

Blockchain Cryptography - Demystified

Part - I

(Hashing, ECC, ECDSA, Bilinear Pairing, zk-SNARK (Groth16 and PLONK),
Merkle Tree, Verkle Tree & zk-STARK)

S. Venkatesan, IIIT Allahabad
Sandeep Kumar Shukla, IIIT Hyderabad
Sathya Priya S, SRMIST, Ramapuram, Chennai

October 2025

Acknowledgment: This document is based on inputs from various sources including ChatGPT, GeminiAI, Wikipedia, Medium.com (Vitalik Buterin, Encoding Labs, Bhaskar Krishnamachari, etc.), StackExchange, Lecture Video of Dan Boneh (Stanford University), Risen Crypto, Groth16, KZG, Lecture Video of Eli Ben-Sasson, etc. We thank all authors and sources for providing their resources.

If any comments, mail to venkat@iiita.ac.in

Contents

1	Overview	4
2	Introduction	5
3	Preliminary	5
3.1	Number Theory	5
3.2	Group	5
3.2.1	Finite Group	6
3.2.2	Finite Group and Cyclic Group	7
3.3	Field	7
3.3.1	Finite Field	8
3.3.2	Extension Field	8
3.4	Ring	10
3.5	Why group defined over Field?	11
3.6	Why Prime?	12
4	Hash Function	12
4.1	Merkle-Damgård Construction	13
4.2	Length Extension Attack	14
4.2.1	Attack Possibility	15
4.3	Wide-Pipe Construction	17
4.4	Sponge Construction	17
5	Elliptic Curve	18
5.1	Discriminant	19
5.2	Weierstrass Equation	19
5.3	Elliptic Curve Construction	20
5.4	Discriminant Example	20
5.5	Group Operation	20
5.5.1	Slope	20
5.5.2	Point Addition	21
5.5.3	Point Doubling	22

5.5.4	Point of Infinity	23
5.6	Mirroring	23
5.7	Elliptic Curve Example	24
5.8	Why EC for Cryptography	28
6	Digital Signature	29
7	ECDSA	29
7.1	Parameter Computation	29
7.2	Generation	30
7.3	Verification	31
7.4	Proof of Correctness	31
7.5	Public Key Recovery	32
7.6	Private Key Recovery	33
8	Bilinear Pairing	34
8.1	Preliminaries	34
8.1.1	Torsion Point	34
8.1.2	Divisor	34
8.1.3	Pairing and Mapping	36
8.1.4	Linear Pairing	36
8.1.5	Bilinear Pairing	36
8.1.6	Why Bilinear pairing for Cryptography	37
8.1.7	Properties of Bilinear Pairing	37
8.1.8	Different Bilinear Mappings	38
8.1.9	Twist Curve	38
8.2	Symmetric Pairing	40
8.2.1	Supersingular Curve	41
8.3	Asymmetric Pairing	41
8.3.1	Summary of types of pairing	43
8.4	Group Homomorphism	44
8.5	Consequence of Pairing	44
8.5.1	Decisional Diffie-Hellman (DDH) problem	44
8.5.2	Discrete Log Problem (DLP)	44
8.5.3	Bilinear DDH (BDDH) Problem	44
8.6	Miller’s Algorithm	45
8.7	Weil Pairing	45
8.8	Tate Pairing	47
8.9	Linearity and Bilinearity	49
9	BLS (Boneh–Lynn–Shacham) Signature	50
9.1	Curve	50
9.2	How the hashing maps to group point	51
10	Zero-Knowledge	51
10.1	Zero-Knowledge Proof	51
10.2	Interactive Zero-Knowledge Protocol	51
10.3	Vanishing Polynomial	52
10.3.1	Zero Test	53
10.4	Commitment Scheme	54
10.5	Polynomial Commitments	55
10.5.1	KZG (Kate-Zaverucha-Goldberg) polynomial commitment	55
10.6	Schwartz–Zippel Lemma	59
10.6.1	Product Check	61
10.7	zk-SNARK	64
10.8	Groth-16 Implementation Process	65
10.8.1	Security Issues	72
10.8.2	Randomized Proof in zk-SNARKs	72
10.8.3	Solution to Issues in Randomized Proof	78

10.9	PLONK	79
10.9.1	Lipton's Trick for Permutation Checking	79
10.9.2	Prescribed Permutation Check	81
10.9.3	Implementation Process	83
10.10	Affine Coordinates	98
10.11	Projective Coordinates	98
10.12	zk-STARK	99
10.12.1	Fast Reed-Solomon Interactive Oracle Proof (FRI)	99
10.12.2	Merkle Tree	101
10.12.3	Membership Verification of Merkle Tree	102
10.12.4	Verkle Tree and Vector Commitment	104
10.12.5	Fiat-Shamir's Transformation	107
10.12.6	Implementation Process	108
10.12.7	Why Polynomial Commitments?	115
11	ZK-Rollup	115
12	Practical Experiment Questions	116
13	References	116
14	Annexure	117
14.1	Additional Pairing Properties and Relations	117

1 Overview

Blockchain technology is based on several fundamental computer science principles, such as data structures, cryptography, distributed systems, and computer networks. Cryptographic concepts such as hashing and digital signatures are implemented to ensure integrity, ownership, and authentication. Blockchain advancements, such as Layer 2 smart contracts, depend heavily on cryptographic techniques. This material thoroughly explains the important cryptographic concepts and techniques implemented in blockchain. The objective of this document is to explain these concepts in a simple, step-by-step manner so readers can easily understand them.

Blockchain implementation uses the cryptographic hashing to ensure the integrity of blocks, transactions and other data. In addition, hashing is used in the digital signature. The popular hashing algorithms such as MD-5, SHA-1, SHA-2 uses the Merkle Damgård construction, which is vulnerable for the length extension attack. However, the wide-pipe construction and the sponge construction is not susceptible for the length extension attack. The hash algorithms using the wide-pipe construction and the sponge construction is preferred in the Blockchain.

The digital signature is used in blockchain to ensure only the owner does the activity which means only the owner makes transaction for his/her cryptocurrency or data. Also, used to verify the integrity and ensure that the authorized party is performing the activity. The preferred digital signature algorithm in the Blockchain is Elliptic curve based Digital Signature Algorithm (ECDSA) because ensures security with less key size.

The advancement in Blockchain Technology, such as Layer 2 smart contract (certain implementations) needs privacy and that is achieved through the Zero-Knowledge Protocols. The Elliptic curve Bilinear pairing is preferred in some of the implementations to achieve the zero-knowledge. Even though normal Elliptic Curve algorithms are there, computationally heavy pairing is preferred since special algebraic property (bilinear mapping) can be achieved through pairing only.

Three types (Symmetric Pairing - Type I, Asymmetric Pairing - Type II and Type - III) of pairing can be applied according the the requirement however the type III pairing is more secure. To implement the pairing concept, we can use the Tate or Weil pairing or their variants however both uses the Miller's algorithm to solve the line equation efficiently.

The zk-SNARK is a zero knowledge method with properties of Succinctness, Non-Interactive and Arguments of Knowledge. The implementation of the zk-SNARK are Groth 16 and PLONK (others are also there) and these uses polynomial commitments to prove the proof to the verifier without disclosing the actual polynomial that contains the secret values. The KZG polynomial commitment is preferred and it needs the vanishing polynomial concept. The prover proves to the verifier the entire data however verification will be done on the random points may be one or two points. This allows the prover to modify the data and make the verifier to accept the modified data. However, the Schwartz-Zippel Lemma states that the chance of such acceptance is very low or very very low.

The zk-SNARK uses the polynomial commitments however the values are given as the input in the beginning. To bring the polynomial from the given inputs, the Lagrange's interpolation or Fast Fourier Transform can be used.

The Elliptic Curve based Cryptography and the factorization based Cryptography (RSA) are vulnerable for the quantum computers as shown by the Shor's algorithm hence, zk-STARK is preferred. The zk-STARK works based on the hashing and Merkle-Tree. The Merkle-tree is costly with respect to storage and hash value sharing. Hence, the verkle-tree based on the polynomial and vector commitment is preferred even though it is computationally heavy when compared with the Merkle-Tree. Even though zk-STARK is quantum secure, it is computationally heavy when compared with the zk-SNARK because of Merkle-Tree usage. The zero-knowledge protocols are both interactive and non-interactive however in blockchain non-interactive is preferred. The Fiat-Shamir's transformation is used in zk-SNARK and zk-STARK to achieve the non-interactive zero-knowledge proving.

The zk-Rollup uses the zk-SNARK or zk-STARK to achieve the zero-knowledge in the layer 2 smart contracts.

First, this material introduces the necessary abstract algebra, followed by the elliptic curve. Specifically, it covers elliptic curve cryptography: ECDSA, and pairing. The ECC application zk-SNARK is presented alongside the Groth 16 and PLONK implementations that includes the polynomial commitment (KZG), vanishing polynomial, permutation check, etc. Also discuss zk-STARK, which is a quantum-safe zero-knowledge protocol along with Merkle-Tree, Verkle-Tree and Fiat-Shamir's Transformation.

2 Introduction

The Elliptic Curve Cryptography (ECC) is the asymmetric key cryptography or public key cryptography preferred largely these days in applications. ECC is based on the algebraic structure of elliptic curves over finite fields. It is preferred in applications when compared with RSA algorithm because of less key size and better performance. ECC achieves the security of 128-bit with 256 bit key size but the same with RSA needs approximately 3072 bits. The 128 bit security means, 2^{128} attempts required to break the algorithm. The algorithm that needs less attempt to break ECC is Pollard's Rho algorithm by John Pollard in 1975. For example, a 256 or 512 ($N = 256$ or $N = 512$) bit curve can be broken in $\sqrt{256} \approx 2^{128}$ or $\sqrt{512} \approx 2^{256}$ attempts, however, infeasible to break in today's available hardware.

Elliptic curve is applied to the key agreement, digital signature and encryption in cryptography. For the key exchange: Elliptic Curve Diffie-Hellman (ECDH), digital signature: Elliptic Curve Digital Signature Algorithm (ECDSA) and encryption: Elliptic Curve Integrated Encryption Scheme using AES or other algorithms. The EC is also used in Bilinear pairing-based cryptography and to generate pseudo-random numbers.

3 Preliminary

The basics about the elliptic curve and its algebraic operations are discussed.

3.1 Number Theory

The number theory primarily focus on the Integers and arithmetic functions (additive and multiplicative). The reason behind using Integers in cryptography is that computers has the problem of precision in decimal numbers.

3.2 Group

A group is an algebraic structure which is widely used in number theory and contributes to the cryptography. The following is a group of integers ($Z, +$) that satisfies the group properties for addition. Here Z refers the set of integers.

$$G(Z, +) = \{.., -3, -2, -1, 0, 1, 2, 3, ..\}$$

Properties:

- Closure: For any $a, b \in Z$ and the sum that is $a + b$ will belongs to Z . Here Z is the Integers.
- Associativity: For all $a, b, c \in Z$, $(a + b) + c = a + (b + c)$ will satisfy.
- Identity Element: There exist an identity element 0, because for any $a \in Z$ then $a + 0 = 0 + a = a$.
- Inverse Element: For each $a \in Z$, there exists an inverse $-a$, so that $a + (-a) = 0$, gives identity element as the output.

Let us see an example, we take $a = 10$, $b = 12$ and $c = 4$ and all belongs to integer set Z

- Closure: $a + b = 10 + 12 = 22$, which is belongs to Integer that is Z

- Associativity: $(a + b) + c = a + (b + c)$, which is $(10 + 12) + 4 = 10 + (12 + 4)$
- Identity: $a + 0 = 0 + a = a$, which is $10 + 0 = 0 + 10 = 10$
- Inverse Element: $a + (-a) = 0$, which is $12 + (-12) = 0$

A group can be an **abelian** group if it supports the commutative property that is $a + b = b + a$ which is $10 + 12 = 12 + 10$ along with other discussed properties.

G(Z, ×): We can apply the multiplicative operation to achieve the multiplicative group. However, the element 0 has to be removed since the multiplication of 0 with any number will produce 0 and not satisfy the group property. The identity element in case of the multiplicative group is 1.

3.2.1 Finite Group

The finite group means number of elements in that group is finite that means size is known not infinite. The finite group has the limitation means the integer set cannot include more than the size defined. For an example, we can say that $(Z_n, +)$ refers that the group can have only n elements that is from 0 to n . However in case of the multiplicative group represented as $(Z_n, ×)$ or $(Z_n^*, .)$, will have the size not n and it will be $\varphi(n)$ which can be computed using the Euler-totient function.

Euler-Totient Function: This counts the number of positive integers less than or equal to n with coprime to n . Here coprime (relatively prime) means, the *GCD* of any number less than or equal to n should produce the output as 1. Greatest Common Divisor (GCD): It is the largest number that divides two (or more) integers without leaving a remainder. Let us see an example,

We take $n = 7$, which is prime then all values less than 7 will have GCD 1 because the largest integer that can divide both numbers without remainder are 1. Hence the number of elements or size is $\varphi(n) = n - 1$. If we choose the non prime n , then the size will be different however that is not required since we consider only prime numbers.

