UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Elliptic Curves, Cryptographic Proofs and Signature Schemes: A Comprehensive Analysis of the Pinocchio Protocol and Implementation Frameworks

Autor: Claudia Llamas Gómez

Director: Dr. Luis Victor Dieulefait
Codirector: Carla Brugulat Rica
Barcelona, January 15, 2025

Contents

Abstract

This paper investigates the mathematical and cryptographic principles underlying elliptic curves and their critical role in modern secure communication systems. It begins with an exploration of elliptic curves and their group properties, emphasizing their utility in cryptography through problems like the discrete logarithm. The discussion transitions to the Pinocchio protocol, a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK), detailing its use of Rank-1 Constraint Systems (R1CS), Quadratic Arithmetic Programs (QAPs), and cryptographic commitments. The final section delves into practical implementations, employing CIRCOM for circuit definition and testing, with specific applications to ECDSA and BLS signature schemes. Through this comprehensive approach, the paper bridges theoretical cryptographic advancements with practical implementation, offering insights into the efficiency, security, and application of elliptic curve-based protocols.

2020 Mathematics Subject Classification. 11G05, 11G10, 14G10

Acknowledgments

I would like to thank my family and friends, both those who have been with me throughout this journey and those I have had the pleasure of meeting along the way, inside and outside the context of mathematics. Their support and company have made these years much easier, and even turned long days at the library into enjoyable moments. A special thanks to Athena for always believing in me and for her diary patience and encouragement over the past five years.

Exploring the world of mathematics has been a journey full of surprises, often rewarding and at times frustrating, but I am truly glad to have taken it.

Finally, I would like to thank ESELEOS for giving me the opportunity to work on such an engaging topic. I am especially grateful to Marc Guzmán for his guidance throughout this project and to Jose Luis Muñoz for all the contents he has taught me.

Introduction

The development of secure communication systems has been driven by the dual forces of advancing mathematical theory and technological innovation. Among the fundamental pillars of modern cryptography, Elliptic Curve Cryptography (ECC) stands out for its ability to provide strong security guarantees with high efficiency. By exploiting the algebraic properties of elliptic curves, ECC achieves levels of security comparable to classical systems such as RSA, but with significantly smaller key sizes. This efficiency has made it a cornerstone of secure digital communication in the era of ubiquitous connectivity.

This thesis focuses on the theoretical and practical aspects of elliptic curve cryptographic systems. It begins by exploring the mathematical foundations of elliptic curves, emphasizing their group structure and applications in cryptography, particularly in the construction of efficient and secure protocols. The discussion then moves on to the Pinocchio protocol, a cryptographic proof system that exploits the properties of elliptic curves to provide concise computational verification. Finally, the paper looks at practical implementations using CIRCOM, highlighting its use in defining and testing circuits for cryptographic schemes such as ECDSA and BLS signatures.

Through this analysis, the thesis underscores the critical role of elliptic curves in modern cryptography and demonstrates how their theoretical foundations translate into practical tools for securing digital interactions. By combining mathematical rigor with computational feasibility, this work provides insights into the design and application of elliptic curve-based cryptographic protocols.

Chapter 1

Elliptic Curves

Elliptic curves are mathematical objects of profound significance in fields such as number theory, algebraic geometry, and cryptography. In particular, elliptic curve cryptography (ECC) has become a cornerstone of modern cryptographic systems, offering strong security guarantees while maintaining computational efficiency. The power of ECC lies in its ability to leverage the mathematical properties of elliptic curves, particularly their group structure, to construct secure cryptographic protocols. By relying on the computational infeasibility of problems like the elliptic curve discrete logarithm problem, ECC achieves security levels comparable to classical systems such as RSA, but with significantly smaller key sizes. To provide a rigorous foundation for the subsequent discussion of cryptographic applications, this first chapter explores some mathematical theory of elliptic curves.

Most of the contents in this chapter is primarily based on the material from Lawrence C. Washington's *Elliptic Curves: Number Theory and Cryptography* (2008) [Was08].

1.1 Definition

Definition 1.1 (Elliptic Curve)**.** *An elliptic curve E is the graph of an equation of the form*

$$E : y^2 = x^3 + Ax + B,$$

*where A and B are constants. The previous equation is referred to as the **short Weierstrass form** for an elliptic curve.*

An elliptic curve is required to be **non-singular**. To avoid singularities we must ensure that there are no multiple roots. More specifically, if the roots of the cubic are r_1, r_2, r_3, then the discriminant is

$$((r_1 - r_2)(r_1 - r_3)(r_2 - r_3))^2 = -(4A^3 + 27B^2)$$

Therefore, we must add the following condition to our definition of elliptic curve:

$$4A^3 + 27B^2 \neq 0.$$

We will need to specify which field A, B, x and y belong to. In our case they will be taken to be elements of a finite field $\mathbb{F}_p$ for a prime p or of a finite field $\mathbb{F}_q$ where $q = p^k$ with $k \geqslant 1$.

In general, if K is a field with $A, B \in K$, then we say that E is defined over K and we denote the points of this elliptic curve by the following way:

$$E(K) = \{\infty\} \cup \{(x, y) \in K \times K \mid y^2 = x^3 + Ax + B\}.$$

Note that we can not draw elliptic curves over most fields but, for intuition, it is useful to think draw them over the field of real numbers $\mathbb{R}$. These have two basic forms, depicted in Figure 1.1.

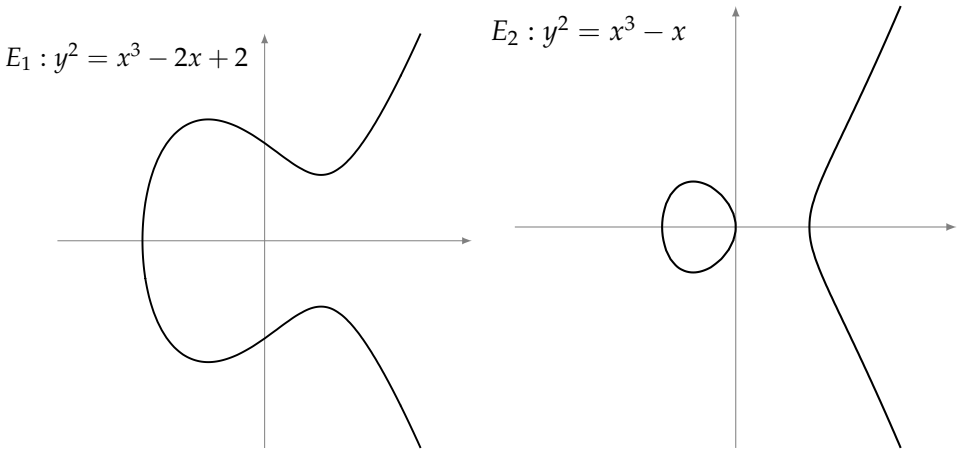

Figure 1.1: Examples of Elliptic Curves on $\mathbb{R}$.

1.2 The Group Law

First, we describe a inner operation between points of the elliptic curve, which is commonly known as **addition** and, consequently, denoted by $+$. This addition differs from simply adding their coordinates. Consider $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \in E(K)$. The sum $P_1 + P_2$ is determined as follows: First, draw the line that intersects P_1 and P_2. This will generally intersect the curve E at a third point, namely P_3'. Reflect P_3' across the x-axis changing the sign of the y-coordinate to obtain another point P_3. We will define the addition of the points P_1 and P_2 to be the point P_3.

Let us now describe the coordinates of P_3 depending on the coordinates of both P_1 and P_2. Let's take a look at the different cases. We will assume that $P_1, P_2 \neq \mathcal{O}$.

Basic case: $P_1 \neq P_2$ **and** $x_1 \neq x_2$**.** This is visually depicted in Figure 1.2. Let $\ell : y = m(x - x_1) + y_1$ be the line that crosses P_1 and P_2. We substitute y in the

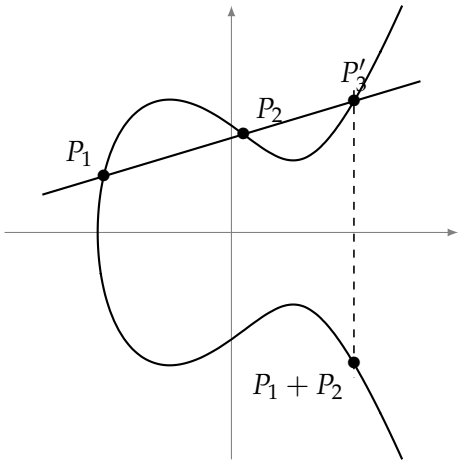

Figure 1.2: Basic case of adding points on a Elliptic Curve.

curve equation in order to find the intersection of the line with E:

$$(m(x - x_1) + y_1)^2 = x^3 + Ax + B.$$

This can be rearranged to the form $0 = x^3 - m^2x^2 + \cdots$. The roots of this cubic are the 3 points of intersection of ℓ with E and we already know two of them: x_1, x_2, since P_1 and P_2 are points on both ℓ and E. If we have a cubic polynomial $x^3 + ax^2 + bx + c$ with roots x_1, x_2, x_3, then:

$$x^3 + ax^2 + bx + c = (x - x_1)(x - x_2)(x - x_3) = x^3 - (x_1 + x_2 + x_3)x^2 + \cdots .$$

That is, the x^2 therm coincide with the sum of the roots in negative. Therefore, $x_1 + x_2 + x_3 = -a$. Since we already know x_1 and x_2, we can express x_3 as $x_3 = a - x_1 - x_2$. Then, we compute $P_3 = P_1 + P_2 = (x_3, y_3)$ with the following formula.

$$x_3 = m^2 - x_1 - x_2, \quad y_3 = m(x_1 - x_3) - y_1, \quad \text{where } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

In certain cases, the line passing through two points on the curve may fail to intersect a third point on the curve. This issue arises particularly when the line is vertical or tangent to the curve. To overcome this limitation and define the operation consistently, it is necessary to introduce the **point at infinity**. It is easiest to regard it as a point $\mathcal{O}$, sitting at **the top and the bottom** of the y-axis. Namely, we think of the ends of the y-axis as wrapping around and meeting in the point $\mathcal{O}$. By incorporating the point at infinity, the curve acquires a well-defined operation applicable to all scenarios, which will allow us to define the group law. The extension from the affine plane to the projective plane will be further developed in Section 1.4.

Different points $P_1 \neq P_2$ with $x_1 = x_2$. If $x_1 = x_2$, but $y_1 \neq y_2$, the line trough P_1 and P_2 is vertical, which therefore intersect E in $\mathcal{O}$. Reflecting $\mathcal{O}$ across the x-axis yields the same point $\mathcal{O}$ (this is why we put $\mathcal{O}$ at the top and the bottom of the y-axis). In this case, $P_1 + P_2 = \mathcal{O}$ as we can observe in Figure 1.3.

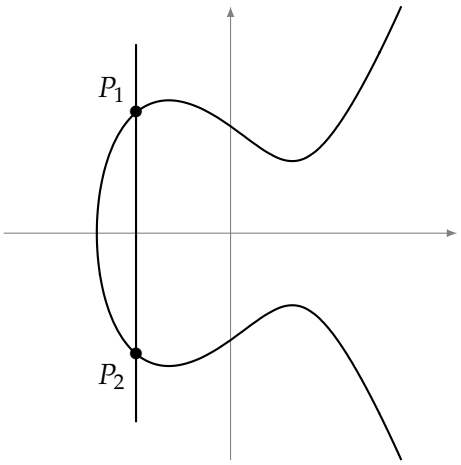

Figure 1.3: Adding points on a Elliptic Curve where $P_1 \neq P_2$ but $x_1 = x_2$.

Doubling: $P_1 = P_2$. Now consider the case where $P_1 = P_2 = (x_1, y_1)$. When two points on a curve are very close to each other, the line through them approximates a tangent line (see Figure 1.4). Therefore, when the two points coincide, we take the line ℓ through them to be the tangent line. Implicit differentiation allows us to find the slope m of ℓ:

$$2y\frac{dy}{dx} = 3x^2 + A, \quad \text{so } m = \frac{dy}{dx} = \frac{3x_1^2 + A}{2y_1}.$$

If $y_1 = 0$ then the line is vertical and we set $P_1 + P_2 = \mathcal{O}$, as before. Note that if $y_1 = 0$, then the numerator $3x_1^2 + A \neq 0$, excluding the case $0/0$. This is because the curve has to remain non-singular. If $3x_1^2 + A = 0$ and $y_1 = 0$, then both partial derivatives of the curves defining equation vanish simultaneously:

$$\frac{\partial F}{\partial x} = 3x^2 + A = 0, \quad \frac{\partial F}{\partial y} = 2y = 0.$$

This would indicate a singular point at (x_1, x_2), compromising the smoothness condition of an elliptic curve.

Therefore, assume that $y_1 \neq 0$. The equation of ℓ is

$$y = m(x - x_1) + y_1,$$

as before. We obtain the cubic equation

$$0 = x^3 - m^2x^2 + \cdots .$$

This time, we know only one root, namely x_1, but it is a double root since ℓ is tangent to E at P_1. Therefore, proceeding as before, we obtain

$$x_3 = m^2 - 2x_1, \quad y_3 = m(x_1 - x_3) - y_1.$$

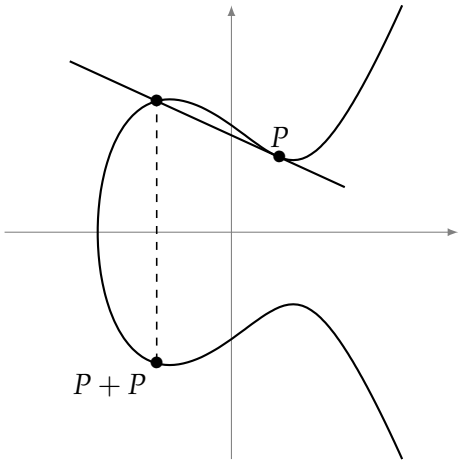

Figure 1.4: Adding points on a Elliptic Curve where $P_1 = P_2$.

Edge Case: $P_1 = \mathcal{O}$ or $P_2 = \mathcal{O}$. Let's suppose $P_2 = \mathcal{O}$. The line through P_1 and $\mathcal{O}$ is a vertical line that intersects E in, P_1' the reflection of P_1 across the x-axis, as showed in Figure 1.5. When we reflect P_1' across the x-axis to get $P_3 = P_1 + P_2$, we are back at P_1. Therefore,

$$P_1 + \mathcal{O} = P_1$$

for all points P_1 on E. Of course, we extend this to include the case $\mathcal{O} + \mathcal{O} = \mathcal{O}$.

Le's summarize all this:

Definition 1.2 (The Group Law)**.** *Let E be an elliptic curve defined by $y^2 = x^3 + Ax + B$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E with $P_1, P_2 \neq \mathcal{O}$. Define $P_3 = P_1 + P_2 = (x_3, y_3)$ as follows:*

1. *If $x_1 \neq x_2$, then:*

$$x_3 = m^2 - x_1 - x_2, \quad y_3 = m(x_1 - x_3) - y_1, \quad \text{where } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

2. *If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = \mathcal{O}$.*

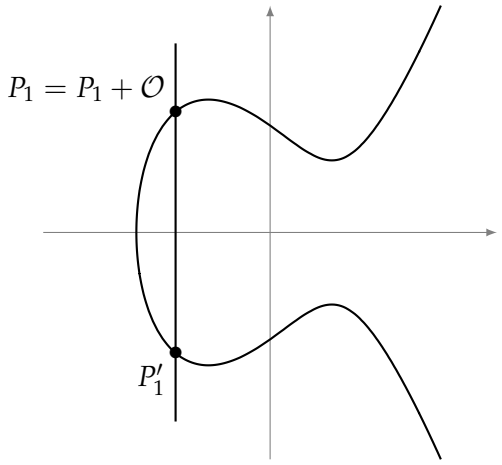

Figure 1.5: Adding points on a Elliptic Curve where $P_2 = \mathcal{O}$.

