not necessarily trust each other. Many industry leaders have already achieved significant business benefits using this technology, including greater transparency, improved traceability, enhanced security, increased transaction speed and costs reduction. While the benefits of blockchain technologies for industrial applications are unquestionable, these technologies have an inherent complexity that might be overwhelming for many users. To decrease entry barriers for industry users to distributed ledger technologies, it is necessary to have an easy user onboarding process and a simple key life-cycle management. In this paper, we propose an architecture that facilitates these processes and simplifies how users utilize decentralized applications without sacrificing on the expected security. To achieve this goal, our architecture uses a middle-

ware that allows us to decouple the digital signatures required for paying blockchain fees from the ones required for authorization. This approach has the advantage that users are not forced to create wallets, buy cryptocurrency, or protect their private keys. For these reasons, our solution is a promising way of enabling a reasonable transition to the integration of distributed ledger technologies in industrial business processes.

2021 R. Genés-Durán, J. Hernández-Serrano, O. Esparza, M. Bellés-Muñoz, J.L. Muñoz-Tapia, "DEFS — Data exchange with free sample protocol", in *Electronics*, vol. 10, no. 12: 1455, doi: 10.3390/electronics10121455.

Abstract. Distrust between data providers and data consumers is one of the main obstacles hampering the take-off of digital-data commerce. Data providers want to get paid for what they offer, while data consumers want to know exactly what they are paying for before actually paying for it. In this article, we present a protocol that overcomes this obstacle by building trust based on two main ideas. First, a probabilistic verification protocol, where some random samples of the real dataset are shown to buyers in order to allow them to make an assessment before committing any payment; and second, a guaranteed, protected payment process enforced with smart contracts on a public blockchain that guarantees the payment of data if and only if the provided data meet the agreed terms, and that honest players are otherwise refunded.

2022 M. Bellés-Muñoz, B. Whitehat, J. Baylina, V. Daza, and J.L. Muñoz-Tapia, "Twisted Edwards elliptic curves for zero-knowledge circuits", in *Mathematics*, vol. 9, no. 23: 3022, doi: 10.3390/math9233022.

Abstract. Circuit-based zero-knowledge proofs have arose as a solution to the implementation of privacy in blockchain applications, and to current scalability problems that blockchains suffer from. The most efficient circuit-based zero-knowledge proofs use a pairing-friendly elliptic curve to generate and validate proofs. In particular, the circuits are built connecting wires that carry elements from a large prime field, whose order is determined by the number of elements of the pairing-friendly elliptic curve. In this context, it is important to generate an inner curve using this field, because it allows to create circuits that can verify public-key cryptogra-

phy primitives, such as digital signatures and encryption schemes. To this purpose, in this article, we present a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We also provide an algorithm for checking the resilience of this type of curve against most common security attacks. Additionally, we use our algorithms to generate Baby Jubjub, a curve that can be used to implement elliptic-curve cryptography in circuits that can be validated in the Ethereum blockchain.

2022 M. Bellés-Muñoz, J. Baylina, V. Daza and J. L. Muñoz-Tapia, "New privacy practices for blockchain software", in *IEEE Software*, vol. 39, no. 3, pp. 43-49, doi: 10.1109/MS.2021.3086718.

Abstract. In this article, we present the software tools we have implemented to bring complex privacy technologies closer to developers and to facilitate the implementation of privacy-enabled blockchain applications.

2022 M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio and J. Baylina, "CIRCOM: A circuit description language for building zero-knowledge applications", in *IEEE Transactions on Dependable and Secure Computing* (early access), doi: 10.1109/TDSC.2022.3232813.

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

a library for generating and validating ZK proofs from R1CS. Altogether, our software tools abstract the complexity of ZK proving mechanisms and provide a unique and friendly interface to model low-level descriptions of arithmetic circuits.

Conference proceedings

2022 E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, A. Rubio, "Distilling constraints in zero-knowledge protocols", in *Computer Aided Verification: 34th International Conference, CAV 2022*, Proceedings, Part I. Lecture Notes in Computer Science, vol 13371, pp. 430–443, Springer-Verlag, Berlin, Heidelberg, doi: 10.1007/978-3-031-13185-1_21.

Abstract. The most widely used zero-knowledge (ZK) protocols require provers to prove they know a solution to a computational problem expressed as a rank-1 constraint system (R1CS). An R1CS is essentially a system of non-linear arithmetic constraints over a set of signals, whose security level depends on its non-linear part only, as the linear (additive) constraints can be easily solved by an attacker. Distilling the essential constraints from an R1CS by removing the part that does not contribute to its security is important, not only to reduce costs (time and space) of producing the ZK proofs, but also to reveal to cryptographic programmers the real hardness of their proofs. In this paper, we formulate the problem of distilling constraints from an R1CS as the (hard) problem of simplifying constraints in the realm of non-linearity. To the best of our knowledge, it is the first time that constraint-based techniques developed in the context of formal methods are applied to the challenging problem of analysing and optimizing ZK protocols.

2023 M. Bellés-Muñoz, J. Jiménez Urroz, and J. Silva, "Revisiting cycles of pairing-friendly curves". To appear in *Crypto'23*.

Abstract. A recent area of interest in cryptography is recursive composition of proof systems. One of the approaches to make recursive composition efficient involves cycles of pairing-friendly elliptic curves of prime order. However, known constructions have very low embedding degrees.

This entails large parameter sizes, which makes the overall system inefficient. In this paper, we explore 2-cycles composed of curves from families parameterized by polynomials, and show that such cycles do not exist unless a strong condition holds. As a consequence, we prove that no 2-cycles can arise from the known families, except for those cycles already known. Additionally, we show some general properties about cycles, and provide a detailed computation on the density of pairing-friendly cycles among all cycles.

Preprints and other publications

2020 B. Whitehat, M. Bellés, J. Baylina, "ERC-2494: Baby Jubjub elliptic curve", *Ethereum Improvement Proposals (EIP)*, no. 2494.

2022 L. Pearson, J. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J.L. Muñoz-Tapia, "PlonKup: Reconciling PlonK with plookup", available in *Cryptology ePrint Archive*, paper 2022/086.

Abstract. In 2019, Gabizon, Williamson, and Ciobotaru introduced PlonK, a fast and flexible ZK-SNARK with an updatable and universal structured reference string. PlonK uses a grand product argument to check permutations of wire values, and exploits convenient interactions between multiplicative subgroups and Lagrange bases. The following year, Gabizon and Williamson used similar techniques to develop plookup, a ZK-SNARK that can verify that each element from a list of queries can be found in a public lookup table. In this paper, we present PlonKup, a fully succinct ZK-SNARK that integrates the ideas from plookup into PlonK in an efficient way.

2022 M. Bellés-Muñoz and V. Daza, "Chaum blind signature scheme", in *Encyclopedia of Cryptography, Security and Privacy (3rd Ed.)*, Springer-Verlag, Berlin, Heidelberg, doi: 10.1007/978-3-642-27739-9_1752-1.

2022 M. Bellés-Muñoz and V. Daza, "Self-sovereign identity", in *Encyclopedia of Cryptography, Security and Privacy (3rd Ed.)*, Springer-Verlag, Berlin, Heidelberg, doi: 10.1007/978-3-642-27739-9_1752-1.