We take $(Z_n^*, .) = (Z_7^*, .) = \{1, 2, 3, 4, 5, 6\}$ without 0 because it is the multiplicative group and the size is $\varphi(n) = n - 1 = 7 - 1 = 6$ and check the properties:

- Closure: Take $a = 3$ and $b = 6$ then $a \times b$ is equal to $a \times b \text{ mod } 7$. Here the *mod* is applied since it is finite means the value should not go more than 6 means not more than the elements in set $(Z_7^*, .)$. Now $3 \times 6 \text{ mod } 7 = 9 \text{ mod } 7 = 2$. The element 2 is part of the set hence satisfies the closure property.
- Associativity: Take $a = 3$, $b = 6$ and $c = 2$ then $(3 \times 6) \times 2 = 3 \times (6 \times 2)$ which is $36 \text{ mod } 7 = 36 \text{ mod } 7 = 1$. Here 1 is part of the set and both side are equal hence satisfied.
- Identity: Take $a = 3$ and identity element is 1 hence multiplicative group so $3 \times 1 \text{ mod } 7 = 3$. Here 3 is part of the set hence satisfied.
- Inverse: Take $a = 3$ and the inverse can be computed as 3^{-1} that means $\frac{1}{3}$ which will give real number (including decimal number) and we do not have interest on it since we need only integers for computers therefore we have to consider $3^{-1} \text{ mod } 7$ that can be written as $3 \times X \equiv 1 \text{ mod } 7$. Here if we take $X = 5$ then $3 \times 5 \equiv 1 \text{ mod } 7$ and satisfies the equation (The inverse of 3 is 5). The X value should not go beyond $n - 1$ or it should be an element of the set. In case you choose X value beyond n then you have to apply modulus for that and reduce it so it will be less than n and member of the set. To compute the inverse on large numbers, it will be complex so we can apply the Extended Euclidean Algorithm for it. However, we will not discuss that in this document.
- Commutative: If we consider the group is abelian then this property should also be satisfied. Take $a = 3$ and $b = 6$ then $3 \times 6 \text{ mod } 7 = 6 \times 3 \text{ mod } 7$ are equal. Here satisfies the commutative property.

Important Note: There is finite group and modular group. In case of the finite group, it is finite set of elements in the group regardless of the operation (addition and multiplication) that will be used and when we say modular group then we apply the modular function. We can say it is a special type of finite group where the group operation is defined over modular arithmetic. In our previous example, we used the modular function since it is finite modular group.

3.2.2 Finite Group and Cyclic Group

In case of finite group, we can say

- Finite group is simply a group that has a finite number of elements.
- The number of elements in the group is called the order of the group.
- A finite group can be any group with a finite number of elements, and it may or may not be cyclic.
- Ex: $Z_5 = \{0, 1, 2, 3, 4\}$. Here the order of the group is 5 and it is additive modulo group means all elements and the output of all operations will be applied to modular operation that is (mod 5) in this example.

In case of cyclic group, we can say

- A cyclic group is a group that can be generated by a single element called generator (g).
- All elements are of the form g^0, g^1, g^2 and so on.
- The g will be chosen randomly however the order of g produced set must be exactly $\varphi(n)$. Let us say $G = (Z_5^\times, \times)$, here the order is $\varphi(n)$ that is $5 - 1 = 4$ (in case n is prime then $\varphi(n) = n - 1$, otherwise it will differ). Now, the g^0, g^1, g^2, g^3 and g^4 (we can ignore g^0) should produce the unique 4 elements otherwise that g is not qualified for generator and we have to choose another value for g . In case we go for a cyclic group $G = (Z_5^\times, \times)$ then g should be member of the group that generates other group elements.
- Cyclic groups can be finite or infinite. This $G = (Z_5^\times, \times)$ indicates the multiplicative group with 0 removed so we write with that form.
- The generate g should be the coprime of the order of the element which here is 5.

There is a difference between multiplication modulo 5 and the multiplicative group modulo 5 that is (Z_5, \times) in the above case. If it is multiplication modulo 5 then 0,1,2,3,4. In case of the multiplicative group modulo 5 that is (Z_5^\times, \times) then 1,2,3,4. Hence the notation has to be written properly. Hence we have to write the multiplicative group modulo as (Z_5^\times, \times) or (Z_5^\times) .

The above is given considering the multiplicative group and we can also write for the additive group.

3.3 Field

Like Group, Field $(F, +, \times)$ is another set satisfying the following properties.

- Closure: For any $a, b \in Z$, both $a + b$ and $a \times b$ will be in Z .
- Associativity: $a + (b + c) = (a + b) + c$ and $a \times (b \times c) = (a \times b) \times c$
- Commutativity: $a + b = b + a$ and $a \times b = b \times a$
- Identity elements: In case of Additive identity, it is 0 ($a + 0 = a$) and for multiplicative 1 ($a \times 1 = a$).
- Additive inverses: For every a , there exists $-a$ such that $a + (-a) = 0$. Here 0 is the additive identity element.
- Multiplicative inverses: For every $a \neq 0$, there exists a^{-1} such that $a \times a^{-1} = 1$. Here 1 is the Multiplicative identity element.
- Distributivity: It should satisfy $a \times (b + c) = a \times b + a \times c$. Here a, b, c are the elements of finite field.

Now the question is what is the different between Group and Field? Group support one operation for any set however Field support two operations (additive and multiplicative) for any set.

Now the next question, how a field can support both additive and multiplicative because if a set contains 0 then multiplicative fails? The zero element is the additive identity and it has no multiplicative inverse and hence multiplication cannot be defined on the whole set F , but the multiplicative group only includes $F \setminus \{0\}$ means it discards the 0 in the set and use all other elements or numbers. Hence we can say

- Addition (+) is an abelian group on all F including 0
- Multiplication (\times) is an abelian group on $F \setminus \{0\}$.

Now the next question, how the abelian group brings the field since it does not support distributive? If we take two abelian group one with the addition and another with multiplication then it supports distributivity. Now, from field we get both additive and multiplicative abelian groups so the elements of the abelian group will support the distributive. To understand clearly, refer the following:

- Take an Abelian group under addition.
- Add another Abelian group under multiplication (excluding 0).
- Now, this defines distributive property.
- We can this as a ring and if every nonzero element also has a multiplicative inverse then it is called a field.

Here, we get both abelian group one with addition and multiplication from the field so will support the distributive.

3.3.1 Finite Field

A finite field (also called a Galois field) is an algebraic set with a finite number of elements where we can perform the additive and multiplicative operation ensuring the field properties.

Multiplicative group over a finite field: Consider the set $F_p^* = \{1, 2, \dots, p-1\}$ (all non-zero elements of the finite field) forms a multiplicative group under multiplication modulo p . This group is also the abelian group. The elements included are the all non-zero elements of F_p . We cannot use 0 since we are defining the multiplicative group.

3.3.2 Extension Field

It is an extension of the base field.

$F_p \Rightarrow F_{p^m}$; where m is the degree of the extension. We can choose any value for this according to our application requirement.

We have to choose an irreducible polynomial of degree m . The irreducible polynomial means it should not be possible to factor. For example, $x^2 + 2x + 1$ should not be chosen because it is reducible to $(x + 1) \times (x + 1)$. However, if we see, the result will not be $x^2 + 2x + 1$ because if we apply modulus 2 considering finite field it will be different. Let us consider, F_2 , in this case $x^2 + 2x + 1 \pmod{2}$ is $x^2 + 1$ because $2x \pmod{2}$ or $2 \pmod{2}$ will be zero. However if we choose $\pmod{3}$ then it is reducible. Another important thing to note that we cannot consider the polynomial $x^2 + 2x + 1$ for $\pmod{2}$ because $2x$ will be out and it is $x^2 + 1$ so reducible $(x + 1) \times (x + 1)$.

Now let us move on computing the extension field elements. We refer the extension with the following representation:

$\mathbb{F}_{q^m} = \mathbb{F}_q[x]/(f(x))$; This equation says that if we consider two polynomials equivalent then their difference is divisible by $f(x)$. Here $f(x)$ is the irreducible over F_q . We have set of polynomials defined over $F_q[x]$ and divide those by an irreducible polynomial $f(x)$ and keep only the remainder, which has degree < 3 . Here, the 3 is degree of irreducible polynomial $f(x)$.

We are defining a quotient of a ring. $F_q[x]$ is the ring of polynomials with coefficients in F_q . $F_q[x]$ is a commutative ring with unity (here the unity refers that the ring has the multiplicative identity), but it is not a field since not all polynomials have inverses. **Ring is the algebraic structure like field however it may not have the multiplicative inverse.**

We take F_2 with the elements $\{0, 1\}$. Let us consider that we go for the extension field F_{2^3} . Now derive the extension field with the irreducible polynomial of degree 3. Let us consider

$$f(x) = x^3 + x + 1, \text{ which is irreducible over } F_2 \text{ that is with mod 2.}$$

We consider, $F_2^3 = F_2[x]/(x^3 + x + 1)$, this means we take all polynomials with coefficients in F_2 and treat any two polynomials as equal if their difference is divisible by $x^3 + x + 1$.

For better understanding, let us see that a polynomial $r(x)$ is divisible by $f(x)$ if there exists some polynomial $s(x)$ then

$$r(x) = f(x) \cdot s(x)$$

$$\text{If } p(x) - q(x) = f(x) \cdot s(x), \text{ then } p(x) \equiv q(x) \pmod{f(x)}$$

Here $s(x)$ is the quotient and we know that the quotient multiple with the divisor gives the actual value that is $r(x)$ provided that no remainder is there. The $p(x)$ and $q(x)$ are two different polynomials and their difference will be divisible by the $f(x)$. Let us see an example for this,

Let's take two polynomials in $\mathbb{F}_2[x]$:

$$p(x) = x^3 + x, \quad q(x) = 1$$

Calculate their difference:

$$p(x) - q(x) = x^3 + x - 1$$

In \mathbb{F}_2 , subtraction is the same as addition (because $-1 \pmod{2} = 1$), so:

$$p(x) - q(x) = x^3 + x + 1$$

Now note that:

$$f(x) = x^3 + x + 1$$

So:

$$p(x) - q(x) = f(x)$$

Which means the difference is divisible by the $f(x)$ (that is $\frac{p(x)-q(x)}{x^3+x+1}$ is $\frac{x^3+x+1}{x^3+x+1}$ without any remainder) and also we can write as follows because both the $p(x)$ and $q(x)$ remainder will be equal when we divide by the $f(x)$. :

$$p(x) \equiv q(x) \pmod{f(x)}$$

Or:

$$x^3 + x \equiv 1 \pmod{f(x)}$$

$p(x) \equiv q(x) \pmod{f(x)}$ is like integer modular arithmetic, for example we consider $p(x) = 17$ and $q(x) = 5$:

$$17 \equiv 5 \pmod{12} \text{ because } 17 - 5 = 12 \text{ is divisible by } 12$$

Here we will be dividing the polynomial with $x^3 + x + 1$. For example, we take x^3 and divide by $x^3 + x + 1$ then we will get $x^3 = x + 1$, hence we can replace x^3 with $x + 1$.

Example Each element in \mathbb{F}_{2^3} is a polynomial of degree less than 3. Let us take the following polynomial:

$a_0 + a_1x + a_2x^2$ with $a_i \in \mathbb{F}_2$ and that produces the following elements.

$$\begin{aligned} &0, \quad 1, \quad x, \quad x + 1, \\ &x^2, \quad x^2 + 1, \quad x^2 + x, \quad x^2 + x + 1 \end{aligned}$$

Let us see how the above elements are computed. We apply the value in $a_0 + a_1x + a_2x^2$ and compute the elements.

First, $a_0 = 0, a_1 = 0, a_2 = 0$, now $0 + 0 \times x + 0 \times x^2 \pmod 2 \Rightarrow 0$

Second, $a_0 = 0, a_1 = 0, a_2 = 1$, now $0 + 0 \times x + 1 \times x^2 \pmod 2 \Rightarrow x^2$

Third, $a_0 = 0, a_1 = 1, a_2 = 0$, now $0 + 1 \times x + 0 \times x^2 \pmod 2 \Rightarrow x$

Fourth, $a_0 = 0, a_1 = 1, a_2 = 1$, now $0 + 1 \times x + 1 \times x^2 \pmod 2 \Rightarrow x^2 + x$

Fifth, $a_0 = 1, a_1 = 0, a_2 = 0$, now $1 + 0 \times x + 0 \times x^2 \pmod 2 \Rightarrow 1$

Like the above we can compute all the elements.

Now the question is, how we use the variable (or special element not the variable) x in the computation because we can use 0 and 1 but not x directly. The answer is we can consider the binary representation for all the non value elements.

For example, we can have the binary equivalent for 0 as 000 (0), 1 as 001 (1), x as 010 (2), $x^2 + 1$ as 101 (5) and so on. This can be used to compute the functions may be points (provided if there is a matching in the x and y , this will be clear in Elliptic Curve topics and example) on the elliptic curve of extension field.

3.4 Ring

It is another algebraic structure with two binary operations, Addition and Multiplication, like a field. The Ring satisfies the following properties:

- $(R, +)$ is an abelian group that satisfies the following properties:
 - Closure: $a + b \in R$ for all $a, b \in R$. For example if we form a Ring $R = \{1, 2, 3, 4, 5, 6, 7\}$, take two from the set for example 3 and 4, the addition of the two (7) is R only considering the integers ($R = Z$, where Z refers integers).
 - Associativity: $(a + b) + c = a + (b + c)$
 - Identity: There exists $0 \in R$ such that $0 + a = a$
 - Inverse: For every $a \in R$, there exists $-a \in R$ such that $a + (-a) = 0$
 - Commutativity: $a + b = b + a$
- Multiplication is Associative: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Distributive: $a \cdot (b + c) = a \cdot b + a \cdot c$
- Other properties that will be satisfied based on the ring
 - Commutative ring: $a \cdot b = b \cdot a$ for all $a, b \in R$.
 - Ring with unity (unit ring): There exists $1 \in R$ such that $1 \cdot a = a \cdot 1 = a$ [This refers to the identity element. For example, the even integers are not having the identity element in multiplicative ring because 1 is the multiplicative identity but not part of the even integer ring so even integer ring $R(2Z, +, \cdot)$ does not have unity. Here $2Z$ is the even integer means any integer is multiplied by 2 to bring the even integers]

- Division ring (skew field): Every nonzero element has a multiplicative inverse (but multiplication may not be commutative)
- A commutative division ring. This means, the set will be a field.