3. *If* $P = P_1 = P_2 = (x_1, y_1)$ *and* $y_1 \neq 0$, *then:*

$$x_3 = m^2 - 2x_1, \quad y_3 = m(x_1 - x_3) - y_1, \quad \text{where } m = \frac{3x_1^2 + A}{2y_1}.$$

4. *If* $P = P_1 = P_2 = (x_1, y_1)$ *and* $y_1 = 0$, *then* $P + P = \mathcal{O}$.

Moreover, define $P + \mathcal{O} = P$ *for all points* P *on* E.

We now have a well-defined operation to build the group law:

Theorem 1.3. *The addition of points on an elliptic curve* E *satisfies the following properties:*

1. $P_1 + P_2 \in E$ *for all* $P_1,\ P_2 \in E$. *(closure)*
2. $P_1 + P_2 = P_2 + P_1$ *for all* $P_1,\ P_2$ *on* E. *(commutativity)*
3. $P + \mathcal{O} = P$ *for all points* P *on* E. *(existence of identity)*
4. *Given* P *on* E, *there exists* P' *on* E *such that* $P + P' = \mathcal{O}$. *(existence of inverses) This point* P' *is usually denoted as* $-P$.
5. $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ *for all* $P1,\ P2,\ P3$ *on* E. *(associativity)*

In other words, the points on E *form an* ***additive abelian group*** *with* $\mathcal{O}$ *as the identity element.*

Proof. The commutativity is obvious, either from the formulas or from the fact that the line through P_1 and P_2 is the same as the line through P_2 and P_1. The identity property of $\mathcal{O}$ holds by definition. For inverses, let P' be the reflection of P across the x-axis. Then $P + P' = \mathcal{O}$.

Finally, we need to prove associativity. This is by far the most subtle and non obvious property of the addition of points on E. It is possible to define many laws of composition satisfying $(1), (2), (3)$ for points on E, either simpler or more complicated than the one being considered. But it is very unlikely that such a law will be associative. In fact, it is rather surprising that the law of composition that we have defined is associative. After all, we start with two points P_1 and P_2 and perform a certain procedure to obtain a third point $P_1 + P_2$. Then we repeat the procedure with $P_1 + P_2$ and P_3 to obtain $(P_1 + P_2) + P_3$. If we instead start by adding P_2 and P_3, then computing $P_1 + (P_2 + P_3)$, there seems to be no obvious reason that this should give the same point as the other computation. The associative law can be verified by calculation with the formulas. There are several cases, depending on whether or not $P_1 = P_2$, and whether or not $P_3 = (P_1 + P_2)$, etc., and this makes the proof rather messy. However, we prefer a different approach, which we give in Section 2.4. □

Example 1.4. *1. On the curve with equation*

$$y^2 = \frac{x(x+1)(2x+1)}{6}$$

we have

$$(0,0) + (1,1) = \left(\frac{1}{2}, -\frac{1}{2}\right),$$
$$\left(\frac{1}{2}, -\frac{1}{2}\right) + (1,1) = (24, -70).$$

2. On the curve with equation

$$y^2 = x^3 - 25x$$

we have

$$2(-4,6) = (-4,6) + (-4,6) = \left(\frac{1681}{144}, -\frac{62279}{1728}\right),$$
$$(0,0) + (-5,0) = (5,0),$$
$$2(0,0) = 2(-5,0) = 2(5,0) = \mathcal{O}.$$

Scalar Multiplication. Every commutative group admits the structure of a $\mathbb{Z}$-module. This means that the operation of **scalar multiplication** is well-defined in these groups. Specifically, if G is a commutative group with an additive operation, and $P \in G$ is an element, then for any integer $n \in \mathbb{Z}$, the scalar multiplication nP

can be defined as the repeated addition of P to itself n times. This operation is expressed as:

$$nP = \underbrace{P + P + \cdots + P}_{n},$$

where n is a natural number. The group of points on an elliptic curve, being abelian, allows the definition of this operation.

At first glance, computing nP in this manner appears to require n successive additions. If the cost of a single addition is $\mathcal{O}(1)$, this direct approach would result in an algorithm with a complexity of $\mathcal{O}(n)$. However, such an algorithm is far from being optimal.

A more efficient algorithm for doing scalar multiplication is called **Double and Add**. The key idea is to express the scalar n in binary, double the point P iteratively and add P selectively based on the binary representation of n. This reduces the complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$.

More specifically, let us suppose that the binary representation of n is

$$(b_k b_{k-1} ... b_1 b_0)_2,$$

where $b_k \in \{0, 1\}$ and b_{k-1} is the most significant bit. Therefore

$$n = \sum_{i=0}^{k-1} b_i 2^i,$$

and n has k bits. First of all, let us define $Q = \mathcal{O}$. Then, for each bet b_i in the binary decomposition of n, starting from the most significant bit ($i = k-1, \ldots, 0$), we do the following. First, we double the Q point $Q \leftarrow 2Q$ and, if b_i equals to 1, we add P to the result $Q \leftarrow Q + P$. After processing all the bits of n, we output Q as the result of nP.

Example 1.5. Let's see it with an example. Suppose $n = 13$ and P is a point on the curve. The binary representation of 13 is $(1101)_2$.

1. Begin with $Q = \mathcal{O}$.
2. First bit, which is 1. We double Q (still $\mathcal{O}$) then add $P \implies Q = 2\mathcal{O} + P = P$.
3. Second bit, which is 1. We double $Q = 2P$ then add $P \implies Q = 3P$.
4. Third bit, which is 0. We double $\implies Q = 6P$
5. Fourth bit, which is 1. We double $Q = 12P$ then add $P \implies Q = 13P$, which is the final output, as expected.

Conversely, if we are working over a large finite field and are given points P and nP, it is very difficult to determine the value of n. This is called the **discrete logarithm problem** for elliptic curves and is the basis for the cryptographic applications.

1.3 Points on a Elliptic Curve

In fact, since we are doing cryptography, we are specifically interested on elliptic curves defined over finite fields $\mathbb{F}_q$ where $q = p^k$ and $q > 3$. This ones are fundamental in number theory and cryptography. These curves are described by the equation:

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_q \mid y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\},$$

where $\mathcal{O}$ is the point at infinity. A natural question arises: how many points lie on such curves?

Initial estimates provide simple bounds for the total number of points. Since each $x \in \mathbb{F}_q$ corresponds to at most two values of y, the number of points is less than $2q$, that is, $|E(\mathbb{F}_p)| < q^2$. However, more precise results can be obtained using the **Hasse-Weil Theorem**, which states that the number of points satisfies the inequality

$$|E(\mathbb{F}_q)| \in \left[q + 1 - 2\sqrt{q},\, q + 1 + 2\sqrt{q}\right].$$

Equivalently, this can be expressed as $|E(\mathbb{F}_q)| = q + 1 - t$, where t, known as the Frobenius Trace, satisfies $|t| \leqslant 2\sqrt{q}$.

To compute the exact number of points on an elliptic curve over a finite field, an efficient but complex method called the **Schoof Algorithm** is commonly used. For any point P on the curve, the order n of the point satisfies the relation

$$\underbrace{P + P + \cdots + P}_{n} = \mathcal{O}.$$

If the total number of points on the curve is denoted by n, that is, $|E(\mathbb{F}_q)| = n$, then Lagranges theorem implies $n = h \cdot r$, where r is the largest prime factor of n, and h is known as the cofactor.

In cryptographic applications, the security of elliptic curve systems depends on the size of the largest prime order subgroup. This requires r to be sufficiently large, typically $\gtrsim 256$ bits, to ensure resistance to cryptographic attacks. The careful design of elliptic curves with such properties is critical for secure implementations.

1.4 Projective Space

We all know that parallel lines meet at infinity. Projective space allows us to make sense out of this statement and also to interpret the point at infinity on an elliptic curve. Let K be a field. Two-dimensional projective space $\mathbb{P}^2_K$ over K is given by equivalence classes of triples (x, y, z) with $x, y, z \in K$ and at least one of x, y, z nonzero. Two triples (x_1, y_1, z_1) and (x_2, y_2, z_2) are said to be equivalent if there exists a nonzero element $\lambda \in K$ such that

$$(x_1, y_1, z_1) = (\lambda x_2, \lambda y_2, \lambda z_2).$$

Equivalently, the two dimensional projective space consists on all the lines through the origin of K^3. We write $(x_1, y_1, z_1) \equiv (x_2, y_2, z_2)$. The equivalence class of a triple only depends on the ratios of x to y to z. Therefore, the equivalence class of (x, y, z) is denoted $(x : y : z)$, called homogeneous coordinates.

If $(x : y : z)$ is a point with $z \neq 0$, then $(x : y : z) = (x/z : y/z : 1)$. These are the "finite" points in $\mathbb{P}^2_K$. However, if $z = 0$ then dividing by z should be thought of as giving $\mathcal{O}$ in either the x or y coordinate, and therefore the points $(x : y : 0)$ are called the *points at infinity* in $\mathbb{P}^2_K$. The point at infinity on an elliptic curve will soon be identified with one of these points at infinity in $\mathbb{P}^2_K$.

The two-dimensional affine plane over K is often denote $\mathbb{A}^2_K = \{(x, y) \in K \times K\}$. We have an inclusion

$$\mathbb{A}^2_K \hookrightarrow \mathbb{P}^2_K$$

given by

$$(x, y) \mapsto (x : y : 1).$$

In this way, the affine plane $\mathbb{A}^2_K$ is identified with the finite points in $\mathbb{P}^2_K$. Adding the points at infinity to obtain $\mathbb{P}^2_K$ can be viewed as a way of "compactifying" the plane.

Now lets look at the elliptic curve E given by $y^2 = x^3 + Ax + B$. Its homogeneous form is $y^2z = x^3 + Axz^2 + Bz^3$. The points (x, y) on the original curve correspond to the points $(x : y : 1)$ in the projective version. To see what points on E lie at infinity, set $z = 0$ and obtain $0 = x^3$. Therefore $x = 0$, and y can be any nonzero number (recall that $(0 : 0 : 0)$ is not allowed). Rescale by y to find that $(0 : y : 0) = (0 : 1 : 0)$ is the only point at infinity on E. As we saw above, $(0 : 1 : 0)$ lies on every vertical line, so every vertical line intersects E at this point at infinity. Moreover, since $(0 : 1 : 0) = (0 : -1 : 0)$, the "top" and the "bottom" of the y-axis are the same.

1.5 Pairings

Elliptic curve pairings, such as the Weil and Tate pairings, have become a cornerstone of modern cryptographic systems by enabling protocols that extend the possibilities of traditional elliptic curve cryptography. These bilinear maps, which combine elements from elliptic curve groups to produce values in a finite field, are the foundation of pairing-based cryptography. Their bilinearity and non-degeneracy allow for operations like efficient key agreement, identity-based encryption, and attribute-based encryption, providing tools to address problems that were previously intractable in cryptography.

To construct pairings, it is essential to have a well-defined structure on the elliptic curve, which is where torsion points and divisors become indispensable. Torsion points are required because pairings operate on elements of finite order, ensuring that the resulting map has the necessary algebraic properties. Divisors,

in turn, provide a framework for defining these maps rigorously, linking the arithmetic of points on the curve with the field elements they map to.

1.5.1 Torsion Points

Let E be an elliptic curve defined over a field K. Let n be a positive integer. We are interested in the set of points of order n of $E(\overline{K})$, where $\overline{K}$ denotes the algebraic closure of K. This set, denoted $E[n]$, is defined as:

$$E[n] = \{P \in E(\overline{K}) | nP = \mathcal{O}\}.$$

The point P is said to be a **torsion point**. The set $E[n]$ of torsion points form a finite subgroup of $E(\overline{K})$. These points are defined over $\overline{K}$ because their coordinates satisfy polynomial equations derived from the elliptic curve equation and the condition $[n](P) = \mathcal{O}$. These equations may have roots in algebraic extensions of K and not in K.

Theorem 1.6. *Let E be an elliptic curve over a field K and let n be a positive integer. If the characteristic of K does not divide n, or is 0, then*

$$E[n] \simeq \mathbb{Z}_n \oplus \mathbb{Z}_n$$

If the characteristic of K is $p > 0$ and $p|n$, write $n = p^r n'$ with $p \nmid n'$. Then

$$E[n] \simeq \mathbb{Z}_{n'} \oplus \mathbb{Z}_{n'} \quad or \quad \mathbb{Z}_n \oplus \mathbb{Z}_{n'}$$

Proof. This proof can be found in Section 3.2 of Washington's Elliptic Curves: Number Theory and Cryptography [Was08]. □

1.5.2 Divisors

Torsion points $P \in E[n]$ on an elliptic curve play a fundamental role in the construction of **divisors**. Specifically, they often appear as the support of divisors on the curve.

Let E be an elliptic curve over K. A divisor D is a formal sum of points on the curve:

$$D = \sum_{P \in E} n_P \cdot P, \quad n_P \in \mathbb{Z},$$

where only finitely many coefficients n_P are nonzero.

For a rational function f on E, its *principal divisor* is defined as:

$$\operatorname{div}(f) = \sum_{P \in E} v_P(f) \cdot P,$$

where $v_P(f)$ denotes the *order* of f at P:

- $v_P(f) > 0$: f has a **zero** of order $v_P(f)$ at P,

- $v_P(f) < 0$: f has a **pole** of order $|v_P(f)|$ at P,
- $v_P(f) = 0$: f is **regular** at P (neither zero nor pole).

The divisor of f encodes the points where f has zeros and poles, and its degree is always zero:

$$\deg(\operatorname{div}(f)) = \sum_{P \in E} v_P(f) = 0.$$

When a rational function f is constructed such that its zeros and poles lie entirely on the torsion points $E[n]$ of the curve, its associated divisor is said to be supported on the torsion points. Formally, such a divisor can be written as:

$$\operatorname{div}(f) = \sum_{P \in E[n]} v_P(f) \cdot P,$$

where $P \in E[n]$ and $v_P(f)$ is nonzero only for torsion points.

For example, the divisor associated with a torsion point $P \in E[n]$ can be expressed as:

$$D_P = P + (-P) - 2\mathcal{O}.$$

A divisor is called *symmetric* if it remains invariant under negation of the points. This property arises naturally with divisors supported on torsion points, as these points inherently satisfy the symmetry $P + (-P)$. For instance, a symmetric divisor associated with $P \in E[n]$ has the form:

$$D = P + (-P) - 2\mathcal{O}.$$

In order to be able to construct the Weil pairing in the next section, we need to state the following theorem:

Theorem 1.7. *Let E be an elliptic curve. Let D be a divisor on E with $deg(D) = 0$. Then there is a function f on E with*

$$div(f) = D$$

if and only if

$$sum(D) = \mathcal{O}.$$

1.5.3 The Weil Pairing

Pairings, such as the *Weil pairing* and the *Tate pairing*, leverage the structure of torsion points and divisors to construct bilinear maps that are fundamental to cryptographic protocols. These pairings are defined algebraically, using divisors and their associated functions, and geometrically, via properties of elliptic curves. In this paper I am going to focus on explaining the Weil Pairing.

The Weil pairing on the n-torsion on an elliptic curve is a major tool in the study of elliptic curves. For example, it is used to prove Hasses theorem on the

number of points on an elliptic curve over a finite field and to attack the discrete logarithm problemfor elliptic curves.

Let E be an elliptic curve over a field K and let n be an integer not divisible by the characteristic of K. Then $E[n] \simeq \mathbb{Z}_n \oplus \mathbb{Z}_n$. Let

$$\mu_n = \{x \in \overline{K} | x^n = 1\}$$

be the group of nth roots of unity in $\overline{K}$. Since the characteristic of K does not divide n, the equation $x^n = 1$ has no multiple roots, hence has n roots in $\overline{K}$. Therefore, μ_n is a cyclic group of order n. Any generator ξ of μ_n is called a **primitive nth root of unity**. This is equivalent to saying that $\xi^k = 1$ if and only if n divides k.