Here, the division ring means, it should satisfy all the following properties:

- $(R, +)$ is an abelian group like the usual ring.
- Multiplication is associative:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c), \quad \forall a, b, c \in R.$$

For example, the integer set Z , $a = 2, b = 3, c = 5$ then $2 \cdot (3 \cdot 5) = 30$ and $(2 \cdot 3) \cdot 5 = 30$, here 30 is the part of the integer number set.

- There should be a multiplicative identity $1 \neq 0$.
- Every nonzero element $a \in R$ has a **multiplicative inverse** $a^{-1} \in R$ such that

$$aa^{-1} = a^{-1}a = 1.$$

- **In addition, if it supports the commutative property then we can call this as the Commutative Division Ring**

Difference: Group vs Ring vs Field:

- In case of Group, either additive or multiplicative operation is supported not both on the same set.
- In case of Field, both addition and multiplication are supported in the same set however the identity elements cannot support both hence we keep separate multiplicative group from the field that is $(F \setminus \{0\}, \times)$. This means, a set excluding 0 can be the multiplicative group.
- In case of ring, it will support both addition and multiplication however not all ring will have the multiplicative identity. Even it has the multiplicative identity, there is no guarantee that all elements will have the multiplicative inverse

Important Note: Why always addition and the multiplication operation is considered while describing the algebraic structure not the subtraction and division? Because subtraction and division are the derived operators. We can write $A - B = A + (-B)$ similarly $\frac{A}{B} = A \times B^{-1}$. But, it is not possible to bring the addition and multiplication from the subtraction and division. Hence, the addition and multiplication is the core operators

3.5 Why group defined over Field?

The elliptic curve defined over the field uses the group operation. Now, we will get the following questions:

- Why not, we use the field only? Ans: If we use the field, then inverse the number is easy and not complex so not secure. For example, if we know x and $y = k + x$ then you can easily bring $k = y - x$. Similarly, if $y = x \cdot k$ then $k = y/x$ is easy. Even, we consider F_p , modular inversion is efficient using Euclidean algorithm. However, the inverse in Group is complex. In Group it is complex because the inverse operation needs not only addition or only multiplication, it may need both along with subtraction and division and that is not supported. Hence, getting inverse using the Group alone is very hard.
- Why not, we use the Group directly? Ans: The Elliptic Curve has the point addition and point doubling laws and those need the addition, subtraction, division, and multiplication (you may understand why all these operations are needed while discussing point addition, slope and point doubling in coming sections). The curve defined over the Group cannot support all operations because it can give either addition (with subtraction) or multiplication (with division) with inverses. However, we need both so using the group defined over the Field is the choice.

3.6 Why Prime?

This is a natural number from 1 that has no positive divisors other than 1 and itself. Prime numbers are fundamental in cryptography because it has various features including the following:

- Primes make factorization computationally hard provided large prime integers. There is no polynomial time algorithm exist to find the prime factors of the composite number. The best known algorithms, such as the *General Number Field Sieve (GNFS)*, run in sub-exponential time, which grows extremely fast as N increases. Hence, hard to find the factors. We will not discuss here about the GNFS algorithm.
- Every whole numbers (integer greater than 0) can be written uniquely as a product of prime numbers. If we take 8 then it can be written as $2 \times 2 \times 2 = 2^3$ and if we take 19 then 1×19 and itself the prime number.
- Primes guarantees uniformity in group operations that is no unexpected repetitions or small cycles. For example, if we take non-prime 8 and compute $Z_8^* = 1, 3, 5, 7$ under multiplication modulo 8. If we pick 3 from that group and compute $3^1 \text{ mod } 8 = 3$, $3^2 \text{ mod } 8 = 1$, $3^3 \text{ mod } 8 = 3$, we see repetition quickly however with prime (for example, 7) it will be repeating after all the numbers (1,2,3..6) will be computed.
- Prime makes many assumptions valid. For example, prime orders r of the group G_1, G_2, G_T ensure that Decisional Diffie-Hellman (DDH) problem is hard (however depending on the group construction - for example, in pairing based cryptography, DDH problem will be hard in some groups not others).

You will understand the DDH problem and pairing based cryptography in future sections.

4 Hash Function

It is a cryptographic hash function that takes arbitrary size input and produces the fixed size output. We can see in equation 1, a function takes arbitrary size of input sequence with the combination of 0s and 1s and produces output of size n .

$$f\{0, 1\}^* \rightarrow \{0, 1\}^n \quad (1)$$

A cryptographic hash function should satisfy the following three properties:

- **Pre-Image Resistance:** Hash function should be an one-way function that means it should be computationally hard to reverse a hash function. Given a message M , the cryptography hash function f produces the output H . As given below, from the hash output H , it should be hard or infeasible to bring out the message M using the inverse function of f or f itself.

$$\begin{aligned} f\{M\} &\rightarrow H \\ M &\not\leftarrow f\{H\} \\ M &\not\leftarrow f'\{H\} \end{aligned}$$

- **Second Pre-Image Resistance:** Given an input and its hash output, it should be hard to find a different input with the same hash. For example, given input M_1 and its equivalent hash H_1 , it should be hard or infeasible to find message M_2 which will produce same hash output H_1 as shown in below.

$$\begin{aligned} f\{M_1\} &\rightarrow H_1 \\ f\{M_2\} &\rightarrow H_1 \end{aligned}$$

This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value.

- Collision Resistance: It should be hard to find two different inputs of any length that result in the same hash. Difference between second pre-image resistance and collision resistance is: In the second pre-image resistance, first message/ input m_1 and its equivalent hash value H_1 is fixed and attacker has to find a message m_2 , which will produce the output H_1 but in case of this (collision resistance) property, attacker is free to choose any two inputs, which can bring out same hash value. If any hash function is collision resistant then it is second pre-image resistant also.

In the cryptographic hash function, collision do exist because the output set is limited when the input set is high or infinite. For example, figure 1 shows the mapping of the input set (space) to the output set (space). The size of input set is higher than the output set. Hence, there will always be a possibility that collision will occur. According to Birthday paradox, if the output set size is n with possible 0, 1 combination then there is 50% chance that in $2^{\frac{n}{2}}$ input, collision will occur. This says, if we try computing hash for $2^{\frac{n}{2}}$ inputs then there is 50% or more chance the collision can occur. However, it is computationally hard to achieve the collision because the size of n will be high like 256 or 512 bits and values in output space has the equal probability of occurrence (is not biased - for example, if we say that the set $A = \{1, 2, 3, 4, 5\}$ has size 5 and occurrence of each element is $\frac{1}{5}$. If we see the set $B = \{4, 2, 4, 4, 5\}$ has size 5 and occurrence of each element is not $\frac{1}{5}$, in case of 4 it is $\frac{3}{5}$ and in case of 5 it is $\frac{1}{5}$ hence biased).

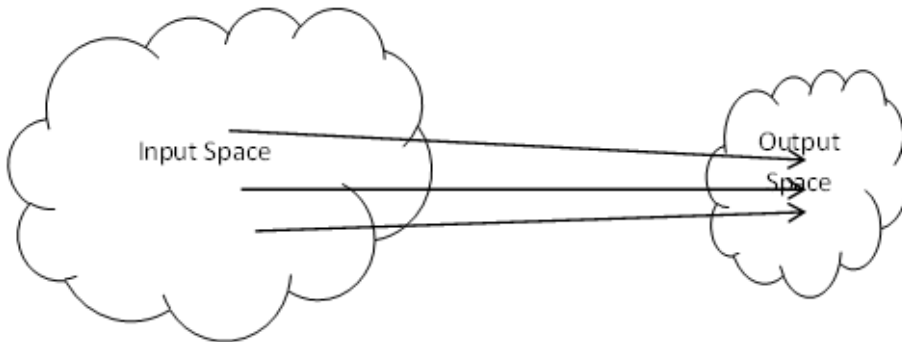


Figure 1: Hash function mapping

Verification: In verification also, the hash function is used to verify the integrity of the message. To verify the integrity, give the message and the hash output to a function as in equation 2, which will identify whether message is valid or not that is modified or not.

$$V(M, H) = \begin{cases} Valid & \text{If } f(M) == H \\ Invalid & \text{otherwise} \end{cases} \quad (2)$$

Assume, the sender computed the hash H for a message m and send to the receiver. Receiver uses the same hash function f , message m and hash output $H = f\{m\}$ verify whether the message m is same or modified. The verifier will take the message m and apply the hash function $H' = f\{m\}$ and compare the output with H . If $H = H'$ then message is not modified means integrity is preserved otherwise message is modified.

There are different hash functions exist, commonly used are Secure Hash Algorithm (SHA) versions. Other hash algorithms are Message Digest (MD5), RIPEMD, etc.

4.1 Merkle-Damgård Construction

The popular hash algorithms like MD5, SHA-1 and SHA-2 uses the Merkle-Damgård Construction. The Merkle-Damgård construction was initially described by Ralph Merkle's in Ph.D. thesis in 1979 and later Ralph Merkle and Ivan Damgård independently published a more formal and general proof of the construction's security in 1989. Since, both contributed it is referred as the Merkle-Damgård Construction. The construction is shown in figure 2 considering the SHA-256.

The process of the constructions is as follows:

- At first, the given input message will be partitioned into t multiple blocks of size N . In the figure 2, the size of N is 512 bits.

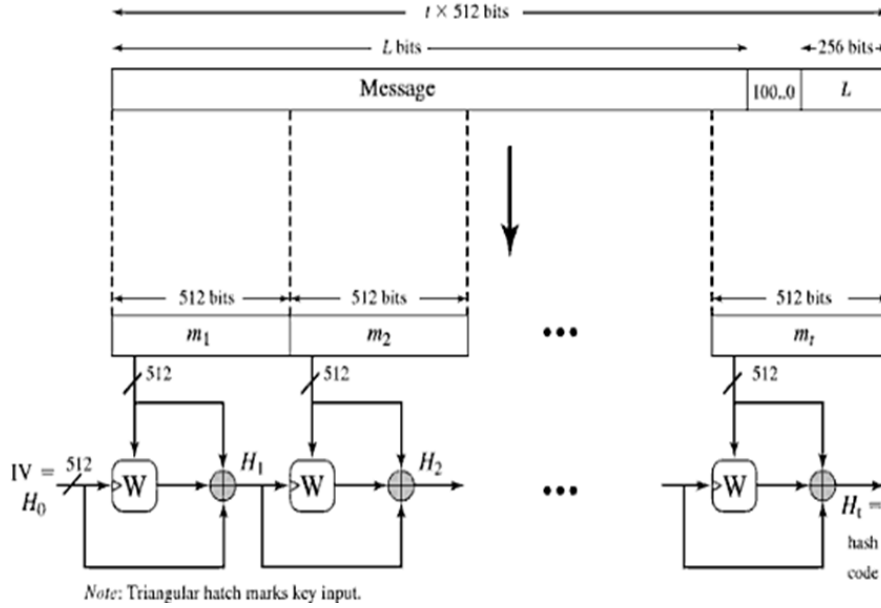


Figure 2: Merkle-Damgård Construction (Copied)

- The t is decided based on the size of the input message (M) and N . we can say that $t = \lceil \frac{|M|}{N} \rceil$. For example, let us say the message size is 1500 bits ($|M| = 1500$) then $t = \lceil \frac{|M|}{N} \rceil = \lceil \frac{1500}{512} \rceil = 3$. If we see, 3×512 then it is 1536 bits however our message is only 1500 bits. The remaining 36 bits are padded bits of 0's in the $t - 1^{th}$ block considering the blocks are indexed from 0 to $t - 1$.
- Every message block will be given to an internal function that does rotation, XOR, majority and others. In this material, we will not discuss those.
- The function takes two inputs: Along with the message block the Initialization Vector (IV) of size equal to the hash algorithm output size (if SHA 256 then output size is 256 bits, SHA 512 then output size is 512 bits) will also be given to the function for the very first message block hash computation. The output of the first block computation goes as the input for the next block computation. So, that output replaces the IV from the second block with the outputs of the previous block. In the figure 2, the IV size is given as 512 bits but consider as 256 bits for our discussion. The IV is predefined and should be known to both parties that is the prover and verifier.
- This process repeats for all the message blocks. In the 1500 bits example, the function is repeated three times because it has three blocks. The output that is coming out from the last block computation is the hash value for the input message.

This construction is vulnerable to the Length Extension Attack.

4.2 Length Extension Attack

In this attack, an attacker can use $Hash(A)$ and the length of message A to calculate $Hash(A||B)$ (B is an attacker-controlled message) without knowing A . Let us discuss this attack with an example:

- Consider that there is a message ("Transfer_1_Lakh_INR_to_me") sent by *Alice* to *Bob* along with the Hash value computed for it.
- Assume the message block size is 10, so the message blocks are as follows:
 - Message Block 1: "Transfer_1"
 - Message Block 2: "_Lakh_INR_"

– Message Block 3: "to_me"

- In the third block, we do not have 10 characters, so pad 0's to bring 10 characters: "_to_me00000".
- Now, if *Alice* compute the hash based on the Merkle-Damgård Construction for example using well known hash algorithm then the final block computation is the hash value for the input message "Transfer_1_Lakh_INR_to_me". Let us consider that the final hash value is "CDA123" [The output of less size is given only for easy understanding. In actual it will be of 256 bits, or 512 bits or 128 bits or more.]
- Assume that the message "Transfer_1_Lakh_INR_to_me" contains two parts: one is the secret (Transfer_1_Lakh_INR_) of 20 characters and another is open message (to_me) of 5 characters.
- Now, assume that the attacker got access to the hash "CDA123", open message (to_me) and the secret size which is 20. Attacker will use different techniques to get the size.
- Attacker can compute the new hash for the modified message as follows:
 - Knowing the secret size (20) and open message with its size (5), attacker can compute the number of padding characters required. As considered earlier, the algorithm is dividing message in to 10 characters block and message is of only 25 characters so 5 are the padding characters.
- Now, assume attacker (*Eve*) want to pad a message "and_Eve" in to the actual message. Also, *Eve* want that message should be accepted by *Bob*.
- *Eve* takes "CDA123" and give as the IV for the same hash function along with the new message "and_Eve000". Three 0 characters are padded since the block size is 10 that is known to *Eve*. The output produced is for example "DE672A". Now this hash is for the message "Transfer_1_Lakh_INR_to_me00000and_Eve000".
- After computing the hash value "DE672A", *Eve* forwards the new hash value and modified open message "to_me00000and_Eve" to *Bob*. The *Eve* may not required to send the three padding characters with the message, if he send "Transfer_1_Lakh_INR_to_me00000and_Eve" also sufficient because the algorithm will automatically inserts the padding characters while computing the hash as per the requirement.
- Assume that the *Bob* receives the secret message (Transfer_1_Lakh_INR_) from *Alice* without any modification.
- Now, *Bob* concatenates the secret message and the open message. The complete message is "Transfer_1_Lakh_INR_to_me00000and_Eve".
- *Bob* compute the hash for "Transfer_1_Lakh_INR_to_me00000and_Eve" and according to the algorithm, 3 character will be padded. So, the message is "Transfer_1_Lakh_INR_to_me00000and_Eve000". The computed hash value will be "DE672A"
- Now, *Bob* compares both the hash values that is received from *Alice* (actually from *Eve* who intercepted the *Alice* open message, and hash value and then send the modified message along with the hash computed for the modified message) and he has computed. In this case, both are "DE672A" so the modified message is accepted by *Bob*. Hence, the length extension attack is successful.

Any hash algorithm that uses the Merkle-Damgård Construction are vulnerable to the length extension attack however, the algorithms like SHA-384, SHA-512/256 and SHA 3 family are not vulnerable. Also, length extension attack is not possible if the implementation follows the asymmetric key based digital signature.

4.2.1 Attack Possibility

The length extension attack is possible when the Merkle-Damgård Construction is used in the hashing algorithm and Message Authentication Code (MAC) is the implementation. Let us discuss the cases, where the length-extension attack is possible and where it is not. The objective of the attacker is to intercept and modify the original message sent by the sender and make the receiver to accept it.

- Case 1 (Possible): Assume, the message is hashed using the Merkle-Damgård Construction based hashing algorithm and sent to the receiver but intercepted by the attacker. Now, attacker can do two things:
 - Since the attacker captured the plain message and the hash, he/she can modify the message and recompute the hash. Then forward the modified message and its equivalent hash to the receiver. The receiver will accept the message as valid. Without the length-extension attack, the objective of the attacker will be achieved.
 - Attacker apply the length extension attack on the intercepted message and hash value. He will append the additional message on the original message and compute the new hash using the intercepted hash and the appended message. The modified message and the updated hash will be sent to the receiver and it will be accepted. The process of computing the new hash from the appended message is already discussed.
- Case 2 (Possible): Let us consider that the Message Authentication Code (MAC) is used for the authenticity and integrity. If the Message Authentication Code (MAC) is used then also the length extension attack is possible. In MAC, we use the key based hashing (HMAC - Hash-Based Message Authentication Code) and the key is symmetric. Assume, two parties that is the sender and receiver knows the symmetric key k and that will be used for the hash computation. When the message will be sent, the hash is computed as follows:

$$Signature = h(k||message)$$

Now, the *Signature* along with the message is sent to the receiver but that is intercepted by the attacker. The attacker knows the message and *signature* but not the key. Now, the attacker can perform the length extension attack and the attack will be successful. Even though, key k is not known, attacker will append the additional message in the original message and compute the new hash (*Revised_Signature*) as follows. Assume size of the key k is known to the attacker.

$$Revised_Signature = h(Signature||appended_message)$$

$$Revised_message = message||appended_message$$

On receiving the *Revised_Signature* and *Revised_message*, the receiver computed the following and he/she knows the key k .

$$Signature' = h(k||Revised_message)$$

The received *Revised_Signature* and computed *Signature'* will match and the message (*Revised_message*) will get accepted. The attacker objective of length extension attack will be successful. We already discuss this process in the Length Extension Attack section 4.2.

- Case 3 (Not Possible): We know that the Digital Signature is used these days to ensure the message integrity (to ensure message is not modified) and message authenticity (to ensure it is coming from the right source). The MAC is also similar to the digital Signature but the digital signature uses the Asymmetric key cryptosystem. The hash produced from the message is encrypted using the private key and get decrypted using the public key. The attacker cannot see the plain hash since it is encrypted so cannot perform the length extension attack. In this case, even the hash algorithm used is based on the Merkle-Damgård Construction, the length extension attack is not possible.

Other solutions to avoid the length extension attack are: Wide-Pipe Construction and Sponge Construction.

4.3 Wide-Pipe Construction

To avoid the vulnerability of the length extension attack and other requirements, Stefan Lucks proposed the use of the Wide-Pipe Construction which is similar to the Merkle-Damgård Construction but internally uses the higher bits but outputs the truncated portion as the final hash value.

We have discussed in the Merkle-Damgård Construction that initially IV (size is equal to final output size) will be given as the input for the first message block hash computation and in every next message block hash computation, the output of the previous message block hash computation is used as IV. The outputs that are used is of same size starting from the IV to the final output hence possibility for the length extension attack.

In case of the Wide-Pipe Construction, the IV size is higher than the final output size. For example, let the final hash output size be 384 bits but the IV size and the internal computation output will be of 512 bits. This means, for every message block hash computation, the output size is 512 bits and when the final output comes that is 384 bits which is after the truncation of right-most 128 bits. We can see from figure 3, the input is of three message blocks with size 1024 bits considering the SHA 384 algorithm, IV value is 512 bits and the internal round or each message block hash computation output is 512 bits including the final message block hash. However, when the final output is produced to the user that is the truncation of the original output. Instead of 512 bits the final output is only 384 bits after the truncation of right most 128 bits. In this case, the length extension attack is not possible since the attacker will not be knowing the 128 bits that will be required to compute the hash of the next message block that is padded by the attacker. In figure 3, F is the hash function and TF is truncation function.

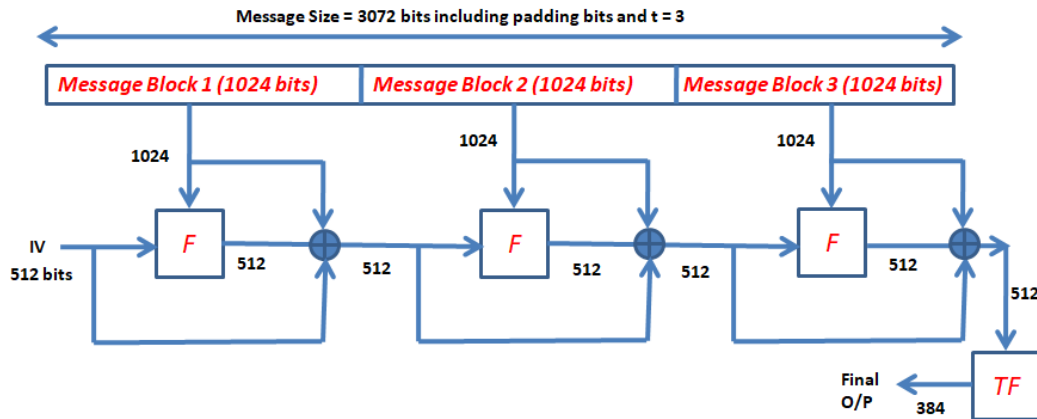


Figure 3: Wide-Pipe Construction

The SHA 384 and SHA-512/256 are not susceptible to the length extension attack since it truncates the final output. In case of SHA-512/256, the internal message block hash computation produces and uses (output of the current block is the input of the next block) the 512 bits however the final output is 256 bits.

Important Note: When the final hash size to be n then in the internal rounds it will use is $2n$ size. In case of SHA 384, the internal size is less than the $2n$ because final output hash value is 384 bits but the internal round (every message block hash computation) output is 512 bits only.

4.4 Sponge Construction

This construction is based on absorbing and squeezing concept. The messages are divided into blocks and it absorbs all blocks first and then squeeze to produce the output layer by layer or round by a round. The figure 4 shows the sponge construction. In the figure, P is in the input message block and Z is the hash output. According the size of the hash output we need, we can go multiple rounds and concatenate the Z_0, Z_1, \dots . For example, let us consider that we want the hash output size as 512 bits and every squeezing round produces 256 bits then the final output will be given by concatenating Z_0 and Z_1 ($Z_0 || Z_1$).

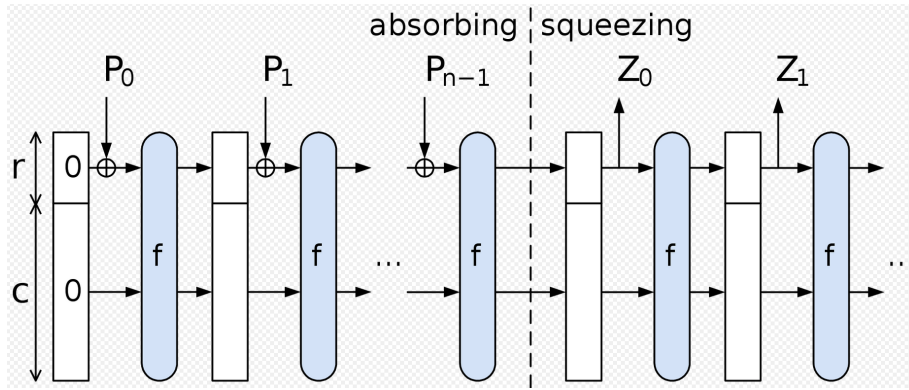


Figure 4: Sponge Construction (Copied)

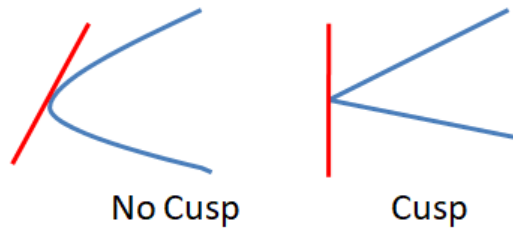


Figure 5: Cusp vs No-Cusp

In case, attacker pads the message to perform the length extension attack, it will not succeed because the padding will happen on the squeezing part which is after the input absorption. Hence, the sponge construction based hash algorithms are not susceptible for the length extension attack. The SHA-3 (from the Keccak (pronounced as Kekjack) family) is not susceptible for the length extension attack. Ethereum uses the SHA-3 hash functions.

5 Elliptic Curve

An elliptic curve (EC) is a plane algebraic curve defined by an equation of the form y^2 . The plane algebraic curve means it is the non-singular curve in two-dimensional space involving the co-ordinate (x, y) (we can say this as cartesian coordinates since we are defining in 2D plane). The term algebraic means the curve is defined by a polynomial equation with two variables for example x and y . The curve should be non-singular (means smooth curve in geometry) means there should not be any cusps or self-intersections.

- Self-Intersections: No point should occur more than once otherwise intersection occurs. Due to the intersection, the group law or operation (will be discussed soon) will break. The key operation in elliptic curves is drawing a line through two points on the curve, and finding where it intersects the curve again (third point) to see the sum of the two points means the addition of two points. In case, the point repeats then line might intersect the curve in fewer or ambiguous places.
- Cusp: There should not be any sharp point on the curve because we will not be in position to draw the tangent line following the curve (in case of cusp, where the curve sharply "comes to a point," and the tangent cannot be well-defined). A tangent line is a line that touches the curve only at that point and in case of elliptic curve it should go towards the curve side otherwise not acceptable. In case of cusp we cannot draw the line using the tangent point that goes along with the curve.

Even in cusp also, we can draw the tangent line, however, for a non-singular, the tangent line slope should follow the slope of curve. In that case tangent line does not behave like a tangent at a smooth point. To compute the doubling, tangent line following the curve is

required. The figure 5 shows the cusp and non-cusp. If we see in cusp, the line is not going towards the curve and there are many tangent lines possible.

An elliptic curve over a field K is a curve defined by an equation of the form

$$y^2 = x^3 + ax + b \quad (3)$$

where $a, b \in K$ and the discriminant $-16(4a^3 + 27b^2) \neq 0$

5.1 Discriminant

The discriminant of a polynomial is a number that can be easily computed from the coefficients of the polynomial and it will be zero if and only if the polynomial has multiple root. Let us consider the discriminant equation,

$$D = -16(4a^3 + 27b^2) \quad (4)$$

If we solve the equation, the output of D defines the following

- $D = 0$, if and only if $f(x)$ has at least two equal roots. Here, the curve intersects that means it is a singular curve so we should choose another a and b value and try again.
- $D < 0$ means one real roots and 2 complex conjugate roots. The complex roots always come in pairs called conjugate roots. If there is a complex root $2 + 3i$ then there will also be $2 - 3i$ and this is referred as complex conjugate roots.
- $D > 0$ means distinct real roots are there. In case of a cubic polynomial (polynomial of degree 3), it can produce three distinct points.