The goal of this section is to construct the Weil pairing. Recall that n is an integer not divisible by the characteristic of the field K, and that E is an elliptic curve such that $E[n] \subseteq E(K)$. We want to construct a pairing

$$e_n : E[n] \times E[n] \to \mu_n,$$

the assumption $E[n] \subseteq E(K)$ forces $\mu_n \subset K$.

Let $T \in E[n]$. By Theorem 1.7, there exists a function f such that

$$div(f) = n[T] - n[\mathcal{O}]$$

.

Choose $T \in E[n^2]$ such that $nT = T$. Well use Theorem 1.7 to show that there exists a function g such that

$$div(g) = \sum_{R \in E[n]} ([T + R] - [R]).$$

We need to verify that the sum of the points in the divisor is $\mathcal{O}$. This follows from the fact that there are n^2 points R in $E[n]$. The points R in $\sum[T + R]$ and $\sum[R]$ cancel, so the sum is $n^2T = nT = \mathcal{O}$. Note that g does not depend on the choice of T' since any two choices for T differ by an element $R \in E[n]$. Therefore, we could have written

$$dig(g) = \sum_{nT''=T} [T''] - \sum_{nR=\mathcal{O}} [R].$$

Let $f \circ n$ denote the function that starts with a point, multiplies it by n, then applies f. The points $P = T + R$ with $R \in E[n]$ are those points P with $nP = T$. It follows from Theorem 1.7 that

$$div(f \circ n) = n\left(\sum_R [T' + R]\right) - n\left(\sum_R [R]\right) = div(g^n).$$

Therefore, $f \circ n$ is a constant multiple of g^n. By multiplying f by a suitable constant, we may assume that

$$f \circ n = g^n.$$

Let $S \in E[n]$ and let $P \in E(K)$. Then

$$g(P+S)^n = f(n(P+S)) = f(nP) = g(P)^n.$$

Therefore, $g(P+S)/g(P) \in \mu_n$. In fact, $g(P+S)/g(P)$ is independent of P. The proof of this is slightly technical: In the Zariski topology, $g(P+S)/g(P)$ is a continuous function of P and E is connected. Therefore, the map to the finite discrete set μ_n must be constant.

Now we can define the **Weil pairing** by

$$e_n(S,T) = \frac{g(P+S)}{g(P)}$$

Since g is determined up to a scalar multiple by its divisor, this definition is independent of the choice of g. Note that the equation is independent of the choice of the auxiliary point P. The main properties of en are given in the following theorem:

Theorem 1.8. *Let E be an elliptic curve defined over a field K and let n be a positive integer. Assume that the characteristic of K does not divide n. Then, there is a pairing*

$$e_n : E[n] \times E[n] \to \mu_n,$$

*called the **Weil pairing** that satisfies the following properties:*

1. *e_n is bilinear in each variable. This means that*

   $$e_n(S_1+S_2,T) = e_n(S_1,T)e_n(S_2,T)$$

   and

   $$e_n(S,T_1+T_2) = e_n(S,T_1)e_n(S,T_2)$$

   for all $S, S_1, S_2, T, T_1, T_2 \in E[n]$.

2. *e_n in nondegenerated in each variable. This means that if $e_n(S,T) = 1$ for all $T \in E[n]$ then $S = \mathcal{O}$ and also that if $e_n(S,T) = 1$ for all $S \in E[n]$ then $T = \mathcal{O}$.*

3. *$e_n(T,T) = 1$ for all $T \in E[n]$.*

4. *$e_n(T,S) = en(S,T)^{-1}$ for all $S, T \in E[n]$.*

5. *$e_n(\sigma S, \sigma T) = \sigma(e_n(S,T))$ for all automorphisms σ of K such that σ is the identity map on the coefficients of E (if E is in Weierstrass form this means that $\sigma(A) = A$ and $\sigma(B) = B$).*

6. *$e_n(\alpha(S), \alpha(T)) = e_n(S,T)^{deg(\alpha)}$ for all separable endomorphisms of E. If the coefficients of E lie in a finite field $\mathbb{F}_q$, then the statement also holds when α is the Frobenius endomorphism ϕ_q.*

Chapter 2

Pinocchio Protocol

2.1 Probabilistic Proofs

For centuries, the foundation of mathematics has been the creation of rigorous proofs: clear and logical sequences of steps that leads to an undeniable conclusion. Traditionally, the verification of such proofs has relied on **deterministic algorithms**, which always produce the same output when given the same input. This deterministic approach underpins the concept of NP problems, a class of problems where solutions can be efficiently verified, even if finding those solutions is computationally challenging or infeasible with current resources. For example, if a statement claims that a graph contains a Hamiltonian cycle, the witness for this statement would be the cycle itself. Given the cycle, verifying its correctness is straightforward, even though it might require significant computational effort.

With the rise of probabilistic polynomial-time algorithms, randomisation has been introduced into the verification process, leading to faster methods with minimal trade-offs. This idea culminated in 1985, when Goldwasser, Micali, and Rackoff [GMR85] proposed the concept of **probabilistic proofs**. These proofs incorporate randomness into verification, enabling efficient checks with a bounded probability of error. While the idea of a verification process that is not always correct might seem counterintuitive, the error rate can be reduced to negligible levels, making such systems practically reliable.

Probabilistic proof systems must satisfy two key properties: **completeness**, meaning true statements are always provable, and **soundness**, meaning false statements cannot be proved except with a very small probability of error. By incorporating **randomness**, these systems relax the classical notion of mathematical proofs, where absolute certainty is replaced by near-certainty. To strengthen soundness, many systems also meet the proof of knowledge (PoW) requirement, which asserts that any prover capable of convincing the verifier of the truth of a statement (with a probability exceeding the soundness error) must know a valid witness for that statement.

One example of probabilistic proof systems is **Interactive Proofs** (IPs)[GMR85].

They allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a computation has been correctly executed. To do so, the prover and the verifier engage in a sequence of interactions by exchanging several messages. The verifier's messages are influenced by its internal randomness. This sequence of interactions is called the transcript. At the end of the exchange, also known as the protocol, the verifier decides whether to accept or reject the prover's claim, based on the transcript and its internal randomness.

Probabilistic proofs can also exhibit the property of **zero-knowledge** [GMR85], where a prover demonstrates the truth of a statement without revealing any additional information. For instance, a zero-knowledge proof could confirm that a person knows the private key corresponding to a public key without disclosing the key itself. Such systems, known as **Zero-Knowledge Proofs** (ZKPs), rely on reasonable complexity assumptions, such as the existence of one-way functions, to guarantee security. Research has shown that every NP problem can be equipped with a zero-knowledge proof system, enabling a wide range of practical applications [G+08].

Efficiency in proving systems is critical, focusing on reducing both communication costs and verifier computation. The concept of succinct proofs addresses these goals, where the size of the proof and the verifiers workload grow only logarithmically with the complexity of the statement. **Probabilistically Checkable Proofs** (PCPs) [FKST94, AS92] achieve succinctness by allowing the verifier to inspect only a few randomly chosen parts of the proof while maintaining high confidence in its validity. Although PCPs reduce communication costs, they impose significant computational demands on the prover, who must generate a large, structured proof for randomised verification.

In contrast to the statistical soundness of probabilistic proofs, argument systems focus on computational soundness, ensuring security against polynomial-time adversaries. Argument systems leverage cryptographic primitives, enabling properties like reusability and public verifiability, and can convert interactive protocols into non-interactive ones using the Fiat-Shamir transformation. These systems form the foundation of **Succinct Non-interactive Arguments** (SNARGs), which achieve succinctness and computational soundness. When coupled with zero-knowledge, they become zk-SNARGs, widely used in privacy-preserving applications like zk-SNARKs for blockchains. The literature has numerous instances of zk-SNARKs, such as those presented in [Gro16], [BGG+18], [GWC19], and [CHM+19]. Each of these instances is built upon unique cryptographic primitives and offers varying trade-offs.

One of the most prominent developments in this field is the **Pinocchio protocol**, which is a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) that enables efficient proof generation and verification for arbitrary computations. This section provides an overview of the protocol's steps and highlights its mathematical foundation.

2.2 Arithmetization

At the core of Pinocchio's efficiency lies the process of translating computations into a mathematical representation amenable to cryptographic proofs. This translation involves two crucial steps: the arithmetisation of **Rank-1 Constraint Systems** (R1CS) and their conversion into **Quadratic Arithmetic Programs** (QAPs), which serve as the basis for efficient proof generation and verification.

The first step, the arithmetisation of R1CS, expresses a computation as a set of linear constraints over a finite field. An R1CS consists of signals, which represent inputs, outputs, and intermediate variables of the computation, along with constraints in the form of polynomial equations. Each constraint is expressed as:

$$A(s) \cdot B(s) - C(s) = 0,$$

where $s = (s_1, s_2, ..., s_m)$ is the vector of all signals (or wires), and A, B, C are vectors defining the linear combinations of these signals. Specifically, the linear combinations $A(s), B(s)$ and $C(s)$ are computed as:

$$A(s) = \sum_{i=1}^{m} a_i s_i, \quad B(s) = \sum_{i=1}^{m} b_i s_i \quad \text{and} \quad C(s) = \sum_{i=1}^{m} c_i s_i$$

where a_i, b_i and c_i are coefficients that specify the R1CS constraints. These coefficients determine how the signals interact and define the relationships that must hold for the computation to be valid.

More explicitly, $\mathsf{s} = (s_1, \ldots, s_m)$ satisfies the constraints if and only if

$$\begin{cases} (a_{1,0} \cdot s_0 + \cdots + a_{1,m} \cdot s_m) \cdot (b_{1,0} \cdot s_0 + \cdots + b_{1,m} \cdot s_m) = \\ \qquad\qquad c_{1,0} \cdot s_0 + \cdots + c_{1,m} \cdot s_m \\ \qquad\qquad \cdots \\ (a_{n,0} \cdot s_0 + \cdots + a_{n,m} \cdot s_m) \cdot (b_{n,0} \cdot s_0 + \cdots + b_{n,m} \cdot s_m) = \\ \qquad\qquad c_{n,0} \cdot s_0 + \cdots + c_{n,m} \cdot s_m. \end{cases}$$

Based on the previous explanation, it is important to note that constraints containing isolated constants, such as the constraint $(s_1 + s_2) \cdot s_3 = 1$ are not permitted in the R1CS formulation. To address the inclusion of constants in constraints, a new signal s_0 is introduced, which is always set to a value of 1. Therefore, the former constraint can be expressed as

$$(s_1 + s_2) \cdot s_3 = 1 = s_0.$$

Each column of coefficients in A, B, C is interpolated into polynomials $a_i(X), b_i(X)$ and $c_i(X)$, where i represents the index of a signal. Interpolation is performed over a chosen domain, such as $\{1, 2, 3, \ldots, n\}$. This process results in polynomials that encode the relationships between signals. However, to optimise computational efficiency, the interpolation domain is replaced with a subgroup of the roots of unity

in a finite field. Roots of unity enable the use of the Fast Fourier Transform (also known as FFT) for efficient interpolation and evaluation. The domain is defined as a subgroup $\mathcal{H} = \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$, where ω is a primitive n-th root of unity in the chosen field.

With the polynomials $a_i(X), b_i(X)$ and $c_i(X)$, the Quadratic Arithmetic Program (QAP) identity is constructed as:

$$p(X) = \left(\sum_{i=0}^{m} a_i(X)s_i\right) \cdot \left(\sum_{i=0}^{m} b_i(X)s_i\right) - \left(\sum_{i=0}^{m} c_i(X)s_i\right).$$

This identity is valid only within the interpolation domain $\mathcal{H}$, meaning $p(X) = 0$ for all $X \in \mathcal{H}$.

For cryptographic purposes, the signals s are divided into public $(s_0, \ldots, s_\ell)$ and private $(s_{\ell+1}, \ldots, s_m)$ signals. The QAP identity is structured to ensure that the verifier can verify the correctness of the computation using only the public inputs. The private signals are exclusively possessed by the prover, who uses them to compute the necessary polynomials $a_i(X), b_i(X)$ and $c_i(X), i \in \{0, \ldots, m\}$. The design of the verification process ensures that the verifier does not need direct access to these private signals. Instead, the verifier evaluates the QAP identity by checking whether the prover's claimed results satisfy the required polynomial relationships, which inherently incorporate the private inputs. This separation of roles ensures privacy: the private signals influence the proof but remain concealed from the verifier, who only interacts with publicly available information and the polynomials derived by the prover.

Example 2.1. *Inside the field $\mathbb{F}_{41}$ consider the following system of R1CS constraints, for which the values of the satisfying signals are $s_1 = 5$, $s_2 = 3$, and $s_3 = 2$:*

$$\begin{cases} (2 \cdot s_1 + 0 \cdot s_2 + 1 \cdot s_3) \cdot (1 \cdot s_1 + 3 \cdot s_2 + 0 \cdot s_3) - (0 \cdot s_1 + 0 \cdot s_2 + 2 \cdot s_3) &= 0, \\ (1 \cdot s_1 + 3 \cdot s_2 + 2 \cdot s_3) \cdot (0 \cdot s_1 + 4 \cdot s_2 + 0 \cdot s_3) - (0 \cdot s_1 + 3 \cdot s_2 + 1 \cdot s_3) &= 0, \\ (0 \cdot s_1 + 7 \cdot s_2 + 2 \cdot s_3) \cdot (2 \cdot s_1 + 1 \cdot s_2 + 2 \cdot s_3) - (3 \cdot s_1 + 0 \cdot s_2 + 0 \cdot s_3) &= 0, \\ (3 \cdot s_1 + 2 \cdot s_2 + 0 \cdot s_3) \cdot (0 \cdot s_1 + 0 \cdot s_2 + 5 \cdot s_3) - (0 \cdot s_1 + 1 \cdot s_2 + 1 \cdot s_3) &= 0. \end{cases}$$

Interpolating the Polynomials *We start by interpolating the coefficients into polynomials over an initial domain $\{1, 2, 3, 4\}$. For example, the coefficients of $a_1(X)$ correspond to the points $(1, 2), (2, 1), (3, 0), (4, 3)$. Interpolation yields to the following polynomial:*

$$a_1(X) = 2X^3 + 29X^2 + 19X + 38.$$

Similarly, other polynomials $a_i(X), b_i(X), c_i(X)$ are constructed.

Switching to Roots of Unity *We have established that it is preferable to choose a group of roots of unity as the interpolation domain to enable FFT interpolation, which is significantly faster. To identify this domain, we first determine the primitive roots of unity*

for subgroups of, for example, $\mathbb{Z}_{41}$*, where* $p - 1 = 2^3 \cdot 5$*. It can be verified that* 6 *is a generator of* $\mathbb{Z}_{41}^*$ *since* $6^{(40/2)} \mod 41 = 40 \neq 1$ *and* $6^{(40/5)} \mod 41 = 10 \neq 1$*. Thus,* $\omega_4 = g^{\frac{p-1}{n}} = 6^{\frac{40}{4}} \mod 41 = 32$ *is a primitive* 4*th root of unity and serves as a generator for the subgroup* $\mathcal{H}$ *of order* 4*:*

$$\mathcal{H} = \{\omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3\} = \{1, 32, 40, 9\}.$$

Therefore, this subgroup $\mathcal{H}$ *is the interpolation domain of interest. So now the coefficients of* $a_0(X)$ *correspond to the points* $(1, 2), (32, 1), (40, 0), (9, 3)$*. Interpolation over* $\mathbb{Z}_{41}$ *yields to the following polynomial:*

$$a_1(X) = 5X^3 + 20X^2 + 37X + 22.$$

Constructing the QAP Identity *Using the satisfying signals* $s_1 = 5$*,* $s_2 = 3$*, and* $s_3 = 2$*, the prover computes:*

$$p(X) = \Big(a_1(X)s_1 + a_2(X)s_2 + a_3(X)s_3\Big) \cdot \Big(b_1(X)s_1 + b_2(X)s_2 + b_3(X)s_3\Big) - \Big(c_1(X)s_1 + c_2(X)s_2 + c_3(X)s_3\Big).$$

This results in:

$$p(X) = 38X^6 + 31X^5 + 3X^4 + 3X^2 + 10X + 38,$$

which evaluates to zero for all $X \in \mathcal{H}$*, confirming that the QAP identity holds.*

2.3 Random Evaluation of Polynomials

A cornerstone of Pinocchio's design is the handling of polynomials, which are used to encode computational constraints. This involves evaluating, encrypting, and committing to polynomials while ensuring their integrity and privacy. This section delves into the foundational concepts, including the Schwartz-Zippel lemma, random polynomial evaluations, encrypted commitments, and the trusted setup process.

The Schwartz-Zippel lemma is a result in algebra that bounds the probability of a polynomial evaluating to zero over randomly chosen inputs. Formally:

Lemma 2.2 (Schwartz-Zippel)**.** *Let* $p(x_1, \ldots, x_n)$ *be a non-zero multivariate polynomial of degree* d *over* $\mathbb{F}_p$*, then the probability of* $p(\alpha_1, \ldots, \alpha_n) = 0$ *for randomly chosen* $(\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}_p^n$ *is at most* d/p*.*

This lemma is particularly useful for polynomial equality testing. For example, given two single-variable polynomials $p_1(x)$ and $p_2(x)$, we can test whether $p_1(x) - p_2(x) = 0$ by evaluating at a random point $\alpha \in \mathbb{Z}_p$. If the polynomials are

identical, the equality holds for all inputs. Otherwise, the equality holds with a probability of at most $\max(d_1, d_2)/p$, where d_1 and d_2 are the degrees of $p_1(x)$ and $p_2(x)$.

In the context of Pinocchio, this lemma underpins the soundness of polynomial commitments, ensuring that any deviation from the committed polynomial can be detected with high probability.

Polynomial Commitment Polynomials in zk-SNARKs encode computational constraints, but verifying their validity requires representing them in a compact and secure way. One way to achieve this is by evaluating the polynomial at a random point.

Example 2.3. *For example, consider the polynomial:*

$$f(X) = 7X^3 + 2X^2 + X + 1.$$

Evaluating $f(X)$ at a randomly chosen point $X = \tau$ ensures the polynomial's properties can be tested without revealing its entire structure. However, if the random evaluation point τ is revealed to the prover, they could build a different polynomial $\tilde{f}(X) \neq f(X)$ that coincides with $f(X)$ at τ.

For instance, if $\tau = 15$ in $\mathbb{F}_{41}$, a malicious prover could construct:

$$\tilde{f}(X) = 27X^3 + 28X^2 + 17X + 6,$$

such that $f(15) = \tilde{f}(15) = 24$. This demonstrates the need for cryptographic protection in random evaluations.

To prevent tampering, the Pinocchio protocol employs elliptic curve cryptography (ECC) to encrypt polynomial evaluations. The protocol creates a cryptographic commitment to the polynomial, denoted as F, which securely represents the polynomial without revealing its structure.

For the polynomial $f(X) = 7X^3 + 2X^2 + X + 1$, the commitment is:

$$F = f(\tau) \cdot G_1 = (7\tau^3 + 2\tau^2 + \tau + 1) \cdot G_1,$$

where G_1 is a generator point on an elliptic curve. The challenge lies in enabling the prover to compute F without knowledge of τ. This is resolved by providing the prover with encrypted powers of τ, specifically:

$$\{H_0 = \tau^0 \cdot G_1, H_1 = \tau^1 \cdot G_1, H_2 = \tau^2 \cdot G_1, H_3 = \tau^3 \cdot G_1\}.$$

Using these encrypted powers, the prover computes:

$$F = 7H_3 + 2H_2 + H_1 + H_0.$$

This cryptographic commitment ensures the polynomial cannot be altered, while the secret τ remains secure.

The Trusted Setup The Trusted Setup generates the encrypted powers of τ, also known as *Structured Reference String (SRS)*, enabling commitments to polynomials of degree less than n.

In the trusted setup process, a trusted entity selects a random secret τ and computes the powers of τ within the elliptic curve group as follows: $H_0 = \tau^0 \cdot G_1$, $H_1 = \tau^1 \cdot G_1$, up to $H_n = \tau^n \cdot G_1$, where G_1 is a generator point on the curve. After generating the set $\{H_0, H_1, \ldots, H_n\}$, the trusted entity must securely discard τ, a value often referred to as the *toxic value* ensuring it is inaccessible to any party to preserve the system's security.

The Structured Reference String (SRS) possesses several key properties that make it essential for zk-SNARK protocols. Firstly, it is **universal**, meaning it can be used for any polynomial of degree less than n. This universality extends to supporting circuits with fewer than n R1CS constraints. Secondly, the SRS is **shared** among all participants, including provers and verifiers, and acts as a public resource for cryptographic commitments.

2.4 Knowledge of Commitment

In the Pinocchio protocol, the verifier seeks to verify the validity of a quadratic arithmetic program (QAP) commitment succinctly and without access to the private s_i values. Specifically, the verifier aims to ensure that the following equation holds:

$$\left(\sum_{i=0}^{\ell} a_i(X)s_i + \sum_{i=\ell+1}^{m} a_i(X)s_i\right) \cdot \left(\sum_{i=0}^{\ell} b_i(X)s_i + \sum_{i=\ell+1}^{m} b_i(X)s_i\right) - \left(\sum_{i=0}^{\ell} c_i(X)s_i + \sum_{i=\ell+1}^{m} c_i(X)s_i\right)\Bigg|_{X\in\mathcal{H}} = 0.$$

The challenge is that the s_i values for $i = \ell + 1$ to m are secret, meaning the prover cannot reveal them to the verifier. Instead, the goal is to demonstrate that the polynomial:

$$f(X) = \sum_{i=\ell+1}^{m} a_i(X)s_i$$

is correctly constructed as a linear combination of the public $a_i(X)$ polynomials, which define the topology of the circuit. This ensures the prover's adherence to the QAP's constraints without revealing private information.

Notation 2.4. *From now on the following notation is going to be used, where G_1 and G_2 are the group generators for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.*

$$[\cdot]_1 : \mathbb{Z} \to \mathbb{G}_1$$
$$z \mapsto z \cdot G_1.$$

$$[\cdot]_2 : \mathbb{Z} \to \mathbb{G}_2$$
$$z \mapsto z \cdot G_2.$$

The prover must first commit to the $a_i(X)$ polynomials, which are public, using the powers of τ generated during the trusted setup. These commitments, denoted A_i, are computed as:

$$A_i = a_i(\tau)G_1 = [a_i(\tau)]_1,$$

where G_1 is a generator of the chosen **elliptic curve group**. These A_i commitments are publicly available and are precomputed by the Trusted Setup during the circuit setup phase. The prover then uses these commitments to create the polynomial commitment π_A for $f(X)$, defined as:

$$\pi_A = \sum_{i=\ell+1}^{m} A_i s_i.$$

The verifier's task is to ensure that π_A is computed as a valid linear combination of the public commitments A_i, using a set of s_i values that the prover knows but does not reveal.

To validate π_A and ensure it represents a legitimate linear combination, the verifier employs a **"random shift" method**: the trusted setup generates an additional toxic random value α_A, which is used to enforce the knowledge of commitment. Specifically, the random shift transforms the commitments A_i into shifted commitments $A'_i = \alpha_A A_i$, provided during the trusted setup. The verifier checks the commitment π_A by comparing:

$$\alpha_A \left(\sum_{i=\ell+1}^{m} A_i s_i \right) = \sum_{i=\ell+1}^{m} A'_i s_i,$$

using an asymmetric pairing

$$e(\cdot, \cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T.$$

For this reason, the selected curve for performing this check must be pairing friendly. The pairing equation is expressed as:

$$e(\pi_A, [\alpha_A]_2) = e(\pi'_A, [1]_2),$$

where $\pi'_A = \sum_{i=\ell+1}^{m} A'_i s_i$ and $[\alpha_A]_2$ is the random shift encrypted in the second elliptic curve group $\mathbb{G}_2$, also provided by the trusted setup. This ensures that the prover's commitment π_A corresponds to a valid combination of the A_i commitments.

Soundness of the Random Shift The soundness of the random shift approach lies in its prevention of unauthorised commitments. A malicious prover would need to produce a valid pairing check for a commitment like:

$$\pi_A = A_1 s_1 + A_2 s_2 + D s_3,$$

where D is a point not derived from the A_i commitments. The verifier detects this because the trusted setup does not include a shifted version of D, and the prover cannot compute $D' = \alpha_A D$ without knowledge of D in terms of the public A_i.

This technique is known as the "knowledge of exponent argument" (KEA), ensuring that the prover can only produce a valid commitment by knowing the exponents s_i used in the linear combination of A_i.

Risks of Toxic Randomness The trusted setup must securely delete the toxic values τ and α_A. If these values are compromised, an adversary could forge valid commitments. For example:

a) Knowing α_A allows an arbitrary point D to be combined into a valid commitment $D' = \alpha_A D$, bypassing the linear combination restriction.

b) Knowing τ, allows computing $a_i(\tau)^{-1}$ and from A'_1, which is $a_i(\tau)\alpha_A G_1$. From here, $a_i(\tau)^{-1} A'_1 = \alpha_A G_1 = [\alpha_A]_1$ can be computed, and finally, generate an arbitrary point $D = dG_1$ and compute $D' = d[\alpha_A]_1$ and again pass the pairing check without actually committing to a linear combination of A_i.

This highlights the critical importance of securely removing toxic randomness during the trusted setup.

Per-Circuit Trusted Setup The random shift method requires a trusted setup specific to each circuit. This setup comprises two phases:

The first phase is the **Universal Phase**. The trusted setup generates the powers of τ, applicable to any polynomial of degree less than n. The second phase is the **Circuit-Specific Phase**, where the shifted commitments $\{A'_i, B'_i, C'_i\}$ are generated for the specific circuit, based on its polynomials. This means that if the circuit changes, the second phase of the setup must be re-executed.

Multi-Party Computation (MPC) To mitigate the risks of toxic randomness, trusted setups are often implemented as Multi-Party Computations (MPCs), commonly referred to as “ceremonies”. In an MPC, multiple participants collectively generate the toxic values, ensuring that no single party possesses them entirely. Additionally, powers of τ must be created for both elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$.

2.5 Proving the QAPs

Recall that Pinocchio is a **Probabilistically Checkable Proof (PCP)**, which is a proof that can be verified by a randomised algorithm using a limited amount of randomness and checking a limited number of bits of the proof. Formally:

Definition 2.5 (PCP)**.** *A PCP is a proof that can be checked by a randomised algorithm, which accepts valid proofs and rejects invalid ones with very high probability, while requiring minimal interaction with the proof.*

Pinocchio and Groth16 are among the most efficient and widely used PCP systems for proving **Quadratic Arithmetic Programs (QAPs)**. As we have seen before, a QAP is defined by a polynomial identity constrained to a domain $\mathcal{H}$, of the form:

$$\left(\sum_{0}^{\ell} a_i(X)s_i + \sum_{\ell+1}^{m} a_i(X)s_i\right) \cdot \left(\sum_{0}^{\ell} b_i(X)s_i + \sum_{\ell+1}^{m} b_i(X)s_i\right) - \left(\sum_{0}^{\ell} c_i(X)s_i + \sum_{\ell+1}^{m} c_i(X)s_i\right) = 0 \quad \text{for } X \in \mathcal{H}.$$

The prover is tasked with demonstrating knowledge of the public signals $(s_0, s_1, \ldots, s_\ell)$ and private signals $(s_{\ell+1}, \ldots, s_m)$ that satisfy this QAP.

2.5.1 QAP Divisibility

The Quadratic Arithmetic Program (QAP) divisibility property is a key mechanism for verifying that a polynomial $p(X)$ satisfies the constraints encoded in a circuit. Instead of directly verifying that $p(X) = 0$ for all $X \in \mathcal{H}$ (where $\mathcal{H}$ is a cyclic group of roots of unity), the goal is to demonstrate that $p(X)$ is divisible by the vanishing polynomial $z_{\mathcal{H}}(X)$, defined as:

$$z_{\mathcal{H}}(X) = X^{n+1} - 1.$$

The prover must demonstrate the following identity:

$$\frac{p(X)}{z_{\mathcal{H}}(X)} = h(X) \quad \text{where} \quad X \in \mathbb{F},$$

or equivalently:

$$p(X) = h(X) \cdot z_{\mathcal{H}}(X),$$

where $h(X)$ is a polynomial of degree $d < n$.

In zk-SNARKs, $p(X)$ is constructed using circuit polynomials $a_i(X)$, $b_i(X)$, and $c_i(X)$, which define the topology of the circuit, as well as public $(s_0, \ldots, s_\ell)$ and

private $(s_{\ell+1}, \ldots, s_n)$ signals, as we have seen in the previous section:

$$p(X) = \left(\sum_{i=0}^{\ell} a_i(X)s_i + \sum_{i=\ell+1}^{m} a_i(X)s_i\right) \cdot \left(\sum_{i=0}^{\ell} b_i(X)s_i + \sum_{i=\ell+1}^{m} b_i(X)s_i\right) - \left(\sum_{i=0}^{\ell} c_i(X)s_i + \sum_{i=\ell+1}^{m} c_i(X)s_i\right).$$

To verify this, the computation is evaluated at an encrypted random point $X = \tau$, ensuring the integrity of the witness polynomials while maintaining privacy for the private inputs. That is, we evaluate on the powers of tau as explained in 2.3.

Example 2.6. *Following the example in Example 2.1, we had*

$$p(X) = 38X^6 + 31X^5 + 3X^4 + 3X^2 + 10X + 38 = 0$$

Then, we know by construction that the polynomial $p(X)$ has the following form:

$$p(X) = (X - \omega^0) \cdots (X - \omega^3) \cdot h(X)$$

Where $z_{\mathcal{H}}(X) = (X - \omega^0)...(X - \omega^3) = X^4 - 1$. The prover can compute h(X), of degree $d < n$, in this example $2 < 3$:

$$h(X) = p(X)/z_{\mathcal{H}}(X) = 38X^2 + 31X + 3$$

If p(X) is not zero at $\mathcal{H}$, then, dividing $p(X)$ by $z_{\mathcal{H}}(X)$, we will not obtain a polynomial $h(X)$ but a fraction of polynomials, for which we cannot create a commitment.

As we have seen, now the prover must verify the following equation:

$$p(X) = h(X) \cdot (X^{n+1} - 1).$$

At a random point $X = \tau$, the QAP equation becomes:

$$\left(\sum_{i=0}^{\ell} a_i(\tau)s_i + \sum_{i=\ell+1}^{m} a_i(\tau)s_i\right) \cdot \left(\sum_{i=0}^{\ell} b_i(\tau)s_i + \sum_{i=\ell+1}^{m} b_i(\tau)s_i\right) - \left(\sum_{i=0}^{\ell} c_i(\tau)s_i + \sum_{i=\ell+1}^{m} c_i(\tau)s_i\right) = h(\tau) \cdot (\tau^{n+1} - 1).$$

However, direct computation in polynomial space is not possible due to elliptic curve multiplications. Instead, the verification is performed using asymmetric pairings.

The verification process leverages pairings to evaluate the relationships between commitments:

$$
\begin{aligned}
e\left(G_1, G_2\right)^{\left(\sum_0^\ell a_i(\tau)s_i+\sum_{\ell+1}^m a_i(\tau)s_i\right)\cdot\left(\sum_0^\ell b_i(\tau)s_i+\sum_{\ell+1}^m b_i(\tau)s_i\right)-\left(\sum_0^\ell c_i(\tau)s_i+\sum_{\ell+1}^m c_i(\tau)s_i\right)} \\
= e\left(G_1, G_2\right)^{h(\tau)\,Z_{\mathcal{H}}(\tau)} \iff \\
e\left(G_1, G_2\right)^{\left(\sum_0^\ell a_i(\tau)s_i+\sum_{\ell+1}^m a_i(\tau)s_i\right)\cdot\left(\sum_0^\ell b_i(\tau)s_i+\sum_{\ell+1}^m b_i(\tau)s_i\right)} \\
= e\left(G_1, G_2\right)^{h(\tau)\cdot Z_{\mathcal{H}}(\tau)} \cdot e\left(G_1, G_2\right)^{\left(\sum_0^\ell c_i(\tau)s_i+\sum_{\ell+1}^m c_i(\tau)s_i\right)} \iff \\
e\left(\sum_0^\ell A_i s_i+\sum_{\ell+1}^m A_i s_i, \sum_0^\ell B_i s_i+\sum_{\ell+1}^m B_i s_i\right) = \\
e\left(([h(\tau)]_1, [z_{\mathcal{H}(\tau)}]_2\right) \cdot e\left(\sum_0^\ell C_i s_i + \sum_{\ell+1}^m C_i s_i, [1]_2\right).
\end{aligned}
$$

Note that we do as much group operations as possible in $\mathbb{G}_1$, as operations in this group are approximately 3.5 times faster than in $\mathbb{G}_2$. Additionally, the SRS includes powers of τ in $\mathbb{G}_2$ for commitments to $b_i(X)$ and $z_{\mathcal{H}}(X)$, so we need them.