5.2 Weierstrass Equation

Weierstrass equation is the standard way to write the curve in a simple (avoids unnecessary coefficients or terms) and canonical form. A canonical form means a standard or “normal” way to write or represent something that is unique (or nearly unique mean different complex curves that are essentially the same can be transformed into this form.) and follows agreed-upon rules. We can see from the equation 5.

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a \quad (5)$$

After removing the characteristic 2, we get the equation 6

$$y_1^2 = x^3 + a'_2x^2 + a'_4x + a'_6 \quad (6)$$

After removing the characteristic 3, we get the equation 7. The reason behind going for the equation 7 is that it is simplified.

$$y^2 = x^3 + ax + b \quad (7)$$

How the equation 5 reduced to equation 7?. By removing certain linear terms (that is removing a_1, a_3 as well as x^2) we get the simplified equation. While using this simplified equation, we should not have the characteristic 2 and 3. The reason is given in the upcoming *slope* sub-section.

The characteristic means a smallest positive integer for example $n = 5$ such that $1+1+1+1+1 = 0$. If no such n exist then we can say the characteristic of the field is 0. Now, the question is how this produce 0 while adding one n number of times. Let us consider Field of integers modulo 5 (F_5) then $1 + 1 + 1 + 1 + 1 \equiv 0 \pmod{5}$ and characteristics of this is 5. We are taking the identity element that is 1 of the field, doing the additive operation n number of times and if we get 0 (additive identity) as the output then that is the characteristic otherwise 0 is the characteristic.

Order: There is also a term that is order. Here, we pick an element (g) from the group and perform the repeated group operation ($g^0, g^1, g^2 = g \cdot g, g^3 = g \cdot g \cdot g, \dots$) to get the identity. If we get the identity element of that group then the number of times the repetition occur is the order. In general, the order is in case of group and characteristic in case of field. In case of **characteristic**, we pick the identity element (that is 1) and perform the repetitive addition to get 0 however in **order** we pick any element and perform the repetitive group operation to get the identity element.

5.3 Elliptic Curve Construction

Given the equation 7, we choose the elements that is a , b and k . However, we should take the discriminant as the condition to finalize the value for a and b . The popular curves are

- secp256k1 (used by Bitcoin):
Defined over \mathbb{F}_p with $p = 2^{256} - 2^{32} - 977$
 $a = 0$, $b = 7$.
- NIST P-256 (secp256r1):
Defined over a 256-bit prime field \mathbb{F}_p . $a = -3$, b is a large constant defined by NIST.
 $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

The $r1$ refers to “random” or “recommended” curve means chosen randomly and The $k1$ refers to “Koblitz” curve. Koblitz curves has efficient arithmetic using Frobenius endomorphisms, allowing faster computations.

5.4 Discriminant Example

Let us see a simple example using the weierstrass simplified curve $y^2 = x^3 + ax + b$ over F_p considering $a = 2$, $b = 2$ and $p = 17$.

At first, let us compute discriminant and check whether this can produce the non-singular curve or not.

$$D = 16(4a^3 + 27b^2)$$

The cubic curve (cubic polynomial curve) discriminant $D = -4a^3 - 27b^2$ and Elliptic curve discriminant is $D = 16(4a^3 + 27b^2)$. Here the factor -16 arises from the relationship between the curve’s invariants and the polynomial discriminant (means the sign and constant ensure that the discriminant is compatible with other properties of elliptic curves).

Now let us compute the discriminant for the take values,

$$4(a = 2)^3 \text{ mod } 17 = 4 \times 8 \text{ mod } 17 = 32 \text{ mod } 17 = 15$$

$$27(b = 2)^2 \text{ mod } 17 = 27 \times 4 \text{ mod } 17 = 108 \text{ mod } 17 = 6$$

$$\text{Now add } 15 + 6 \text{ mod } 17 = 4$$

$$\text{Next, } -16 \text{ mod } 17 = 1$$

Finally $D = 1 \times 4 \text{ mod } 17 = 4$ and $D > 0$. Hence the parameters qualify to be non-singular.

5.5 Group Operation

The Elliptic Curve supports two group operation: Point Addition and the Point Doubling. Both these operations utilizes the slope equation to compute the results. The elliptic-curve points form an additive group because of that we cannot do exponentiation. For example, in the Diffie-Hellman, we use g^a , where g is the generator and a is the random number. The Diffie-Hellman work with the group operation over multiplication. However, in elliptic curve we cannot do such exponential operation because the group operation in elliptic curve is addition.

5.5.1 Slope

When we draw the line on the elliptic curve means draw the line touching the elliptic curve points then the slope is important.

For Addition Consider that given two distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ of the elliptic curve. The slope m of the line through these points is computed as in equation 8.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{8}$$

Now we can compute the third point $R = P + Q = (x_3, y_3)$ as follows.

$$x_3 = m^2 - x_1 - x_2$$
$$y_3 = m(x_1 - x_3) - y_1$$

For Doubling In case of doubling, we will be knowing only one point that is P and this has to be doubled. We can say as $2[P]$ means double this point. In this case, finding the slope to draw the line is very important because if we take one point then we can draw line on any direction however the line should go along with the side of the curve. Recall that there is a property of the curve that the curve should not have cusp otherwise tangent line cannot go towards the slope side and cannot perform point doubling. Since, the curve will not have the cusp, we can find the slope with the equation 9. The equation 9 shows that from the given point $P = (x_1, y_1)$, the slope m is computed.

$$m = \frac{3x_1^2 + a}{2y_1} \quad (9)$$

Now the question is, how this equation is computed or derived. Let us take the curve equation $y^2 = x^3 + ax + b$ and do the differentiation. We consider the differentiation with respect to x

$\frac{d}{dx}$ is the derivative of a function with respect to x

$\frac{dy}{dx}$ is the derivative of y with respect to x .

Left-hand side: $\frac{d}{dx}(y^2) = 2y \cdot \frac{dy}{dx}$

Even though we are differentiating with respect to x the left-hand side y^2 is differentiated because y depends on x means we get y from x only so differentiation is applied on y .

The reason for having $\frac{dy}{dx}$ is, y is not a constant and it is a function of x , So when differentiating y^2 it is actually $(y(x))^2$. Hence, we differentiate y^2 , knowing that y changes with respect to x . It is based on chain rule that is like a function inside a function.

Right-hand side: $\frac{d}{dx}(x^3 + ax + b) = 3x^2 + a$

So: $2y \cdot \frac{dy}{dx} = 3x^2 + a$

Now
 $\frac{dy}{dx} = \frac{3x^2 + a}{2y}$

Important Note: When we present the simplified Weierstrass equation $y^2 = x^3 + ax + b$, it was discussed that characteristic 2 and 3 should not be there that means p value should not be 2 and 3. The reason is, if we use $\text{mod } 2$ or $\text{mod } 3$ for the slope equation, we end up with 0 either in the numerator or denominator and we get no useful integer output that is either we get infinity or 0. To understand better, let us see the equation as follows:

$\frac{3x^2+a}{2y} \text{ mod } 2 = \frac{3x^2+a}{0}$. The denominator is 0 because irrespective of y value it is multiple of 2 and when we compute $\text{mod } 2$ then it will be 0 because it is divisible by 2 since it get multiplied by 2. Likewise for 3 in the following.

$$\frac{3x^2+a}{2y} \text{ mod } 2 = \frac{0+a}{2y}$$

After computing the slope value, we have to compute the final doubling point as follows.

$$\begin{aligned} x_3 &= m^2 - 2x_1 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

In the points, it is written as x_3 and y_3 not as x_2 and y_2 because we consider that two points are given as the input for doubling function however the two points P and Q are equal ($P = Q$).

5.5.2 Point Addition

This is the addition of two given points. Let us consider that we choose two points that is P_1 and P_2 from the curve. We do the point addition on this to get the third point. Figure 6 shows that

$$P_1 + P_2 = P_3$$

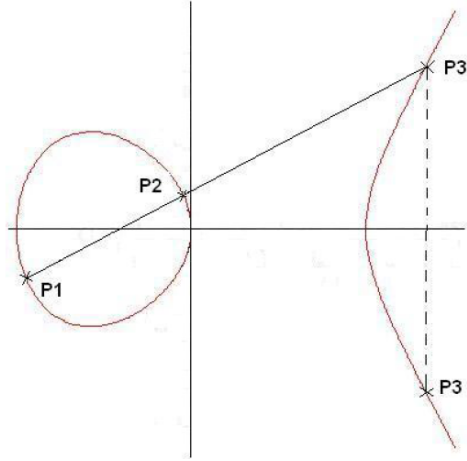


Figure 6: Point Addition [Image Copied]

the given points P_1 and P_2 are added and produce the third point that is P_3' . The line is drawn considering the points P_1 and P_2 so slope of the line is very clear that it has to touch the two given points and the next point it touch is the addition point. The elliptic curve equation that is the weierstrass equation is 3-degree polynomial so the line will cross in three points of the curve.

If we see, P_1 is (x_1, y_1) and P_2 is (x_2, y_2) then the output of addition is P_3' is (x_3, y_3) however actual result is P_3 is $(x_3, -y_3)$ since we do mirroring (The details are discussed in mirroring section). It is clear from the figure that the third point is where the line touches the curve however how to compute that point. Hence, we need to compute the slope eventhough the slope is very well fixed in addition. Here the slope is very well fixed means the line has to be drawn in that direction only if we change the line direction then it will not touch the points P_1 and P_2 , which are the inputs.

We can compute P_3 is (x_3, y_3) with the help of slope equation 8 and the following.

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

5.5.3 Point Doubling

In case of point doubling, we will be having only one point $P = (x_1, y_1)$ and we have to compute $2[P]$. We see from figure 7 that a given point P , the doubling $2P$ is computed and the result is R . We can compute this by having the slope equation 9 and the following:

$$\begin{aligned} x_3 &= m^2 - 2x_1 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

The doubling is represented as $2P$ and $2[P]$ in the above but in cryptography $2P$ is used as standard.

We will see how to compute $5P$. Here we have to use both point addition and doubling. Let us see this in steps:

- Step 1: Convert the 5 to binary 101
- Step 2: Initialize the Ouput R as point of infinity.
- Step 3: For all bits from the Most Significant Bit (MSB), we do the following process.

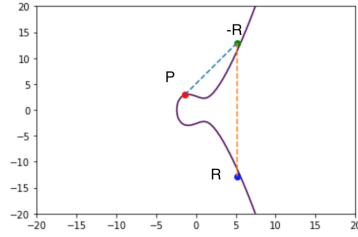


Figure 7: Point Doubling [Image Copied]

- Step 3.1: For the first bit 1, we say $R = P$. Always the MSB will be 1 because we cannot write 5 as 0101. Here, we consider that we will write all numbers exactly with the number of bits required means for 5 it will be 101, 9 will be 1001, 2 will be 10 and so on.
- Step 3.2: Now, next bit, if bit is 0 (in our example case second bit is 0 only) then do the doubling which means $R = 2P$ or we can write as $R = 2R$ [This means, if bit is 1 then do the doubling]
- or
- Step 3.2: Now, next bit, if bit is 1 (in our example case it is 1 when we reach to the third bit from MSB) then do the doubling which means $R = 2R$ actually in our example case it will be $R = 4P$ because we already done one doubling when the bit was 0 and again we done once. In addition to the doubling, we should do point addition that is $R = R + P$ means $R = 5P$ [This means, if bit is 1 then perform doubling and addition.]

Important Note: The step 3.2 will be in iteration until all bits will be completed.

5.5.4 Point of Infinity

In Elliptic Curve (EC), the point of infinity plays the major role. We know that if we draw a line which is not vertical on the curve then it will touch three points on the curve. However, when we draw the vertical line then it will touch only in two point and we consider that at some third point it will touch the curve again however that is imaginary so that is called as the point of infinity (\mathcal{O}). Even, in case of point doubling we can see from the figure 7 that the curve touches only two points and third will be the point of infinity. An important point that we cannot have any (x, y) co-ordinate for the point of infinity however can be represented as $(0 : 1 : 0)$ in 3D projective co-ordinates. The importance of the point of infinity is as follows.

- It is used as the identity element in the Elliptic Curve. If we add $P + \mathcal{O}$ then the output is P .
- If we compute the inverse of any point then we get the point of infinity that is $P + (-P) = \mathcal{O}$. This means, let us say we are drawing the vertical line that touches the two points that is P and $-P$ then addition of these two will be third point that touches the curve and we know that no such point exists hence point of infinity \mathcal{O} . Here P is (x, y) and $-P$ means $(x, -y)$. In elliptic curve, all point will have its mirror that means (x, y) will have $(x, -y)$ as mirror or reflection similarly $(x, -y)$ will have (x, y) because elliptic curve is symmetric in nature. It will be more clear from the example that will be discussed in next section.

5.6 Mirroring

It is shown in figures 6 and 7 that the final result is not the actual point, we take the mirror (reflection over the x-axis). This means, we add two points P_1 and P_2 and it produce P'_3 however we take the output as P_3 by mirroring. This means,

$$P_1 + P_2 = P'_3 \Rightarrow P_3$$

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3) \Rightarrow (x_3, -y_3)$$

The reason for taking the mirroring is as follows:

- Mirroring with respect to inverse: When we add the point with its inverse then we get the identity element. In the case of Elliptic curve, P and its inverse $-P$ getting added will produce the identity element that is \mathcal{O}
- Mirroring with respect to Point Addition: Let us say we have P, Q and the addition gives R then the reflection or mirroring is $-R$. We want to ensure that $P + Q + R = \mathcal{O}$ and if we see by moving the R on other side then $P + Q = -R$. So, the $P + Q$ output is not R , it is $-R$.
- Mirroring with respect to Doubling: Now consider a case where we do not mirror. First we calculate $2P$ by drawing a line tangent to the curve at point P , and call this Q . To calculate $3P$, we add $P + Q$, so we draw a new line that goes through P and Q which we be the exact same line, so the "third point" on the curve is P itself. However, if it is mirrored and added then the point will be different.