The pairing check can be rewritten as:

$$
\begin{gathered}
e\left(vk_x^A + \pi_A, vk_x^B + \pi_B\right) = e\left(\pi_h, vk_z\right) \cdot e\left(vk_x^C + \pi_C, [1]_2\right) \\
vk_x^A = \sum_0^\ell A_i s_i \quad \pi_A = \sum_{\ell+1}^m A_i s_i \quad vk_x^B = \sum_0^\ell B_i s_i \quad \pi_B = \sum_{\ell+1}^m B_i s_i \\
vk_x^C = \sum_0^\ell A_i s_i \quad \pi_C = \sum_{\ell+1}^m C_i s_i \quad \pi_h = [h(\tau)]_1 \quad vk_z = [z_{\mathcal{H}}(\tau)]_2
\end{gathered}
$$

Where π_A, π_B, π_C, and π_h, are points provided by the prover. The point vk_z is provided by the trusted setup (for verifiers). Finally, vk_x^A, vk_x^B and vk_x^C are computed by the verifier for the given public signals.

Limitations of QAP Divisibility While the QAP divisibility check is a foundational element in ensuring the correctness of zk-SNARKs, it alone does not guarantee complete soundness. Specifically, it is possible to construct valid-looking equations that satisfy the QAP divisibility property but do not adhere to the intended circuit constraints. This occurs because we are not checking that π_A, π_B, and π_C are linear combinations of the committed polynomials $a_i(X)$, $b_i(X)$, and $c_i(X)$, respectively.

Example 2.7. *Consider the following example:*

$$
(X^{n+1} - 1) \cdot X^2 = X^n \cdot X^3 - X^2,
$$

or equivalently:

$$z_{\mathcal{H}}(X) \cdot h(X) = a(X) \cdot b(X) - c(X).$$

This equation can pass the QAP divisibility check:

$$e(vk_x^A + \pi_A, vk_x^B + \pi_B) = e(\pi_h, vk_z) \cdot e(vk_x^C + \pi_C, [1]_2),$$

with the following commitments:

$$\begin{aligned} vk_x^A + \pi_A &= [\tau^n]_1, \\ vk_x^B + \pi_B &= [\tau^3]_2, \\ vk_x^C + \pi_C &= [\tau^2]_1, \\ \pi_h &= [\tau^2]_1. \end{aligned}$$

As we can observe, while QAP divisibility ensures $p(X)$ satisfies the QAP constraints, it is not sufficient for soundness. Two additional checks are required:

1. **Knowledge of commitments:** The prover must demonstrate that π_A, π_B, and π_C are linear combinations of the committed polynomials $a_i(X)$, $b_i(X)$, and $c_i(X)$, respectively, and that the prover knows the coefficients.
2. **Same coefficients:** The prover must show that the same coefficients s_i were used to compute π_A, π_B, and π_C.

These checks are implemented using toxic random values introduced during the trusted setup phase.

2.5.2 Knowledge of Commitments Check

Since we check the same coefficients we can simplify the divisibility check by only exposing publics in the A_i part:

$$e(vk_x + \pi_A, \pi_B) = e(\pi_h, vk_z)\ e(\pi_C, [1]_2)$$

With $vk_x = \sum_0^\ell A_i s_i \quad \pi_A = \sum_{\ell+1}^m A_i s_i \quad \pi_B = \sum_0^m B_i s_i \quad \pi_C = \sum_0^m C_i s_i$,

The trusted setup provides $vk_A = [\alpha_A]_2$, which is a point encoding the toxic randomness α_A and A_i and $A_i' = A_i \cdot \alpha_A$ for $i \in \{\ell+1, \ldots, m\}$, points derived from the circuit-specific polynomials.

Using these, we verify the provers commitment π_A through pairings:

$$e([1]_1, [1]_2)^{\alpha_A \sum_{i=\ell+1}^n A_i s_i} = e([1]_1, [1]_2)^{\sum_{i=\ell+1}^m A_i' s_i}.$$

This can be expressed as:

$$e(\pi_A, [\alpha_A]_2) = e\left(\sum_{i=\ell+1}^{m} A_i' s_i, [1]_2\right),$$

or equivalently:

$$e(\pi_A, vk_A) = e(\pi'_A, [1]_2),$$

where $\pi'_A = \sum_{i=\ell+1}^{n} A'_i s_i$.

Similarly, we extend the pairing check to the commitments π_B and π_C:

$$e(vk_B, \pi_B) = e(\pi'_B, [1]_2),$$

$$e(\pi_C, vk_C) = e(\pi'_C, [1]_2).$$

Here, the trusted setup provides:

$$vk_B = [\alpha_B]_1, \qquad \pi'_B = \sum_{i=0}^{m} B'_i s_i,$$

$$vk_C = [\alpha_C]_2, \qquad \pi'_C = \sum_{i=0}^{m} C'_i s_i.$$

Note that, for efficiency, π'_B is defined in $\mathbb{G}_1$ rather than in $\mathbb{G}_2$. This choice minimises computational overhead, as operations in $\mathbb{G}_1$ are faster than in $\mathbb{G}_2$.

By implementing these checks, we ensure that the commitments π_A, π_B, and π_C are valid linear combinations of the circuit-specific polynomials $a_i(X)$, $b_i(X)$, and $c_i(X)$.

2.5.3 Same Coefficients Check

The "same coefficients" check ensures that the same set of coefficients $s_0, s_1, \ldots, s_n$ are used consistently across the commitments A_i, B_i, and C_i, i.e., in $vk_x + \pi_A, \pi_B$, and π_C. This section outlines the rationale and implementation of the check.

Let us consider a triplet of points A_0, B_0, C_0. The prover might construct a linear combination such as:

$$s_0^A A_0 + s_0^B B_0 + s_0^C C_0.$$

If $A_0 \neq B_0 \neq C_0$, the only way to achieve a common factor is to use the same coefficient, $s_0^A = s_0^B = s_0^C = s_0$, yielding:

$$s_0(A_0 + B_0 + C_0).$$

We extend this idea to construct a pairing-based check. The prover is required to provide four points:

$$P_1 = s_0 A_0, \quad P_2 = s_0 B_0, \quad P_3 = s_0 C_0, \quad P_4 = s_0(A_0 + B_0 + C_0).$$

The verifier checks:

$$e(P_1 + P_2 + P_3, [1]_2) = e(P_4, [1]_2).$$

However, this approach is vulnerable since the prover can fabricate valid points without using consistent coefficients or the original points A_0, B_0, C_0.

To enforce the usage of A_0, B_0, C_0, and $A_0 + B_0 + C_0$ with consistent coefficients, a random shift is introduced. The trusted setup provides a toxic randomness β and defines:

$$K_0 = \beta(A_0 + B_0 + C_0).$$

The verifier checks:

$$e(s_0(A_0 + B_0 + C_0), [\beta]_2) = e(s_0 K_0, [1]_2),$$

where $K_0 = \beta(A_0 + B_0 + C_0)$ is shared with the prover, $[\beta]_2$ is shared with the verifier, and β itself is destroyed.

To address cases where $A_i = B_i = C_i$ for some i, we randomize the points using toxic random values ρ_A and ρ_B. The randomized linear combination becomes:

$$(s_0^A \rho_A A_0 + s_0^B \rho_B B_0 + s_0^C \rho_A \rho_B C_0)\beta = s_0(\rho_A \beta A_0 + \rho_B \beta B_0 + \rho_A \rho_B \beta C_0).$$

This extends naturally to all A_i, B_i, and C_i, resulting in:

$$\sum_{i=0}^{m}(s_i \rho_A a_i(\tau) + s_i \rho_B b_i(\tau) + s_i \rho_A \rho_B c_i(\tau))\beta = \sum_{i=0}^{m} s_i(\rho_A \beta a_i(\tau) + \rho_B \beta b_i(\tau) + \rho_A \rho_B \beta c_i(\tau)).$$

Implementing the Check The above equation is converted into a pairing-based check:

$$e\left(\sum_{i=0}^{m} A_i s_i + \sum_{i=0}^{m} C_i s_i, [\beta]_2\right) \cdot e([\beta]_1, \sum_{i=0}^{m} B_i s_i) = e\left(\sum_{i=0}^{m} K_i s_i, [1]_2\right),$$

where:

$$A_i = [\rho_A a_i(\tau)]_1, \quad B_i = [\rho_B b_i(\tau)]_2, \quad C_i = [\rho_A \rho_B c_i(\tau)]_1,$$
$$K_i = [(\rho_A a_i(\tau) + \rho_B b_i(\tau) + \rho_A \rho_B c_i(\tau))\beta]_1.$$

Avoiding Issues with Encrypted Shifts To prevent a malicious prover from exploiting the provided encrypted shifts $[\beta]_1$ and $[\beta]_2$, an additional random value γ is introduced. The check is modified to:

$$e(vk_x + \pi_A + \pi_C, vk^2_{\beta\gamma}) \cdot e(vk^1_{\beta\gamma}, \pi_B) = e(\pi_K, vk_\gamma),$$

where $vk^1_{\beta\gamma} = [\beta\gamma]_1, vk^2_{\beta\gamma} = [\beta\gamma]_2$, and $vk_\gamma = [\gamma]_2$.

2.6 Summary of the Pinocchio SNARK

1. The **TS** provides a proving key $\vec{pk}$ to provers, where

   $\vec{pk} = \{\{H_i\}_0^n, \{A_i\}_0^m, \{B_i\}_0^m, \{C_i\}_0^m, \{A'_i\}_{l+1}^m, \{B'_i\}_0^m, \{C'_i\}_0^m, \{K_i\}_0^m\}$

   $H_i = [\tau^i]_1$

   $A_i = [\rho_A\, a_i(\tau)]_1 \quad B_i = [\rho_B\, b_i(\tau)]_2 \quad C_i = [\rho_A \rho_B\, c_i(\tau)]_1$

   $A'_i = [\alpha_A\, \rho_A\, a_i(\tau)]_1 \quad B'_i = [\alpha_B\, \rho_B\, b_i(\tau)]_2 \quad C'_i = [\alpha_C\, \rho_A\, \rho_B\, c_i(\tau)]_1$

   $K_i = [(\rho_A\, a_i(\tau) + \rho_B\, b_i(\tau) + \rho_A \rho_B\, c_i(\tau))\, \beta)]_1$

   The **TS** also provides a verifying key for verifiers

   $\vec{vk} = \{vk_Z = [z_H(\tau)\, \rho_A\, \rho_B]_2, vk_A = [\alpha_A]_2, vk_B = [\alpha_B]_1, vk_C = [\alpha_C]_2, vk^2_{\beta\gamma}, vk^1_{\beta\gamma}, vk_\gamma = [\gamma]_2\}$.

2. **Prover** computes Π with $\vec{pk}$: 8 points π_h, π_A, π_C, π'_A, π'_B, π'_C, π_K in $\mathbb{G}_1$ and π_B in $\mathbb{G}_2$.

   $\pi_h = \sum_{i=0}^{n-1} c_i^h\, H_i$

   $\pi_A = \sum_{i=l+1}^{m} A_i\, s_i \quad \pi_C = \sum_{i=0}^{m} C_i\, s_i \quad \pi'_A = \sum_{i=l+1}^{m} A'_i\, s_i \quad \pi'_C = \sum_{i=0}^{m} C'_i\, s_i \quad \pi'_B = \sum_{i=0}^{m} B'_i\, s_i$

   $\pi_K = \sum_{i=0}^{m} K_i\, s_i \quad \pi_B = \sum_{i=0}^{m} B_i\, s_i$

3. The **verifier** computes the verification part of the public inputs:

   $vk_x = \sum_{i=0}^{l} A_i\, s_i$

   Then, the **verifier** checks 5 equations with 12 pairings to accept the proof:

$$
\begin{aligned}
e(vk_x + \pi_A, \pi_B) &= e(\pi_h, vk_z)\; e(\pi_C, [1]_2) \\
e(\pi_A, vk_A) &= e(\pi'_A, [1]_2) \\
e(vk_B, \pi_B) &= e(\pi'_B, [1]_2) \\
e(\pi_C, vk_C) &= e(\pi'_C, [1]_2) \\
e(vk_x + \pi_A + \pi_C, vk^2_{\beta\gamma}) \cdot e(vk^1_{\beta\gamma}, \pi_B) &= e(\pi_K, vk_\gamma)
\end{aligned}
$$

2.7 From a SNARK to a zk-SNARK

The SNARK described thus far **does not provide zero-knowledge**. The authors of Pinocchio were primarily focused on verifiable computation rather than zero-knowledge (ZK); however, transforming the Pinocchio SNARK into a zk-SNARK is relatively straightforward.

One of the reasons for this is that in the proof, the prover is required to provide π_A, π_B, and π_C, which are linear combinations of A_i, B_i, and C_i using the witness

set of signals $\mathsf{s} = \{s_0, s_1, \ldots, s_m\}$. This process inadvertently leaks certain information to the verifier, as the verifier cannot compute these linear combinations independently without knowledge of s.

To ensure zero-knowledge, the prover introduces three random values, δ_A, δ_B, and δ_C, to create perturbed versions $\bar{\pi}_A$, $\bar{\pi}_B$, and $\bar{\pi}_C$. These modified terms are **no longer linear combinations of the witness** evaluated at known points.

Perturbing the Linear Combinations The perturbation is achieved by adding a randomised version of $z_{\mathcal{H}}(X)$ to π_A, π_B, and π_C. This adjustment does not affect the identity being checked at roots of unity since $z_{\mathcal{H}}(X)$ evaluates to zero at $\mathcal{H}$. The perturbed values are:

$$\bar{\pi}_A = \sum_{i=l+1}^{m} A_i\, s_i + [z_{\mathcal{H}}(\tau)\, \rho_A]_1\, \delta_A + 0\, s_{m+2} + 0\, s_{m+3}$$

$$\bar{\pi}_B = \sum_{i=0}^{m} B_i\, s_i + 0\, s_{m+1} + [z_{\mathcal{H}}(\tau)\, \rho_B]_2\, \delta_B + 0\, s_{m+3}$$

$$\bar{\pi}_C = \sum_{i=0}^{m} C_i\, s_i + 0\, s_{m+1} + 0\, s_{m+2} + [z_{\mathcal{H}}(\tau)\, \rho_A\, \rho_B]_1\, \delta_C$$

Shifted Versions with α The same approach is applied to the shifted versions $\bar{\pi}'_A$, $\bar{\pi}'_B$, and $\bar{\pi}'_C$, incorporating randomisation with α terms:

$$\bar{\pi}'_A = \sum_{i=l+1}^{m} A'_i\, s_i + [z_{\mathcal{H}}(\tau)\, \rho_A\, \alpha_A]_1\, \delta_A + 0\, s_{m+2} + 0\, s_{m+3}$$

$$\bar{\pi}'_B = \sum_{i=0}^{m} B'_i\, s_i + 0\, s_{m+1} + [z_{\mathcal{H}}(\tau)\, \rho_B\, \alpha_B]_2\, \delta_B + 0\, s_{m+3}$$

$$\bar{\pi}'_C = \sum_{i=0}^{m} C'_i\, s_i + 0\, s_{m+1} + 0\, s_{m+2} + [z_{\mathcal{H}}(\tau)\, \rho_A\, \rho_B\, \alpha_C]_1\, \delta_C$$

Updating the Trusted Setup Parameters The parameters from the trusted setup must be expanded to include the new values, resulting in an updated public key:

$$\vec{pk} = \{\{H_i\}_0^n, \{A_i\}_0^{m+3}, \{B_i\}_0^{m+3}, \{C_i\}_0^{m+3}, \{A'_i\}_{l+1}^{m+3}, \{B'_i\}_0^{m+3}, \{C'_i\}_0^{m+3}, \{K_i\}_0^{m+3}\}$$

Where:

$$A_{m+1} = [Z_{\mathcal{H}}(\tau)\, \rho_A]_1, \quad B_{m+2} = [Z_{\mathcal{H}}(\tau)\, \rho_B]_2, \quad C_{m+3} = [Z_{\mathcal{H}}(\tau)\, \rho_A\, \rho_B]_1$$

$$A_{m+2} = A_{m+3} = C_{m+1} = C_{m+2} = B_{m+1} = B_{m+3} = \mathcal{O}$$

$$A'_{m+1} = [Z_{\mathcal{H}}(\tau)\,\rho_A\,\alpha_A]_1, \quad B'_{m+2} = [Z_{\mathcal{H}}(\tau)\,\rho_B\,\alpha_B]_2, \quad C'_{m+3} = [Z_{\mathcal{H}}(\tau)\,\rho_A\,\rho_B\,\alpha_C]_1$$
$$A'_{m+2} = A'_{m+3} = C'_{m+1} = C'_{m+2} = B'_{m+1} = B'_{m+3} = \mathcal{O}$$

This process ensures that the Pinocchio SNARK becomes a zk-SNARK while maintaining the original functionality and efficiency.

Chapter 3

Circuits

A **digital signature scheme** is a cryptographic primitive used to ensure the authenticity and the integrity of digital messages or documents. It serves as the digital equivalent of a handwritten signature or a stamped seal, but is much more secure due to its reliance on mathematical tools. Digital signatures are a fundamental component in modern cybersecurity, particularly in secure communications, financial transactions, and software distribution.

The digital signature process can be divided into three stages. It begins with **key generation**, where the signer, typically referred to as Alice, creates a pair of cryptographic keys. The first is a private key, which is kept secret and only known to Alice, and the second is a public key, which is publicly known and can be used by anyone to verify her signature. The next stage is **signing the message**. When Alice wishes to sign a message, she uses her private key along with a cryptographic algorithm to generate a digital signature. This signature is uniquely tied to the specific message being signed and cannot be reused or applied to any other message. The cryptographic algorithm ensures that even the slightest alteration to the message will invalidate the signature, thereby safeguarding the message's integrity. Finally, in the **verification stage**, the recipient of the message, often called Bob, uses Alice's public key to confirm the authenticity of the signature. Bob verifies whether the signature matches the original message and checks for any signs of tampering. If the verification process succeeds, Bob can trust that the message originated from Alice and has not been altered in transit. This robust mechanism guarantees the authenticity, integrity, and non-repudiation of the communication, making digital signatures a cornerstone of secure interactions in the digital world.

Digital signature algorithms employ many diverse schemes fit for different use cases and security requirements. One of the earliest and well-known is the **RSA (Rivest-Shamir-Adleman)** algorithm, a method for creating and verifying signatures based on the modular arithmetic and number theory of large prime numbers. While being quite older, it finds its wide application in secure email communication and software signing. Another one is the very popular **DSA (Digital Signature Algorithm)** which is a product of the U.S. Digital Signature Standard

(DSS). Building on the mathematical basis of discrete logarithms, DSA promises similar security assurances as RSA but with differing performance. Variants like the **Elliptic Curve Digital Signature Algorithm (ECDSA)** have been coming into play in recent times. ECDSA enhances DSA through elliptic curve cryptography, enabling the use of smaller key sizes and leading to faster computations, thus, making ECDSA recently the preferred candidate in blockchain networks like Bitcoin and Ethereum. **EdDSA (Edwards-Curve Digital Signature Algorithm)** is a modern alternative to ECDSA. Built on twisted Edwards curves, EdDSA offers improved performance and resistance to certain types of side-channel attacks. It is increasingly adopted in privacy-focused applications, including secure messaging protocols and in distributed systems requiring high efficiency. The **ElGamal** signature scheme is another fundamental algorithm based on discrete logarithms. Although not commonly used anymore, serves as a stepping stone for other cryptographic signature developments. The influence can be seen in later designs such as **BLS (Boneh-Lynn-Shacham)** signatures, which offer large efficiency through signature aggregation, considerably popular in mega-scale systems like Ethereum 2.0 and decentralized storage networks.

In this chapter, we dive into two widely adopted digital signature schemes: ECDSA and BLS signatures, which are cornerstones of modern cryptography with significant applications in blockchain and secure communication systems. These cryptographic protocols not only provide authentication and integrity but also enable advanced features like aggregation and threshold signatures, making them indispensable in distributed systems.

3.1 CIRCOM

Circom is a novel domain-specific language designed for defining arithmetic circuits that can be used to generate zero-knowledge proofs. The Circom compiler, developed in Rust, generates an R1CS file with a set of associated constraints and a program (written in either C++ or WebAssembly) to efficiently compute a valid assignment for all circuit wires, as we can observe in Figure 3.1. One of Circom's key features is its modularity, which allows programmers to define parameterisable circuits known as **templates**. These templates can be instantiated to create larger circuits, simplifying the processes of testing, reviewing, auditing, and formally verifying large and complex Circom circuits [ic23]. Additionally, Circom users can create custom templates or use templates from `Circomlib`, a publicly available library that includes hundreds of circuits such as comparators, hash functions, digital signatures, binary and decimal converters, and more. To bridge theory and practice, this work focuses on testing Circom libraries designed for signature verification circuits. Rather than developing new circuits, we will utilise existing implementations and perform rigorous testing to validate their correctness and performance. Using JavaScript, we will interact with these Circom circuits to generate comprehensive test cases that simulate realistic scenarios, en-

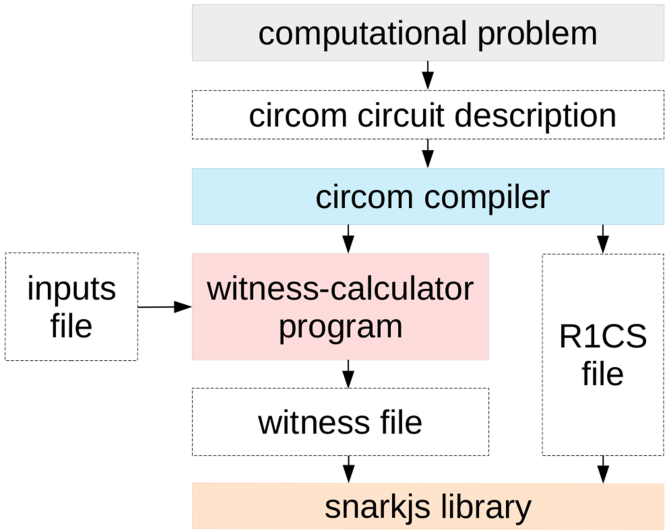

Figure 3.1: The process of generating and validating a ZK proof involves the following steps: First, create a JSON file with inputs. This file is passed to a C++ or WebAssembly program generated by the circom compiler, which produces a binary witness file. After compiling the circuit and running the witness calculator with the inputs, two files are generated: a `.wtns` file containing computed signals and an `.r1cs` file with circuit constraints. Using these files, ZK proofs can be computed and verified with snarkjs.

Source: [BMIMT+23]

suring circuit reliability under diverse conditions. This approach aims to establish a robust workflow for assessing signature verification circuits and their integration into broader cryptographic systems.

3.2 ECDSA Signature

The **ECDSA (Elliptic Curve Digital Signature Algorithm)** is a cryptographic scheme that enhances the traditional DSA by utilizing elliptic curve cryptography (ECC). ECC operates over elliptic curves defined over finite fields, providing the same level of security as traditional algorithms like RSA and DSA but with significantly smaller key sizes and faster computations. These features make ECDSA particularly attractive for applications in constrained environments, such as IoT devices, mobile platforms, and blockchain networks. The algorithm is defined generically to work with any elliptic curve that satisfies cryptographic security requirements. Such a curve, denoted as $E(\mathbb{F}_p)$ is defined over a finite field $\mathbb{F}_p$, which is commonly known as **base field**. In addition to the field and equation of the curve, we need G, a base point of (large) prime order q on the curve. This point will serve as a pivot between the scalar space and the EC space, by means of

the transformation provided by the scalar multiplication

$$\begin{aligned} \mathbb{F}_q &\to E(\mathbb{F}_p) \\ z &\mapsto z \cdot G. \end{aligned}$$

where q denotes the order of G. The field $\mathbb{F}_q$ is also known as the **scalar field** of the curve.

Key Generation Phase Alice wants to sign a document m, represented as an integer. Typically, instead of signing m directly, she signs its hash $H(m)$ for added security. To achieve this, Alice selects an elliptic curve $E(\mathbb{F}_p)$ defined over a finite field $\mathbb{F}_p$ and chooses a base point G on the curve with order q.

Next, Alice randomly generates a secret integer a, which serves as her private key. She computes her public key A by performing scalar multiplication in the elliptic curve space:

$$A = a \cdot G.$$

Finally, Alice publishes the tuple $(\mathbb{F}_p, E, q, G, A)$, which includes the parameters of the elliptic curve and her public key.

Message Signing Phase Alice also selects a random $k \in_R \{1, \ldots, q-1\}$ and computes $K = k \cdot G = (K_x, K_y)$ and $r = K_x$ (if $r \geqslant q$, we repeat this step). In addition, Alice has to compute a integer $s \in \{1, \ldots, p-1\}$ that satisfies the following equation,

$$H(m) \equiv k \cdot s - r \cdot a \mod q$$

where m is the message to be signed. Then, the signature of the message is the tuple (r, s).

Verification Phase Bob, who wants to verify that the signature is correct, has the following information: the signature (r, s), the public key $A = a \cdot G$, the hash of the message $H(m)$, and the parameters of the elliptic curve, which include the generator of the group G, the order of the finite field p, and the order of the group generated by G, denoted by q. The verification procedure in order to check if the signature is correct can be derived as follows:

First, multiply both sides by the generator G:

$$H(m) \cdot G = k \cdot s \cdot G - r \cdot a \cdot G$$

Then, isolate the random point $K = k \cdot G$:

$$K = (H(m) \cdot G + r \cdot A) \cdot s^{-1}$$

Finally, Bob has to check the following equation:

$$(H(m) \cdot G + r \cdot A) \cdot s^{-1}|_x \equiv r \mod q$$

If the message is signed correctly, the verification equation holds. Observe that Bob has all the information needed to check the last equation.

The `secp256k1` elliptic curve One of the most widely recognized elliptic curves used with ECDSA is `secp256k1`. This curve is particularly famous for its role in cryptocurrency systems, such as Bitcoin and Ethereum. It is part of the Standards for Efficient Cryptography (SEC) group of recommended curves. The curve operates over the prime field $\mathbb{F}_p$, where $p = 2^{256} - 2^{32} - 977$, a large prime number which ensures that the field size provides robust security. The equation of the curve is $y^2 = x^3 + 7 \mod p$, distinguished by its simplicity.

The curve `secp256k1` is specifically chosen in blockchain due to its efficient implementation and relatively simple structure. It allows for fast key generation, signature creation, and verification, which are critical for high-performance systems with a high volume of transactions. The curves parameters ensure secure operations against known cryptographic attacks, such as those exploiting weak curve structures.

3.2.1 CIRCOM Circuits

In this section, we want to test the CIRCOM template for the ECDSA signature verification present in the 0xPARC repository [xc23]. In order to explain the signature test itself, we must first consider several circuits that are necessary to perform the test.

First of all, we must take into account that the ECDSA signature involves two primary types of operations. The first type consists of group operations on the points of the `secp256k1` elliptic curve, which has order q, and the second type involves operations in $\mathbb{F}_p$, the finite field where the coordinates of the points on the curve reside.

However, it is crucial to note that the proving system typically imposes a specific field, often the BN128 prime field due to the Groth16 backend. This requirement mandates that all signal values lie within this field. Since circuit values must conform to the BN128 field, and there is frequently a mismatch between this field and the two operations previously defined, inputs are divided into multiple fixed-length chunks to ensure compatibility.

In analogy with programming languages, a number represented in this manner is referred to as a `BigInt`. To manage these numbers effectively, most templates are parameterised by two values: `n`, representing the number of bits in each chunk, and `k`, the total number of chunks.

The templates used for ECDSA signature verification can be categorised into three main groups, each serving a distinct purpose:

- Templates designed for handling `BigInt` operations.
- Templates for performing operations on the `secp256k1` elliptic curve.
- Templates specifically related to the ECDSA Signature. These include one for deriving the public key from a private key and another for verifying the signature itself.

Now, I will focus on developing the two last templates that relate to the signature itself. On one hand, we have the template in charge of derivate the public key `pubkey` from a private key `privkey`, named `ECDSAPrivToPub` template. The computation to be carried out is

$$\texttt{pubkey} = \texttt{privkey} \cdot G,$$

being $\cdot$ the scalar multiplication over the `secp256k1` curve. The scalar multiplication is performed using a specific algorithm known as the **Windowed Algorithm for Elliptic Curve Scalar Multiplication.**

This algorithm works by expressing the private key in base 2^8, dividing it into 32 blocks, each containing 8 bits. Instead of calculating multiples of G dynamically during the execution, the algorithm precomputes these values. This allows to efficiently retrieve and sum the corresponding precomputed multiples for each block, significantly reducing the computational cost by converting complex point multiplications into simpler point additions. Then, multiplexers are employed in the circuit to select the appropriate multiples of G for each block of bits. These values are then progressively accumulated, block by block, to compute the final public key.

The following operations are carried out during the algorithm:

At first, decompose the `privkey` in base 2^8, that is,

$$\texttt{privkey} = \sum_{i=0}^{31} j_i \cdot 2^{8i}$$

with $j_i \in \{0, 1, \ldots, 2^8 - 1\} = \{0, 1, \ldots, 255\}$.

The computation that we want to perform is the following

$$\left(\sum_{i=0}^{31} j_i \cdot 2^{8i} \right) \cdot G = \sum_{i=0}^{31} j_i \cdot 2^{8i} \cdot G.$$

The idea is to precompute all the scalar multiplications

$$j_i \cdot 2^{8i} \cdot G,$$

for $i \in \{0, 1, \ldots, 31\}$ and $j_i \in \{0, 1, \ldots, 255\}$.

The final step is to sum all the corresponding precomputed values to produce the result, requiring at most 32 additions to perform the required scalar multiplication:

$$\sum_{i=0}^{31} j_i \cdot 2^{8i} \cdot G.$$

Observe that we need to precompute a table of $32 \times 256 = 8192$ values.

Let's see it with a simple example. Imagine we want to calculate $n \cdot G$ where $n = 29$. For sake of simplicity, we will work with blocks of 2 bits, instead of 8, so we will have to compute the base $2^2 = 4$ representation of the private key:

$$29 = (131)_4 = 1 \cdot 2^4 + 3 \cdot 2^2 + 1.$$

Then, we want to compute

$$29 \cdot G = \left(1 \cdot 2^4 + 3 \cdot 2^2 + 1\right) G = \underbrace{1 \cdot 2^4 \cdot G}_{\text{precomputed}} + \underbrace{3 \cdot 2^2 \cdot G}_{\text{precomputed}} + \underbrace{1 \cdot G}_{\text{precomputed}}.$$

On the other hand, we have the template responsible for verifying the ECDSA signature, named `ECDSAVerifyNoPubkeyCheck`, takes as inputs the signature components `r` and `s`, the public key `pubkey` and the hash of the message `msghash`. By leveraging the operations defined in the templates of the other two groups, the template produces an output, `result`, which is 1 if the signature inputs are valid and 0 otherwise. Its important to note that this template does not check that `pubkey` is valid from a given private key.