Mirroring helps to ensure the properties (inverse, identity and possibly associativity) or the law of the group.

5.7 Elliptic Curve Example

We consider the weierstrass equation $y^2 = x^3 + ax + b \pmod{p}$, with $a = 2, b = 2$ and $p = 17$. Hence, the equation is $y^2 = x^3 + 2x + 2 \pmod{17}$. Now, we are going to compute the curve points of finite field F_{17} .

- **Discriminant Check Example** First, let us check whether the value applied equation holds the elliptic curve property of non-singular by validating the discriminant.

$$-16(4a^3 + 27b^2) \neq 0$$

Now, can be rewritten as

$$-16(4(2^3) + 27(2^2)) \pmod{17}$$

$$-16(4(8) + 27(4)) \pmod{17}$$

$$-16(32 + 108) \pmod{17}$$

$$-16(140) \pmod{17}$$

$$-2240 \pmod{17}$$

Compute this as follows: $2240 \pmod{17} = 13$ then to compute the $-2240 \pmod{17}$, $-13 + 17 \pmod{17} = 4 \pmod{17} = 4$

$$= 4 \neq 0$$

Hence, qualifies as the elliptic curve.

- **Computer Group Points:** Now let us compute the points.
 - Construct the table 1 using the parameter values and Elliptic curve equation. We take x and y and compute $y^2 = x^3 + 2x + 2 \pmod{17}$, and $y^2 = y \times y \pmod{17}$. If we see in table 1, the row of 17 and 18 are highlighted because those are not required since it is finite field F_{17} and value will repeat after 16 since it is modulo 17. If we go more than 16 then we can see the output repeats, in case of 17 it is of 0 and in case of 1 it is of 1. Hence, unnecessary so will drop those. **Another important note:** this field is not multiplicative (group F_{17}^\times) hence we consider 0 in points computation. You consider like this, we computed the multiplicative group points from the finite field F_{17} . But if we see the output points that includes 0 however, it is not (0,0), it is (0, 6) or (0,11). If it is (0,0) then it will be cusp cannot accept this as elliptic curve.
 - Now select the points based on condition: We take x and see its y^2 output in second column, then see the equivalent value in the fourth column, if any match then take the equivalent y from third column and now those (first and third column) forms the point. For example, let us take the value $x = 0$ and its equivalent $y^2 = 2$ in second column and we see the fourth column where 2 is present (it is present in seventh row that is for

x	$y^2 = x^3 + 2x + 2 \pmod{17}$	y (candidate)	$y^2 = y \times y \pmod{17}$
0	2	0	0
1	5	1	1
2	14	2	4
3	1	3	9
4	6	4	16
5	1	5	8
6	9	6	2
7	2	7	15
8	3	8	13
9	1	9	13
10	2	10	15
11	12	11	2
12	3	12	8
13	15	13	16
14	3	14	9
15	7	15	4
16	16	16	1
17	2	17	0
18	5	18	1

Table 1: Elliptic curve values for $y^2 = x^3 + 2x + 2 \pmod{17}$

$y = 6$) and the equivalent y value in the third column should be taken that is 6. So, one point is $(0,2)$ and also we can see there is a match with 11 means that also has 2 in the fourth column so next point is $(0,11)$ that is the reflection or mirror point. All point will have its additive inverse means if we take any x value then that will have two y values to satisfy (x, y) and $(x, -y)$. In general or geometry y and simply with minus symbol of y however it is not in case of our example because simply having minus will not give the inverse in modulo.

- Important point: We are not deriving the y value from y^2 that is in the second column. We take the third column y value and then compute y^2 because if we take square root of the value in second column to find y then there is a possibility of decimal (irrational) numbers (It will not be good for the finite field arithmetic) and precision issue will arise when we apply in digital system (if we use the float value then there will be a precision error while storing and retrieving it in the digital system because the system can support up to a certain limit of decimal digit appropriately afterwards no proper precision possible). Hence, we compute y^2 from given y .

- Now, let us list all the valid points

$(0, 6), (0, 11)$
 $(3, 1), (3, 16)$
 $(5, 1), (5, 16)$
 $(6, 3), (6, 14)$
 $(7, 6), (7, 11)$
 $(9,1), (9,16)$
 $(10,6), (10,11)$
 $(13,7), (13,10)$
 $(16,4), (16,13)$

There are many points that are not part of the curve. For example, we are not taking $x = 14$, because it produce $y^2 = 3$ in second column and that is not in fourth column so we ignore. Now total number of points we have is 18 and we have to add one for point of infinity \mathcal{O} . Hence the number of points or order of the curve or cardinality of the curve is $18 + 1 = 19$.

Important Note: The number of points is 19 (order of the curve) and more it is than the p value that is 17 (F_{17}). The number of points can be more than p or less than p , it

Points on Elliptic Curve over \mathbf{F}_{17}

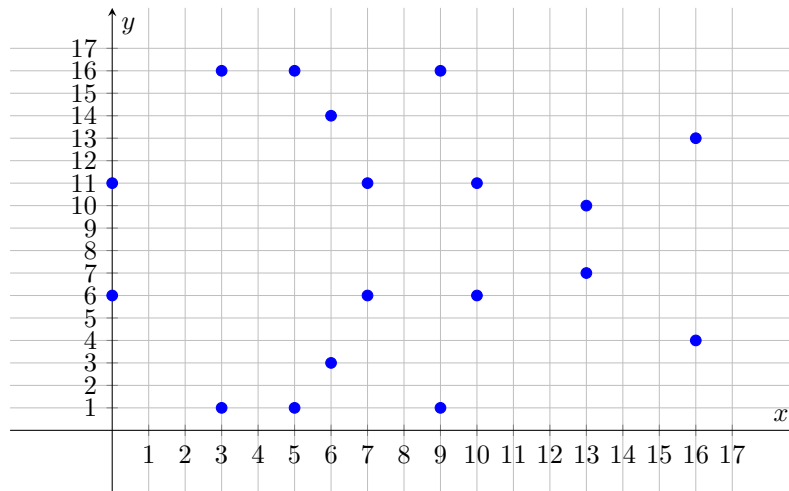


Figure 8: Elliptic Curve Plot

is based on the Hasse's theorem. We can see it from the following however, we do not need to go in very deep on this theorem.

$$|\#E(\mathbf{F}_p) - (p + 1)| \leq 2\sqrt{p}$$

This implies that the number of points lies within the interval:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbf{F}_p) \leq p + 1 + 2\sqrt{p}$$

- Now let us plot the graph using computed points. The figure 8 shows the elliptic curve for the equation $y^2 = x^3 + 2x + 2 \pmod{17}$ based on the computed points.
- **Point Addition:** Let us do a point addition by taking two points. For example, let us take the points $P = (x_1, y_1) = (5, 1)$ and $Q = (x_2, y_2) = (6, 3)$. First, let us compute the slope. All equations will go with the modulus p because of finite field.

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \pmod{p} \\ m &= \frac{3 - 1}{6 - 5} \pmod{17} \\ m &= \frac{2}{1} \pmod{17} \\ m &= 2 \pmod{17} \\ m &= 2 \end{aligned}$$

Now let us compute the third point $R = (x_3, y_3)$.

$$x_3 = m^2 - x_1 - x_2 \text{ mod } p$$

$$y_3 = m(x_1 - x_3) - y_1 \text{ mod } p$$

replace the value for x_3

$$x_3 = 2^2 - 5 - 6 \text{ mod } 17$$

$$x_3 = 4 - 5 - 6 \text{ mod } 17$$

$$x_3 = 7 \text{ mod } 17$$

$$x_3 = 10$$

replace the value for computing y_3

$$y_3 = m(x_1 - x_3) - y_1 \text{ mod } p$$

$$y_3 = 2(5 - 10) - 1 \text{ mod } 17$$

$$y_3 = 2(-5) - 1 \text{ mod } 17$$

$$y_3 = -10 - 1 \text{ mod } 17$$

$$y_3 = -9 \text{ mod } 17$$

$$y_3 = 6$$

Now the addition is $R = (x_3, y_3) = (10, 6)$, however as per the rule we have to take the mirror so the other point which has x as 10 is $(10, 11)$ have to be chosen. But it is important to remember that we are computing the $R = (x_3, y_3)$ using the slope equation and other two equations respectively for x_3 and y_3 . This already ensures the inverse or mirror or reflection. So we can say that $R' = (x_3, y_3) = (10, 6)$ and $R = (10, 11)$ whose mirror is already taken while computing the equation. Finally the $P + Q$ after mirror is $(10, 6)$ which is point on the curve (means on the group) only otherwise not valid.

- **Point Doubling:** Now let us see an example for point doubling that is $2P$. First let us take a point $P = (10, 11)$ from the group and compute the slope.

$$m = \frac{3x_1^2 + a}{2y_1} \text{ mod } p$$

Replace with value

$$m = \frac{3(10^2) + 2}{2(11)} \text{ mod } 17$$

$$m = \frac{3(100) + 2}{22} \text{ mod } 17$$

$$m = \frac{302}{22} \text{ mod } 17$$

Apply the modulus in both numerator and denominator

$$m = \frac{302 \text{ mod } 17}{22 \text{ mod } 17}$$

$$m = \frac{13}{5}$$

We bring 5 to the numerator

$$m = 13 \times 5^{-1} \text{ mod } 17$$

Now, let us compute the 5^{-1} .

$$5 \times 7 \equiv \text{ mod } 17$$

So 7 is inverse. We can use Extended Euclidean Algorithm to compute inverse efficiently.

$$m = 13 \times 7 \text{ mod } 17$$

$$m = 91 \text{ mod } 17$$

$$m = 6$$

Now, let us compute the coordinates, using the predefined equation that we have seen in point doubling discussion earlier. We consider here to refer that two points that is $P = (x_1, y_1)$ and $Q = (x_1, y_1)$ where $P = Q$ so it is only one point. However just to show difference we will be computing R as (x_3, y_3) not as (x_2, y_2) ,

$$\begin{aligned}
 x_3 &= m^2 - 2x_1 \text{ mod } p \\
 y_3 &= m(x_1 - x_3) - y_1 \text{ mod } p \\
 &\text{replace the value for } x_3 \\
 x_3 &= 6^2 - 2(10) \text{ mod } 17 \\
 x_3 &= 36 - 20 \text{ mod } 17 \\
 x_3 &= 16 \text{ mod } 17 \\
 x_3 &= 16 \\
 &\text{replace the value for } y_3 \\
 y_3 &= 6(10 - 16) - 11 \text{ mod } 17 \\
 y_3 &= 6(-6) - 11 \text{ mod } 17 \\
 y_3 &= -36 - 11 \text{ mod } 17 \\
 y_3 &= -47 \text{ mod } 17 \\
 &= 47 \text{ mod } 17 \\
 &= 13 \text{ mod } 17 \\
 y_3 &= -13 \text{ mod } 17 = 17 - 13 \text{ mod } 17 \\
 y_3 &= 4
 \end{aligned}$$

Now the doubling is $R = (x_3, y_3) = (16, 4)$, however as per the rule we have to take the mirror so the other point which has x as 16 is $(16, 13)$ has to be chosen. But it is important to remember that we are computing the $R = (x_3, y_3)$ using the slope equation and other two equations respectively for x_3 and y_3 . This already ensures the inverse or mirror or reflection. So we can say that $R' = (x_3, y_3) = (16, 4)$ and $R = (16, 13)$ whose mirror is already taken while computing the equation. Finally $2P$ after mirror is $(16, 4)$ which is point on the curve (means on the group) only otherwise not valid.

Important Note: There is mandate of mirroring to support the group properties. In the above examples, we need not to take the mirroring because the equation $y_3 = m(x_1 - x_3) - y_1$ is already mirrored like it is $-y_3$. In case, we want the non-mirrored point that is y_3 then we take the negative that is $-(m(x_1 - x_3) - y_1)$. For example $m(x_1 - x_3) - y_1 \text{ mod } 17 = 4$, We take $-4 \text{ mod } 17$, then will produce the output 13 and that point $(x_3 = 16, y_3 = 13)$ is the non-mirrored output.

5.8 Why EC for Cryptography

We have public key cryptographic algorithms such as RSA, Diffie-Hellman, and others. Let us see what Elliptic Curve Cryptography (ECC) achieves. We will start with hardness problems. Consider the group Z_q and variables $a, b, c \in Z_p$, where p is the prime order. Also, we consider the generator g that belongs to Z_p .

- Discrete Log Problem (DLP): Given g and g^x , it is hard to find the x that is randomly chosen long integer and belongs to Z_p . In Diffie-Hellman algorithm, this x is considered as the private key.
- Computational Diffie-Hellman (CDH) problem: Given g , g^x and g^y , it is hard to compute g^{xy} .
- Decisional Diffie-Hellman (DDH) problem: Given g^x , g^y and g^z , deciding $c = xy$ is very hard.

When we go with Diffie-Hellman (DH) key exchange that use the group over finite field F_p^* , the hardness lies on the DLP however there are sub exponential algorithm exist to solve in sub-exponential time. Therefore we need to use the large prime. There are other options like extension field however that suffers with the existence of sub-exponential discrete log problem solving algorithm or inefficient group operation means it takes time to do the group operation.