This can be seen in the diagram corresponding to 3.2. The following templates are employed: `BigMultModP`, which performs the modular multiplication of two large integers and reduces the result modulo p; `Secp256k1AddUnequal`, which adds two points on the elliptic curve under the assumption that the points are distinct; `Secp256k1ScalarMult`, which computes the scalar multiplication of a curve point by a given scalar; `ECDSAPrivToPub`, a template discussed in detail later; and `IsEqual`, which verifies if the input value matches a predefined parameter.

3.2.2 ECDSA Signature Test

After having introduced some of the templates, we can now move on to what is the ECDSA Signature Test.

To generate the appropriate inputs, we need to utilise the `@noble/secp256k1` package, which provides a range of operations for the `secp256k1` elliptic curve. Among these, the `sign` function is used to generate an ECDSA signature for a given message and private key. From the `Point` class, which represents a point on the `secp256k1` curve, we use the `fromPrivateKey()` function to derive the `pubKey` from a private key. The `CURVE` module contains various curve parameters, including the base field prime (`CURVE.P`). Additionally, the `utils` module provides utilities for generating a random `privKey` and hashing messages with `sha256`.

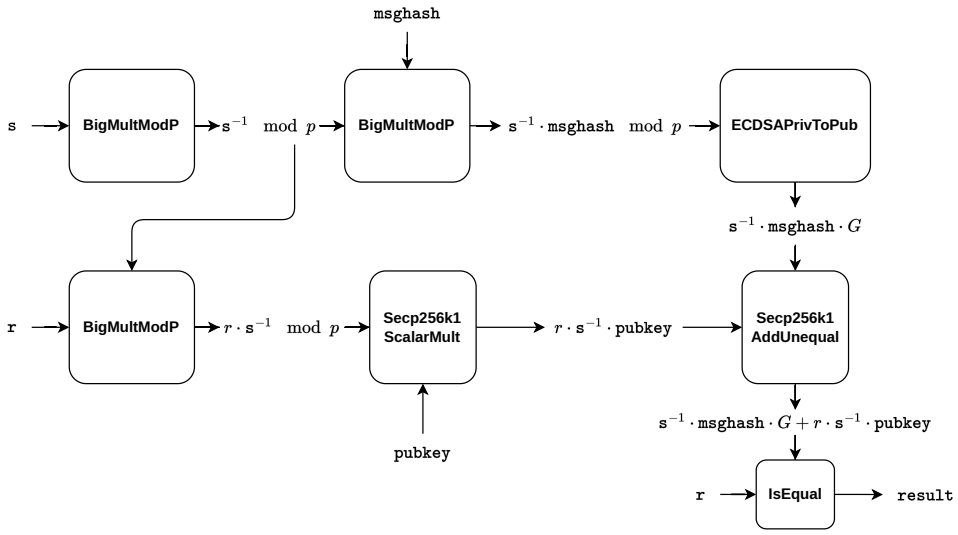

Figure 3.2: This diagram shows the `EDSAVerifyNoPubkeyCheck` template flow. Each box represents a template designed to perform a specific operation, taking an input parameter and producing the result of that operation as output. This interconnected flow of templates works collectively to enable the verification of the signature.

Since various parts of the code require inputs and outputs in specific formats, three functions are used to facilitate format conversion. The `bigint_to_array` function converts a `BigInt` into `k` chunks of `n` bits. Additionally, the `bigint_to_Uint8Array` function converts a `BigInt` into a `Uint8Array` (an array of 8-bit unsigned integers) in big-endian format, where the most significant byte appears first. Conversely, the `Uint8Array_to_bigint` function transforms a `Uint8Array` in big-endian format back into a `BigInt`, treating each byte as part of the big-endian representation of the number.

The ECDSA Signature Test is divided in two tests: the **PrivToPub Test** for testing that the obtained public key is the correct one and the **Verify Test** for verifying the signature, both are written in JavaScript.

ECDSAPrivToPub Test

Before running tests, the circuit `ecdsa_priv2pub.test.circom` (see Listing 1) is compiled. This test circuit instantiates the `ECDSAPrivToPub` template, selecting 4 chunks of 64 bits each, allowing the representation of up to 256-bit values as inputs.

```
pragma circom 2.0.2;
```

```
include "../../circuits/ecdsa.circom";

component main {public [privkey]} = ECDSAPrivToPub(64, 4);
```

Listing 1: `ecdsa_priv2pub.test.circom`

Five random test cases are generated as we can observe in Listing 2, from line 13 to 17. The `privateKey` for each test case is also randomly generated using the `utils.randomPrivateKey()` function provided by the `@noble/secp256k1` library. The corresponding `pubKey` is derived from the private key using the `Point.fromPrivateKey(privateKey)` function. Each `test_case` array includes the `privateKey` and the two coordinates of the public key: `pubKey.x` and `pubKey.y`. These components are then converted into the required input format, consisting in four chunks of 64 bits each.

```
1  describe("Test ECDSAPrivToPub", function () {
2
3  this.timeout(1000 * 1000);
4
5  let circuit; // Variable to store the compiled circuit.
6  let test_cases = []; // Array to store generated test cases.
7
8  before(async function () {
9    circuit = await wasm_tester(path.join(__dirname, "test",
10   "circuits", "ecdsa_priv2pub.test.circom"));
11 });
12
13 for (var test = 0; test < 5; test++) {
14   let privateKey = utils.randomPrivateKey();
15   let pubKey = Point.fromPrivateKey(privateKey);
16   test_cases.push([privateKey, pubKey.x, pubKey.y]);
17 }
18
19 var test_priv2pub_instance = function (test_case, idx) {
20   n = 64;
21   k = 4;
22
23   let privKey = Uint8Array_to_bigint(test_case[0]);
24   let pubKeyX = test_case[1]; // Public key x-coordinate.
25   let pubKeyY = test_case[2]; // Public key y-coordinate.
26
27   var privKey_tuple = bigint_to_array(n, k, privKey);
28   var pubKeyX_tuple = bigint_to_array(n, k, pubKeyX);
29   var pubKeyY_tuple = bigint_to_array(n, k, pubKeyY);
```

Listing 2: Obtaining public and private keys of each test case using the `@noble/secp256k1` library

Next, we verify that the coordinates of the `pubKey` calculated by the `ECDSAPrivToPub` template, using the witness calculator in WASM, match the public key coordinates generated by the `@noble/secp256k1` library, which are assumed to be correct (see Listing 3). At the end, we check that all the imposed constraints within the circuit are satisfied for this witness, in order to ensure consistency.

```
it(`Test ${idx}: Correct ECDSAPrivToPub.`, async function () {

  let witness = await circuit.calculateWitness({"privkey":
    privKey_tuple});

  expect(witness[1]).to.equal(pubKeyX_tuple[0]);
  expect(witness[2]).to.equal(pubKeyX_tuple[1]);
  expect(witness[3]).to.equal(pubKeyX_tuple[2]);
  expect(witness[4]).to.equal(pubKeyX_tuple[3]);
  expect(witness[5]).to.equal(pubKeyY_tuple[0]);
  expect(witness[6]).to.equal(pubKeyY_tuple[1]);
  expect(witness[7]).to.equal(pubKeyY_tuple[2]);
  expect(witness[8]).to.equal(pubKeyY_tuple[3]);


  await circuit.checkConstraints(witness);
});
```

Listing 3: Checking that the public keys are correct and constraints.

ECDSAVerify Test

Before running tests, the circuit `ecdsa_verify.test.circom` (see Listing 4) is compiled. This test circuit instantiates the `ECDSVerifyNoPubkeyCheck` template, selecting 4 chunks of 64 bits each, allowing the representation of up to 256-bit values as inputs.

```
pragma circom 2.0.2;

include "../../circuits/ecdsa.circom";

component main {public [r, s, msghash, pubkey]} =
ECDSAVerifyNoPubkeyCheck(64, 4);
```

Listing 4: `ecdsa_verify.test.circom`

Given the large size of the circuit used to verify the ECDSA signature, we optimise the witness computation process by employing the C implementation of the witness computation program, which necessitates the use of `c_tester`. In Listing 5 we can observe that to streamline testing and avoid recompiling the circuit for each test, the circuit is precompiled and stored in its designated path (`test/bin`), where it is assigned to the `circuit` variable. With the circuit precompiled, the configuration is set to `compile: false, recompile: false`.

```
const c_tester = circom_tester.c;
// ...
before(async function () {
  circuit = await c_tester(
  path.join(__dirname, "test", "circuits", "ecdsa_verify.test.circom"),
  {
    output: path.join(__dirname, "test", "bin"),
    compile: false,
    recompile: false
  }
```

```
  );
});
```

Listing 5: Precompiling the circuit with C

The strategy involves generating five random private keys, deriving their corresponding public keys, and signing a predefined message (see Listing 6). Each message is hashed using the `sha256` function, with the resulting hash reduced modulo the curve's base field, `CURVE.P`, to ensure it falls within the valid range. For simplicity, the message to be signed corresponds to the index of each private key in the array of test cases. As the signature output is a concatenation of the `r` and `s` values, these components will be split into separate values for further processing.

```
var test_ecdsa_instance = function (test_case, idx) {
  let n = 64;
  let k = 4;
  let privKey = test_case[0];
  let pubKeyX = test_case[1];
  let pubKeyY = test_case[2];

  it(`Test ${idx}: Correct ECDSA Signature.`, async function () {
    let msghash_bigint =
    Uint8Array_to_bigint(await utils.sha256(idx.toString()))%CURVE.P
    let msghash = bigint_to_Uint8Array(msghash_bigint);
    let sig = await sign(msghash, privKey,
    {canonical: true, der: false})
    let r = sig.slice(0, 32);
    let r_bigint = Uint8Array_to_bigint(r);
    let s = sig.slice(32, 64);
    let s_bigint = Uint8Array_to_bigint(s);

    let r_array = bigint_to_array(n, k, r_bigint);
    let s_array = bigint_to_array(n, k, s_bigint);
    let msghash_array = bigint_to_array(n, k, msghash_bigint);
    let pub0_array = bigint_to_array(n, k, pubKeyX);
    let pub1_array = bigint_to_array(n, k, pubKeyY);
  }
}
```

Listing 6: Obtaining the inputs for testing a correct signature.

In Listing 7 the signature is validated. For testing a valid signature case, we set the output signal to 1 and store it in a variable named `res`, which will be modified to 0 for the incorrect cases. We then compute the witness for the given input values and confirm that the circuit's (`ECDSAVerifyNoPubkeyCheck`) output signal, `result`, matches the expected value of 1. Finally, as with all cases, we ensure that all constraints imposed within the circuit are satisfied for this witness.

```
let res = 1n;

let witness = await circuit.calculateWitness({
  "r": r_array,
  "s": s_array,
  "msghash": msghash_array,
  "pubkey": [pub0_array, pub1_array]
```

```
});

expect(witness[1]).to.equal(res);
await circuit.checkConstraints(witness);
});
```

Listing 7: Checking the signature via the `ECDSAVerifyNoPubkeyCheck` template.

In the final part of the test (Listing 8), an incorrect signature is verified to ensure that the test appropriately responds when the signature is invalid.To evaluate the circuit's behaviour with an incorrect signature, we generate an invalid signature and set the expected result to 0 by modifying the `res` variable. To invalidate the signature, we increment the `r` value by 1, making it incorrect. The witness is then computed for the modified input values, and we verify that the circuit's output signal, `result`, correctly matches the expected value of 0.

```
it(`Test ${idx}: Incorrect ECDSA Signature.`, async function () {
  let msghash_bigint =
  Uint8Array_to_bigint(await utils.sha256(idx.toString()))%CURVE.P
  let msghash = bigint_to_Uint8Array(msghash_bigint);
  let sig = await sign(msghash, privKey,
  {canonical: true, der: false})
  let r = sig.slice(0, 32);
  let r_bigint = Uint8Array_to_bigint(r);
  let s = sig.slice(32, 64);
  let s_bigint = Uint8Array_to_bigint(s);
  let r_array = bigint_to_array(n, k, r_bigint + 1n);
  // Modified 'r'(incorrect)
  let s_array = bigint_to_array(n, k, s_bigint);
  // Keep 's' unchanged.
  let msghash_array = bigint_to_array(n, k, msghash_bigint);
  let pub0_array = bigint_to_array(n, k, pubKeyX);
  let pub1_array = bigint_to_array(n, k, pubKeyY);
  let res = 0n;

  let witness = await circuit.calculateWitness({
    "r": r_array,
    "s": s_array,
    "msghash": msghash_array,
    "pubkey": [pub0_array, pub1_array]
  });

  expect(witness[1]).to.equal(res);
  await circuit.checkConstraints(witness);
});
```

Listing 8: Testing an incorrect ECDSA signature.

3.3 BLS Signature

The BLS signature is a digital signature algorithm that uses **bilinear pairings** and is typically implemented using the `BLS12-381` elliptic curve, which is optimised for pairing operations. `BLS12-381` is a **pairing-friendly elliptic curve** designed for pairing-based cryptography. Over the past two decades, this field has

enabled innovations such as compact, aggregatable digital signatures, identity-based cryptography, and efficient polynomial commitment schemes like KZG commitments. Pairing-friendly curves are rare and require specific properties, such as a favourable embedding degree and a large prime-order subgroup. This curve is defined over a finite field of prime order q, where q is a 381-bit number.

3.3.1 Pairing Based Cryptography

Recall that a pairing is a function that maps two elements from specific groups to an element in a target group, enabling complex cryptographic operations. Formally, a pairing is defined as a function

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T,$$

where $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are cyclic groups of the same prime order r. These groups are typically derived from elliptic curves in cryptographic contexts.

The pairings used in cryptography are the **bilinear pairings**, and must satisfy the following properties:

1. **Bilinearity**: The pairing is linear in both inputs, meaning for all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$, and scalars $a, b \in \mathbb{Z}_p$:

   $$e(aP, bQ) = e(P, Q)^{ab}$$

   This property ensures that scalar multiplications on the inputs are preserved in the pairing's output.

2. **Non-degeneracy**: The pairing is non-trivial, meaning there exist $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$ such that

   $$e(P, Q) \neq 1_{\mathbb{G}_T}$$

   This guarantees that the pairing conveys meaningful information.

3. **Efficient Computation**: It is computationally feasible to evaluate $e(P, Q)$ for all $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.

Cryptographic pairings are typically constructed using pairing-friendly elliptic curves, which possess specific characteristics to enable efficient pairing computations. These curves are defined over a base field $\mathbb{F}_q$, which is a finite field of large prime order q. The elliptic curve $E(\mathbb{F}_q)$ includes a subgroup of prime order r, where r divides $(q^k - 1)$. Here, k, known as the embedding degree, is a small positive integer that determines the extension field $\mathbb{F}_{q^k}$ in which the pairing is evaluated.

In the case of `BLS12-381`, the elliptic curve is defined over a finite field of prime order q, where q is a 381-bit number and the embedding degree of the curve is 12.

A bilinear pairing takes as input two points, each belonging to different groups of the same prime order r, which is a large number. However, our base curve $E(\mathbb{F}_q)$ has only one subgroup of order r, making it unsuitable for defining a pairing directly.

To solve this inconvenience and enable pairing operations, the field is extended to $\mathbb{F}_{q^{12}}$, where the curve has additional subgroups of order r. The value 12 represents the embedding degree of the curve, which allows for the definition of pairings.

Arithmetic over $\mathbb{F}_{q^{12}}$ is computationally expensive. To mitigate this, a **twist** (a form of coordinate transformation) is applied, redefining the curve $\mathbb{F}_{q^{12}}$ over $\mathbb{F}_{q^2}$. This transformation simplifies arithmetic while preserving the essential subgroups of order r.

At the end, we are working with these two groups of order r:

$$\mathbb{G}_1 \subset E(\mathbb{F}_q),$$
$$\mathbb{G}_2 \subset E'(\mathbb{F}_{q^2}),$$

where E' denotes the twisted curve.