Hence, we go for the Elliptic Curve and the best known discrete log solving algorithm is exponential means the existing algorithm like Pollard's Rho algorithm can solve in $O(p^{1/2})$ or \sqrt{p} ; where p is the order of the group and also the group operations in elliptic curve are efficient. We can see the security comparison in the following table 2. The size in table 2 is the size of the prime order p of F_p and in case of symmetric key cryptography it is the key size. The table 2 shows that if we want to achieve the security equal to symmetric key cryptography size of 128 then in case of Diffie-Hellman (F_p) we need 3092 bits and in case of Elliptic-Curve Cryptography ($E(F_p)$) need 256 bits.

Security Comparison	Symmetric	F_p	$E(F_p)$
	128 bits	3092 bits	256 bits

Table 2: Security level comparison between symmetric and asymmetric cryptography

6 Digital Signature

Digital Signature is to verify source authenticity and integrity of the message that is sent by the sender to the receiver. Through this, receiver can ensure that the message is sent by the respective user and it is not tampered. Digital Signature uses asymmetric key cryptography and hash function for signature creation and verification. The process of generating signature is shown in equation 10. It uses the private key ($Priv$) to encrypt the hashed message $h(M)$, where M is the actual message and h is the hash function. After signature, sender sends M and Sig to the receiver. It is important to note that for signature, private key of the sender should be used because only the owner of the key can sign the message not any one else. Public key cannot be used for signature because it is shared with all (In case symmetric key sharing public key used for encryption and private key for decryption). This means, when we use the asymmetric key cryptography, if we need confidentiality then the public key for encryption and private key for decryption. If we need message authenticity then private key for encryption and public key for decryption.

$$Sig = E_{K_{Priv}}(h(M)) \quad (10)$$

Receiver can verify signature of a message using the public key (already shared with the verifier) of the sender. The process of verifying signature is shown in equation 11. Receiver computes the hash for the message (M) he/she received, decrypt (D) the Sig using the sender's public key K_{Pub} and compare the decrypted hash and computed hash. Message will be accepted if the hash matches otherwise it will be rejected. If accepted means the message from the authentic source and message is not tampered.

$$Decision = \begin{cases} Valid & \text{If } h(M) == D_{K_{Pub}}(Sig) \\ Invalid & \text{otherwise} \end{cases} \quad (11)$$

7 ECDSA

The ECDSA is the Elliptic Curve Digital Signature Algorithm used in various applications. This is used in the cryptocurrency implementation such as Bitcoin, Ethereum, etc. This algorithm verifies the integrity of the data that is shared with the receiver. We will discuss the parameter construction, proof generation, verification, correctness, public key derivation and secret key derivation if reused in this section.

7.1 Parameter Computation

Let us assume, Alice (A) wants to send a signed message to Bob. The parameters that are required are as follows:

- Initially, they must agree on the curve parameters $(Curve, G, N)$.
 - The publicly know curve is $E(F_p) = y^2 = x^3 + ax + b \text{ mod } p$; Here, $E(F_p)$, refers the elliptic curve defined over finite field F_p . The value of $a, b \in F_p$ and p should be large prime, however, should ensure the properties of elliptic curve means without cusp and intersection. We can use the discriminant to test it. Refer the Discriminant Example section for this.
 - $\#E(F_p)$ denotes the cardinality (number of points) of the elliptic curve E defined over the finite field F_p . We cannot say that the number of points on the group will be equal to p that we have seen the Elliptic Curve example.
 - N is the order of the sub-Group.
 - Pick a point G from the base group that is generated from the curve equation $E(F_p)$. We will pick a random point G to form the cyclic group which is the subgroup. Even though, it can be randomly chosen but need to ensure that order is a large prime N means $N \cdot G = \mathcal{O}$. We use the G as the base point and choose a large prime $N(N \leq \#E(F_p))$ and perform $N \cdot G$ equal to $N \times G$ means N scalar multiplication of G that should produce the point of infinity \mathcal{O} . Here, it produce point of infinity \mathcal{O} means, let us say the operation will produce the outputs with different points and at one stage it will repeat the produced output point and that stage will be considered as point of infinity.
 - Note: Base group is generated from the $E(F_p)$ and we generate the cyclic subgroup by picking a point from the $E(F_p)$ and doing the scalar multiplication means point addition/doubling with N and produce it. We have seen already the process of $N \cdot G$ in doubling. It may be possible that the subgroup will be equal to base group if number of points on both are equal (that is $N = \#E(F_p)$).
- Choose the Private Key d_A which is between $[1, N - 1]$ that is the scalar value (means an integer like 47) and a public key curve point $Q_A = d_A \times G$, which is the point co-ordinate in the form (x, y) . Here multiple means scalar multiplication that is doubling or adding the points as per the example we have earlier seen.
- Hashing Algorithm: We can choose any cryptographic hashing algorithm that can take any size of input and produce the fixed size output. In the equation 7.1, the hash function f takes any size of input (mentioned with $*$ means any size) and produce the output in fixed size that is n . It will always produce the output with same size and it is deterministic (will produce the same output for the same input whatever number of times you run) and random (output cannot be predicted and no shortcut method exist to predict the output). Here, we have given 0 and 1 since the final message or input will be in binary form once it will be processed at the digital system (computer). There are hash algorithms like MD5, SHA-256, SHA-512 and others exist for usage however MD5 is considered as insecure due to possibility of quick collision in the outputs.

$$f\{0, 1\}^* \rightarrow \{0, 1\}^n$$

- Message (m): Take the input m and compute the hash for example SHA-256 and it will produce the output of 256 bits. In that, only the partial bits will be chosen for signature that is only L_N leftmost bits bits we can say that L_N is the bit length of subgroup order N (for example, if $N = 18$ then the bit length is 5 because 18 is represented in binary as 10010).

7.2 Generation

The Algorithm 1 shows the process of signature generation for the message m .

Algorithm 1 Generation

- 1: Calculate $e = \text{HASH}(m)$ where HASH is a cryptographic hash function, such as SHA-2.
 - 2: Let z be the L_N leftmost bits of e , where L_N is the bit length of the group order N .
 - 3: Select a cryptographically secure random integer k from $[1, N - 1]$
 - 4: Calculate the curve point $(x_1, y_1) = k \times G$.
 - 5: Calculate $r = x_1 \bmod N$. If $r = 0$, go back to step 3.
 - 6: Calculate $s = k^{-1}(z + rd_A) \bmod N$. If $s = 0$, go back to step 3.
 - 7: The signature is the pair (r, s) .
-

The algorithm 1 shows that we have to choose a secret parameter k from $[1, N - 1]$, however it should not be 1 because G is public and we choose $k = 1$ then $k \times G = 1 \times G = G$ so the secret will be easily known as 1. Hence, it should be between 2 and $N - 1$. Here, why it should be $N - 1$ not N ? because if we take N and do $N \times G$, we get the point of infinity \mathcal{O} so $N - 1$. Why not more than N because it is our cyclic subgroup size. The algorithm produces the output that signature pair (r, s) .

7.3 Verification

The algorithm 2 shows the process of signature verification. Let us say, Alice generated the algorithm and sent the signature (r, s) and message (m) to the receiver Bob. The Bob also knows the curve parameters and the public key Q_A . The verification is done using the algorithm 2.

Algorithm 2 Verification

- 1: Verify that r and s are integers in $[1, N-1]$. If not, the signature is invalid.
 - 2: Calculate $e = \text{HASH}(m)$ where HASH is a cryptographic hash function, such as SHA-2.
 - 3: Let z be the L_N leftmost bits of e .
 - 4: Calculate $w = s^{-1} \bmod N$.
 - 5: Calculate $u_1 = zw \bmod N$ and $u_2 = rw \bmod N$.
 - 6: Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$
 - 7: If $(x_1, y_1) = \mathcal{O}$ then the signature is invalid.
 - 8: The signature is valid $r \equiv x_1 \pmod{n}$, invalid otherwise.
-

7.4 Proof of Correctness

Let us see how the ECDSA ensures the integrity verification based on the mathematical proof. We know the signature given is r and s along with that receiver get the message m . Also, the parameters (except d_A and k) that are considered as public (curve, N , Q_A) are known to the receiving party who will be doing the verification. We know from step 6 of algorithm 2 that is (x_1, y_1) is to verify the signature and that is derived from $u_1 \times G + u_2 \times Q_A$. Hence, we start with this.

$$\begin{aligned} &= u_1 \times G + u_2 \times Q_A \\ &= u_1 \times G + u_2 \times d_A G \end{aligned}$$

We know that Q_A is from $d_A G$ that is public key is derived from private key and the base point of the subgroup. Also make G common in the following.

$$\begin{aligned} &= (u_1 + u_2 d_A) \times G \\ &= (zs^{-1} + rd_A s^{-1}) \times G \end{aligned}$$

Replace u_1 and u_2 with zs^{-1} and rs^{-1} respectively where w is s^{-1} . Also, make s^{-1} common in the following.

$$= (z + rd_A)s^{-1} \times G \text{ Replace } s \text{ with } k^{-1}(z + rd_A) .$$

$$= (z + rd_A)(z + rd_A)^{-1}k^{-1^{-1}} \times G$$

Now $(z + rd_A)$ get canceled due to inverse and $k^{-1^{-1}}$ become k .

$$= k \times G$$

$$= (x_1, y_1) \text{ as per algorithm 1}$$

$r \equiv x_1 \pmod N$ as per algorithm 2 and it matches if no modification so satisfies integrity

Important Note: The $\pmod N$ is given only in $r \equiv x_1 \pmod N$ that is in the last step only not in all the steps. However, it is applicable all the steps. Since, this based on the Finite Field and Group, always the modulus applied in all operations. It is not given in the above for simplified presentation otherwise $\pmod N$ should be there in all steps.

7.5 Public Key Recovery

In ECDSA, we have the provision to recover the public key from the signature that is (r, s) pair. This is leveraged in cryptocurrency to derive the public key from the signature instead of pre-sharing the public key with all participants or including it in the transaction. If we include in the transaction then the size of the transaction will be larger. Now, let us see, how to recover.

- We know that the public key Q_A is from $Q = d_A \times G$. However, now Q_A is not known to the receiver.
- We will assume that R is the temporary public key which is $R = (x_1, y_1) = k \times G$, However, R is not known.
- As we have seen earlier, if we take any x co-ordinate then there will be two y coordinate possible that is y and $-y$. Hence there will be two points: (x, y) and its mirror or the reflection $(x, -y)$.
- In the algorithm 1, the value of r is $x_1 \pmod N$. We can write this as $x_1 = r + N * j \pmod N$ because in $r = x_1 \pmod N$, r is the remainder, N is the divisor and let us say j is the quotient. We know r and N however j is not known so we have to try from $0, 1, 2, \dots$. But, in most of the cases, j with 0 or 1 will satisfy the equation when we try to get the x_1 .
- Now knowing the x_1 value, we can try to find the y value, by applying to the curve equation $y^2 = x^3 + ax + b \pmod p$. We know x , apply to the curve equation then we get y^2 then take the square root so we get y value. In this case, we get $R = (x, y)$ or $R = (x_1, y_1)$ and we can get the mirror or reflection point that is $R' = (x, -y)$. Here, we say from y we can derive $-y$ just by adding minus symbol like $y = 10$ and $-y = -10$. However, this will not be there in the modular operation.
- To compute $-y$ from y , let us say $p = 29$ and $R = (10, 22)$. We compute $-y$ as $(10, -22 \pmod{29}) = (10, 29 - 22) = (10, 7)$, so now $R = (10, 22)$ and $R' = (10, 7)$ because $-22 \pmod{29} = 7$.
- Now we know the temporary public key R and R' generated in algorithm 1. In algorithm 1, only R is focused not R' .

Important Note: We try with $j = 0$ as well as $j = 1$ because it may be possible we will not get the required R with $j = 0$ so need to try $j = 1$ or $j = 2$ in some cases.

Now, knowing R and R' , now let us compute the public key Q_A .

- We know R and R' .
- We have to compute r^{-1} , which is the multiplicative inverse of the value r from the signature (modulo the order of the generator of the curve). We know that modular inverse can be

computed using the Extended Euclidean Algorithm based on $r \times b \equiv 1 \pmod N$, where b is the inverse. We have already seen how to compute the inverse in point doubling example (specifically in slope value computation).

- Then, we compute z which is the leftmost L_N bits of the hash of the message (where L_N is the bit length of subgroup order N). The message m is known because the signature comes with r, s and m .
- There are two points $r^{-1}(sR - zG)$ and $r^{-1}(sR' - zG)$ possible however one is the public key. We know all values now and let us prove in the following why it works? Consider here d as d_A , which is the private key computed earlier.
 - Let us say $Q_A = r^{-1}(sR - zG)$
 - Now replace s with $k^{-1}(z + rd_A)$ as in algorithm 1. So, $Q_A = r^{-1}(k^{-1}(z + rd)R - zG)$
 - Now bring out parameters, $Q_A = r^{-1}k^{-1}zR + r^{-1}rk^{-1}dR - zGr^{-1}$
 - Now $r^{-1}r$, will be canceled because $r \cdot \frac{1}{r} = \frac{r}{r}$ so, $Q_A = r^{-1}k^{-1}zR + k^{-1}dR - zGr^{-1}$
 - We know that $R = k \times G$, so, we replace R with kG . Hence, $Q_A = r^{-1}k^{-1}zkG + k^{-1}dkG - zGr^{-1}$
 - Now $k^{-1}k$, will be canceled so, $Q = r^{-1}zG + dG - zGr^{-1}$
 - Now $r^{-1}zG$ and $-zGr^{-1}$ will be canceled, so $Q = dG$
 - Hence got one probable public key Q_A which is $d \times G$. We use in some steps $d \times G$ and in some steps dG however both are same.
 - Similarly we have to process for R' that is $Q_A = r^{-1}(sR' - zG)$.
 - Now, we will be getting two probable public keys however one will match. We have to check using the message. We take both keys and validate the signature, whichever gives the right validation that will be considered as actual public key. In case, both not validates correctly that means signature is modified hence we cannot derive the public key. We need to check with both probable public keys and then conclude the right public key. This is the reason, in case of Ethereum Cryptocurrency along with r and s another parameter v is sent to receivers to directly pick the R instead of trying both. For example, if we see the v value as 27 (0x1c - in hexadecimal) then $v - 27 = 27 - 27 = 0$ means pick first point that is R and if v value is 28 then $v - 27 = 28 - 27 = 1$ means pick second point that is R' . Ethereum uses $v = 27$ or 28 (pre-EIP-155) and $v = 35, 36, \dots$ (based on chain ID, post-EIP-155).