Hence, in our case, the pairing function is defined as follows:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$$

where $\mathbb{G}_T \subset E(\mathbb{F}_{q^{12}})$

3.3.2 The Signature

The signature scheme works as follows:

First, Alice generates her cryptographic keys. She randomly selects her **private key**, denoted by x, which is an integer chosen uniformly from the range $\{1, \ldots, q-1\}$. Using this private key, Alice can compute her corresponding **public key**. This is achieved by performing scalar multiplication of x with the generator point G_1 of the elliptic curve subgroup $\mathbb{G}_1$. The result, $\mathsf{pk} = x \cdot G_1 \in \mathbb{G}_1$, is shared as her public key while the private key x remains secret.

When Alice wants to sign a message m, she first hashes the message to map it to an element of the elliptic curve subgroup $\mathbb{G}_2$. This is done using a cryptographic hash function H defined as $H : \{0,1\}^* \rightarrow \mathbb{G}_2$, which takes any arbitrary-length message and produces a point $h = H(m) \in \mathbb{G}_2$.

The **signature** of the message is then computed using Alices private key X. Specifically, she multiplies the hash h by her private key to produce the signature $\sigma = x \cdot h \in \mathbb{G}_2$. This signature σ is a point on the elliptic curve and serves as a proof that Alice, who knows the private key x, signed the message.

To verify the signature, anyone with access to Alices public key (pk) and the signed message m can check its validity using the following pairing-based equation:

$$e(\mathsf{pk}, h) = e(G_1, \sigma).$$

Where e is the bilinear pairing function that we have described in the previous subsection.

The correctness of the verification process is guaranteed by the mathematical properties of the pairing function. Expanding the equation step by step:

$$\begin{aligned} e(\mathsf{pk}, h) &= e(x \cdot G_1, h) && \text{(substituting } \mathsf{pk} = x \cdot G_1\text{)}, \\ &= e(G_1, h)^x && \text{(using the bilinearity property of pairings)}, \\ &= e(G_1, x \cdot h) && \text{(reapplying bilinearity)}, \\ &= e(G_1, \sigma) && \text{(substituting } \sigma = x \cdot h\text{)}. \end{aligned}$$

3.3.3 CIRCOM Templates

Now, we want to test the CIRCOM template for the BLS signature verification present in the Yi-Sun's pairings repository [Sc23].

To implement the BLS Signature, it is necessary to develop a range of computational templates, each designed to address specific requirements. These components are integral to ensuring the cryptographic operations are carried out efficiently and accurately. The different types of templates used are:

First, templates for managing operations with **BigInt** numbers must be developed. Handling large integers is a cornerstone of cryptographic systems, as these numbers underpin operations such as key generation, signature computation, and verification. These templates will provide the functionality to perform arithmetic operations, modular reductions, and other necessary manipulations with large integers.

Secondly, it is essential to implement support for operations in the finite fields $\mathbb{F}_q, \mathbb{F}_{q^2}$ and $\mathbb{F}_{q^{12}}$. These fields play a critical role in our specific signature, as the elliptic curve and pairing computations involve arithmetic in these specific fields. Each field requires distinct optimisations to ensure operations such as addition, multiplication, and inversion are performed efficiently.

Additionally, templates must be designed to facilitate efficient interaction with the `BLS12-381` elliptic curve, one of the most widely used curves in modern cryptography. These templates should support basic curve operations, such as point addition and scalar multiplication, as well as more advanced functionalities, such as deriving public keys or mapping hash outputs to points on the curve.

Finally, the implementation of the (optimal Ate) **bilinear pairing** over the `BLS12-381` curve is crucial. As we have seen, the bilinear pairing is the core operation that enables signature verification in the BLS scheme.

Each of these components contributes to a robust and efficient implementation of the BLS Signature, addressing both the mathematical and computational challenges inherent in the signature test.

`CoreVerifyPubkeyG1NoCheck`

Another important template is the `CoreVerifyPubkeyG1NoCheck`. As the name says, it verifies the signature, but does not ensure that the public key is properly derived from the private key.

The verification template for the BLS signature scheme is designed to validate the correctness of signatures using bilinear pairings. It achieves this by leveraging mathematical properties of pairings to confirm the equivalence between the signed message and the provided signature. The process and considerations involved are as follows:

The fundamental idea of the template is to verify the equation $e(G_1, \sigma) \cdot e(\mathsf{pk}, -h) = 1$ This equation is mathematically equivalent to $e(G_1, \sigma) = e(\mathsf{pk}, h)$, which serves as the foundation for signature verification. By confirming this equality, the verifier can ensure that the signature σ was generated using the private key corresponding to the public key (pk), for the given message hash $h = H(m)$.

The template accepts three key inputs: the public key (pk), the hash of the message ($H(m)$), and the corresponding signature (σ). The output of the template is either 1, indicating that the signature is valid, or 0', indicating that it is not.

The verification procedure consists of the following steps:

1. The first step involves computing the negation of the message hash, $-h = -H(m)$. This negation is required for the pairing computation.
2. Next, the bilinear pairings $e(G_1, \sigma)$ and $e(\mathsf{pk}, -h)$ are calculated. These pairings are critical to the verification process, as they encapsulate the relationship between the signature, the public key, and the message hash.
3. The two pairings are then multiplied together, resulting in $e(G_1, \sigma) \cdot e(\mathsf{pk}, -h)$.
4. Finally, the result of the multiplication is checked against the value 1. If the equation holds true, the signature is valid; otherwise, it is invalid.

The template operates under specific assumptions, referred to as `NoCheck`, which streamline the verification process by relying on the correctness of the inputs. These assumptions include the following:

- The elliptic curve points provided as inputs are assumed to correctly belong to the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ as required.
- Input arrays are expected to represent valid elements of the finite field $\mathbb{F}_q$.
- The hash of the message, $H(m)$, must not correspond to the point at infinity, as this would invalidate the pairing computations.

To enhance the robustness of the system, the `CoreVerifyPubkeyG1` template performs additional checks to ensure the validity of the inputs. This goes beyond simply verifying the signature by confirming that the public key and message hash adhere to all necessary constraints.

Furthermore, while the implicit checks within the underlying templates for operations in $\mathbb{G}_1$ and $\mathbb{G}_2$ may be sufficient to validate the group membership of the inputs, these additional safety measures act as a safeguard. They help ensure that the inputs are properly structured and belong to the correct algebraic groups, thereby mitigating risks associated with malformed or maliciously crafted inputs.

In conclusion, this verification template combines the mathematical rigour of bilinear pairings with practical input validation to provide a robust and efficient mechanism for validating BLS signatures. Its design ensures that both the cryptographic computations and the integrity of the inputs are adequately addressed.

3.3.4 BLS Signature Test

Again, before running tests, the circuit `bls_signature.test.circom` (see Listing 9) is compiled. This circuit initiates the `CoreVerifyPubkeyG1NoCheck` template, selecting 7 chunks of 55 bits each, allowing the representation of up to 385-bit values as inputs.

```
include "../../circuits/bls_signature.circom";

component main = CoreVerifyPubkeyG1NoCheck(55, 7);
```

Listing 9: `bls_signature.test.circom`

The testing strategy for the implementation is similar to that used in ECDSA and involves the following steps:

To begin, the circuit is also compiled with C for optimizing the witness computation process, like we did with the ECDSA Test. Then, five random private keys are generated, as we can observe in lines from 17 to 20 in Listing 10. For each private key, the corresponding public key is derived, and a predefined message is signed using the `js` package `@noble/bls12-381`. This package provides functionalities specifically designed for the `BLS12-381` elliptic curve and operates in a manner similar to `@noble/secp256k1`, which is commonly used for ECDSA on the `secp256k1` curve.

The computed private keys, public keys, and signatures are then used as inputs for the CIRCOM circuit.

Finally, the outputs of the CIRCOM circuit are verified to ensure they match the expected results (see line 24 in Listing 11).

```
1  describe("Test Correct BLS Signature", function () {
2
3    let circuit;
4    let test_cases = [];
```

```
5
6   before(async function () {
7     circuit = await c_tester(
8     path.join(__dirname, "test", "circuits", "bls_signature.test.circom"),
9     {
10      output: path.join(__dirname, "test", "bin"),
11      compile: false,
12      recompile: false
13    }
14    );
15  });
16
17  for (var test = 0; test < 5; test++) {
18    let privateKey = utils.randomPrivateKey();
19    let pubKey = point_to_bigint(PointG1.fromPrivateKey(privateKey));
20    test_cases.push([privateKey, pubKey[0], pubKey[1]]);
21  }
```

Listing 10: Precompiling the circuit with C and generating the test cases

In order to simplify the testing process, the message to be signed is chosen as the `idx` (index) of each private key within the array of test cases. This approach ensures simplicity and consistency across the test cases.

The `PointG2.hashToCurve()` function is used to map the message bytes deterministically to a point on the elliptic curve in $\mathbb{G}_2$. This function serves as the hash function $H(\cdot)$, producing the message hash required for signature generation.

To maintain compatibility with the input requirements of the CIRCOM circuit, the inputs must be divided into smaller chunks. This chunking process ensures that the large numerical values can be handled appropriately within the circuit's constraints (lines 5 to 14 in Listing 11). Subsequently, in lines 16 to 20, the processed values, now in the correct format, are provided as inputs to the CIRCOM circuits.

It is important to note that, because elements of $\mathbb{G}_2$ are represented within the finite field extension $\mathbb{F}_{q^2}$, two coordinates are required to fully describe each point in $\mathbb{G}_2$.

```
1  it(`Test ${idx}: Correct BLS Signature.`, async function () {
2    let res = 1n;
3    let Hm = await PointG2.hashToCurve(utils.stringToBytes(idx.toString()));
4    let signature = await sign(Hm, privKey);
5    let pubKeyXArray = bigint_to_array(n, k, pubKeyX);
6    let pubKeyYArray = bigint_to_array(n, k, pubKeyY);
7    let HmX = bigint_to_array(n, k, Hm.toAffine()[0].c0.value);
8    let HmXi = bigint_to_array(n, k, Hm.toAffine()[0].c1.value);
9    let HmY = bigint_to_array(n, k, Hm.toAffine()[1].c0.value);
10   let HmYi = bigint_to_array(n, k, Hm.toAffine()[1].c1.value);
11   let signatureX = bigint_to_array(n, k, signature.toAffine()[0].c0.value);
12   let signatureXi = bigint_to_array(n, k, signature.toAffine()[0].c1.value);
13   let signatureY = bigint_to_array(n, k, signature.toAffine()[1].c0.value);
14   let signatureYi = bigint_to_array(n, k, signature.toAffine()[1].c1.value);
15
16   const input = {
17     "signature": [[signatureX, signatureXi], [signatureY,
18     signatureYi]],
19     "Hm": [[HmX, HmXi], [HmY, HmYi]],
```

```
    "pubkey": [pubKeyXArray, pubKeyYArray]
  };

  const wtns = await circuit.calculateWitness(input);
  expect(wtns[1]).to.equal(res);
});
```

Listing 11: Testing a correct BLS signature.

To verify an **incorrect signature** let's look at Listing 12, a distinct approach is utilised to generate a signature that is invalid while maintaining the mathematical integrity of the elliptic curve point. Instead of directly altering the signature, the incorrect signature is derived using a different message hash, referred to as `BadHm`. This hash does not correspond to the original message intended for signing.

The incorrect hash `BadHm` is produced by hashing the index incremented by 1 (see line 4 in Listing 12), rather than the original index associated with the message. This ensures that the resulting signature is based on a different input while still adhering to the rules of the hashing and signing process.

Unlike in ECDSA, where adding a value to one of the signature's coordinates can create an invalid point, this approach avoids such direct alterations. Modifying the coordinates of a point in $\mathbb{G}_2$ would result in a point that no longer belongs to the group $\mathbb{G}_2$, invalidating the test entirely since group membership checks would fail.

Instead, using `BadHm` generates a signature that appears well-formed mathematically but does not correspond to the correct message. When this signature is verified, the output will correctly return 0, indicating that the signature is invalid for the given public key and message hash. This method ensures robust testing of the signature verification process.

```
1  it(`Test ${idx}: Incorrect BLS Signature.`, async function () {
2    let res = 0n;
3
4    let badHm = await PointG2.hashToCurve(utils.stringToBytes((idx + 1).toString()));
5    let Hm = await PointG2.hashToCurve(utils.stringToBytes(idx.toString())
6    );
7    let signature = await sign(badHm, privKey);
8
9    // Same code as before
10
11   const input = {
12     "signature": [[signatureX, signatureXi], [signatureY,
13     signatureYi]],
14     "Hm": [[HmX, HmXi], [HmY, HmYi]],
15     "pubkey": [pubKeyXArray, pubKeyYArray]
16   };
17
18   const wtns = await circuit.calculateWitness(input);
19
20   expect(wtns[1]).to.equal(res);
21 }
```

Listing 12: Testing an incorrect BLS signature.

Conclusion

This project has provided a comprehensive exploration of elliptic curve cryptography, emphasizing both its theoretical foundations and practical applications. The mathematical structures underlying elliptic curves, particularly their group properties, have proven instrumental in constructing secure and efficient cryptographic protocols. These properties are not only elegant in theory but also highly effective in real-world systems, as demonstrated through the analysis of ECDSA and BLS signature schemes.

The exploration of the Pinocchio protocol has added another dimension to this work, showcasing how advanced cryptographic proof systems can leverage elliptic curve properties for succinct and verifiable computations. Understanding the principles behind Rank-1 Constraint Systems (R1CS) and their practical application within Pinocchio has been a valuable part of this research, illustrating the potential for these techniques to enhance privacy and efficiency in various computational contexts.

At the same time, the practical implementation of these concepts has been an enriching learning experience. Working with tools like CIRCOM to define and test cryptographic circuits has deepened my understanding of the precision and optimization required to build secure systems. This practical aspect of the work has allowed me to connect abstract mathematical principles with tangible applications, reinforcing the importance of bridging theory with practice.

In summary, this project has not only advanced my knowledge of elliptic curve cryptography and cryptographic proof systems but also provided valuable insights into the broader interplay between mathematics and technology.

Bibliography

[AS92] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2–13. IEEE, 1992.

[BGG+18] Sean Bowe, Ariel Gabizon, Matthew Green, Ian Miers, Mary Maller, and Pratyush Mishra. Scalable zero knowledge via cycles of elliptic curves. In *IEEE Symposium on Security and Privacy (SP)*, pages 145–162, 2018.

[BMIMT+23] 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*, 20(6):4733–4751, 2023.

[CHM+19] Alessandro Chiesa, Yan Huang, Ian Miers, Qipeng Liu, Nicholas Spooner, and Pratyush Mishra. Marlin: Preprocessing zksnarks with universal and updatable srs. *IACR Cryptology ePrint Archive*, 2019:1047, 2019.

[FKST94] Lance Fortnow, John C. Kilian, Madhu Sudan, and Luca Trevisan. The power of probabilistic proof systems. *Journal of the ACM (JACM)*, 41(4):721–759, 1994.

[G+08] Oded Goldreich et al. Probabilistic proof systems: A primer. *Foundations and Trends® in Theoretical Computer Science*, 3(1):1–91, 2008.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304. ACM, 1985.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptology ePrint Archive*, 2016:260, 2016.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive

arguments of knowledge. *IACR Cryptology ePrint Archive*, 2019:953, 2019.

[ic23] iden3 and contributors. circom: Circuit compiler for zk-snarks. https://github.com/iden3/circom, 2023. Accessed: 2025-01-10.

[Sc23] Yi Sun and contributors. Circom pairing: Circom templates for pairing-based cryptography. https://github.com/yi-sun/circom-pairing, 2023. Accessed: 2025-01-10.

[Was08] Lawrence C Washington. *Elliptic curves: number theory and cryptography*. Chapman and Hall/CRC, 2008.

[xc23] 0xPARC and contributors. Circom-ecdsa: Circom templates for ecdsa signatures. https://github.com/0xPARC/circom-ecdsa/tree/master, 2023. Accessed: 2025-01-10.