7.6 Private Key Recovery

While using the ECDSA, there is an important requirement that should be followed: The temporary secret key k is used in algorithm 1 and it should not be reused. In case it is used in more than one message and if the attacker captures the signature pairs r_1, s_1 and r_2, s_2 of message m_1 and m_2 then they can recover the temporary secret key k and also the private key d_A . We will see in the following how to recover the keys:

We have messages m_1 and m_2 and both messages use the same k value for signature, then the two signatures are (r, s_1) and (r, s_2) . Here r value is same for both messages because k is same. You may refer the algorithm 1 to know why both will have same r value. The value of s as per algorithm 1 is given in equation 12.

$$s_1 = k^{-1}(z + rd_A) \tag{12}$$

Now, we can rewrite as in equation 13:

$$s_1 = \frac{z_1 + rd_A}{k} \tag{13}$$

Likewise for s_2 in equation 14:

$$s_2 = \frac{z_2 + rd_A}{k} \tag{14}$$

To compute the ephemeral key or temporary secret key k , we subtract s_1 and s_2 as in equation 15 and 16.

$$s_1 - s_2 = \frac{z_1 + rd_A - z_2 - rd_A}{k} \quad (15)$$

rd_A got vanished because $rd_A - rd_A$.

$$s_1 - s_2 = \frac{z_1 - z_2}{k} \quad (16)$$

Now the ephemeral key k can be compute as in equation 17

$$k = \frac{z_1 - z_2}{s_1 - s_2} \quad (17)$$

We know the value of s_1 and s_2 also we can compute the value of z_1 and z_2 from the known messages m_1 and m_2 . You can refer algorithm 1 for generating z from message m .

Now, knowing k , we can recover the private key d_A as follows. According to algorithm 1,

$$s_1 = k^{-1}(z_1 + rd_A)$$

We can write this as

$$s_1 = \frac{z_1 + rd_A}{k}$$

then we can rewrite as

$$s_1 k = z_1 + rd_A$$

$$s_1 k - z_1 = rd_A$$

Finally

$$d_A = \frac{s_1 k - z_1}{r}$$

We know the value of r , s_1 , k and z_1 , so we can compute the private key d_A . Therefore, we should not use the ephemeral key k more than once.

Important Note: In all steps of the above proofs, the ($\text{mod } N$) operation should be applied.

8 Bilinear Pairing

8.1 Preliminaries

In this section, we will discuss various basic concepts that are required for the pairing based cryptography or bilinear pairing.

8.1.1 Torsion Point

Let us take a point P on the elliptic curve defined over F_p and qP should be identity that is the point of infinity then P is called as torsion point. Here q is a scalar value and qP is the scalar multiplication including point doubling and addition. The smallest such positive integer q (if it exists) is called the order of the point P otherwise P is said to have infinite order (means no integer $q > 0$ exists such that $qP = \mathcal{O}$).

8.1.2 Divisor

In general, if we see the divisors or factors are required for the pairing. For example, if we take 12 then we can have factors as (2,6), (3,4) and (1,12). We can use the divisors to find out how numbers can be paired to form a product that is the original number here 12. We know that 1,2,3,4,6 and 12 can divide the number 12 and we can find the pairs using this.

In case of pairing for the cryptography, it is referred to the formal sum of points on the elliptic curve E with integers as shown in below.

$$D = \sum_{P \in E} n_P(P) \quad (18)$$

Here, formal sum is just a way of listing several objects together with integer weights but not going to perform any operation. As per the above equations, we say $D = (P_1, P_2, 3, P_3, 1, P_4, -1, \dots)$ as record of weight list but we do not do any arithmetic operations. This means, we know that point P_1 appears twice, P_2 appears thrice, P_4 appears one time negatively means with negative sign.

In equation 18, where $n_P \in \mathbb{Z}$ that is any integer and only finitely many $n_P \neq 0$. This means there are many P however only finite number of P will get n_P different from zero others may get zero. This means finite number of n_P will be non-zero and others may be zero.

- If $n_P > 0$, then the divisor of a function has a **zero** of order n_P at the point P . A function can be like $f(x, y) = y - mx - c$ that takes input as point $P = (x, y)$ from the curve and evaluate it on $y - mx - c$
- If $n_P < 0$, then the divisor of a function has a **pole** of order $|n_P|$ at the point P .
- If $n_P = 0$, then the function does not appear in the divisor.

We can also say that a divisor is a formal way to describe where a function is zero (vanishes) or goes to infinity (has a pole). For zero (vanishes), let us say, we have function $f = (Z - 3)^2$, then f will be zero at 3 twice means it repeat two times since it has power 2. If we have function $f = (Z - 3)^3$ then f will be zero at 3 thrice means it repeat three times since it has power 3. For pole (infinity), let us consider that $f = \frac{2}{(Z-3)^2}$ then the function f will be infinity on 3 two times.

If we see an example with respect to point, if a function f has a zero of order 2 at P and a pole of order 3 at Q , then the divisor of f is:

$$\text{div}(f) = 2(P) - 3(Q)$$

This means:

- f *vanishes* at P with multiplicity or order 2.
- f *blows up* at Q with multiplicity or order 3.

In summary, if we see a divisor records where a function vanishes or blows up, along with how many times and Zeros contribute positive multiplicities and poles contribute negative ones.

Important Question: What is its role in bilinear pairing? This will be useful when we evaluate f_P at another point Q , we are measuring a deep geometric relationship between P and Q . Let us consider that we have a function f and we take a point P and evaluate using another point that is Q to see their relationship. We can do operations like addition and scalar multiplication on points of the elliptic curve directly comparing points like numbers will not give you the structure or interaction hence the divisor is required. In pairing, we will comparing the points hence needed.

Important Note: If we say f then it is a general function and if we say (f) of $\text{div}(f)$ then it is representing the divisor of a function . For example, we can see the following example, with a following rational function (the rational function is the ratio of two polynomials with no hidden “reminders”). For example, $f(x) = \frac{x-3}{x^2+1}$, here $f(x)$ is rational since both the numerators and denominators are polynomials:

$$f(x) = \frac{(x-2)^2(x+1)}{x^3}$$

Then:

- $f(x)$ is the actual function you can evaluate.

- The **divisor** of f , denoted by (f) , records where $f(x) = 0$ (zeros) and where $f(x) = \infty$ (poles). We can see that in the following:

$$(f) = 2(2) + 1(-1) - 3(0)$$

This means:

- $f(x)$ has a zero of order 2 (twice because of power in sub-function $(x - 2)^2$ for which it reaches 0) at $x = 2$,
- $f(x)$ also has a zero of order 1 (assuming power is 1 in sub-function $(x + 1) = (x + 1)^1$ for which it reaches 0) at $x = -1$,
- and a pole of order 3 (thrice because of power in sub-function x^3 for which it reaches infinity) at $x = 0$.

8.1.3 Pairing and Mapping

Pairing or Mapping is a function that takes input from a single group or multiple groups and map with another group. In general, we say pairing (as bilinear maps) means a function that takes input from two groups and produce the output in target group which is third group. We use the term mapping in case of linear that is linear mapping and pairing is for the bilinear mapping. So, in general the pairing is used for bilinear mapping.

8.1.4 Linear Pairing

The linear pairing can take one input from only one group and perform the operations: addition and scalar multiplication and produce the output in another group. For example,

$$f : G \rightarrow H \tag{19}$$

The function f can take elements from the group G (it can take multiple elements from G like in case of $f(ax + by) = af(x) + bf(y)$, it takes x, y, \dots , however only one group is involved) and performs the operation. However the bilinear pairing will allow us to take input from two groups and perform the operation and produce the output in third group as in the following.

$$f : G_1 \times G_2 \rightarrow H \tag{20}$$

Here G_1 and G_2 may be same ($G_1 = G_2$) or different ($G_1 \neq G_2$) according to the our application requirement and implementation.

8.1.5 Bilinear Pairing

Bilinear pairing is a function that takes input from two different groups and produce the output in a third group as shown in equation 21. In some cases both the input come from the same group and produce the output in another group as shown in equation 22. In the equation 21 and 22, e refers to pairing, G_1 and G_2 are the additive group and G_T is the multiplicative group. It may be possible in some cases the G_1 or G_2 may also be multiplicative. In general pairing refers to the bilinear mapping or bilinear pairing.

$$e : G_1 \times G_2 \rightarrow G_T \tag{21}$$

$$e : G_1 \times G_1 \rightarrow G_T \tag{22}$$

Pairing is mapping from a group to another group satisfying the bilinearity. Here the groups G_1 and G_2 are groups of points on elliptic curves (usually additive groups) and G_T is a multiplicative finite group of a finite field. We can Consider that $G_T \subseteq F_{p^m}^*$

Important Note : Subgroups of finite fields are generally multiplicative groups, whereas G_1 and G_2 are additive groups but not subgroups of the finite field elements themselves.

8.1.6 Why Bilinear pairing for Cryptography

Pairing takes two points from a source group (point on the elliptic curve) and maps to target group (finite field). It takes two points (g^a, g^b) from the source group and map to the target group in such a way that exponent multiplies $(g, g)^{ab}$. Bilinear means multiplication response and can do pairing once because it is moved to the target group after that we cannot do another pairing on it because the target group is the finite field (scalar values for an example) and the input groups are with elliptic curve points. This means, we cannot chain the pairings as pairing on a pairing.

Bilinear pairing can be used in cryptography because

1) it can be used for encryption as follows. left is for the encryption and right for the decryption like applying the public key on one side and private key another side.

$$e(g^a, g^b) = e(g^b, g^a) \quad (23)$$

2) It can allow for the process in the encrypted content like below

$$g^{x_1}, \dots, g^{x_n}, h^{y_1}, \dots, h^{y_n} \rightarrow e(g, h)^{x \cdot y} \quad (24)$$

People see the size and computational time missed the pairing however now is now largely getting explored in Identity Based Encryption, ZK-SNARK, Broadcast based encryption and other protocols. In these protocols, it is not possible to use the existing algorithms such as Elliptic Curve Diffie-Hellman (ECDH) because for an example, we cannot derive the private key from the identity in case of the Identity Based Encryption (IBE).

8.1.7 Properties of Bilinear Pairing

The bilinear pairing has three following properties:

- **Bilinearity:** The bilinear pairing function should map two elliptic curve points of Group G_1 and G_2 to G_T . The bilinearity equation 25 contains:
 - pairing function e
 - P and Q are the points respectively from Group G_1 and G_2 . These points are the vector points (two dimensional that is cartesian co-ordinates (x, y)) from the elliptic curve generated from G_1 and G_2 .
 - The constant scalar values a and b are from the finite integer group (Z_p) ; where p is the prime order of the group and the group will be the multiplicative group (Z_p^*) .

$$e(aP, bQ) = e(P, Q)^{ab} \quad (25)$$

The equation 25 defines that the $e(aP, bQ)$, which is the pairing of two points P and Q multiples with the scalar value a and b respectively are the value of group G_T . The property *bilinearity* means that the scalar multiplication on inputs translates to exponentiation (multiplicative scaling) on the output.

- **Non-Degeneracy:** The output of the pairing should not be 1 that is the identity element of G_T .

$$e(P, Q) \neq 1 \quad (26)$$

If the pairing always produces the output 1, then it carries no meaningful information. In that case, it becomes impossible to verify proofs or signatures correctly, since the pairing check would always succeed even for invalid or forged inputs. It will not be useful for the cryptography. There are case like either P or Q is point of infinity (\mathcal{O}) then it will produce the identity element 1.

$$e(P, Q) = e(\mathcal{O}, Q) = 1 \text{ or } e(P, \mathcal{O}) = 1 \quad (27)$$

Likewise, there are other possibilities that the output will be 1.

- **Computability:** The pairing must be efficiently computable means able to evaluate $e(P, Q)$ in polynomial time.

8.1.8 Different Bilinear Mappings

We can do different bilinear mappings and those are presented in the annexure.

8.1.9 Twist Curve

Let us briefly understand the need of Twist curve then we move on the discussion of Twist curve.

Extension Field In pairing-based cryptography (e.g., BLS, BN, KSS curves), elliptic curves are used over extension fields like F_{q^k} . It is computationally expensive if we directly working with F_{q^k} . However, applying a twist that allows one to map the operation to a smaller subfield: For example, if E is defined over F_{q^k} and has a sextic twist E' , we might be able to do group operations over $F_{q^{k/6}}$ which is much cheaper. Suppose $k = 12$ (common for BN curves) then we can do the group operation with F_{q^2} instead of $F_{q^{12}}$ with sextic twist.

Now let us discuss the twist curve. A twist E' of degree d is another curve that becomes isomorphic to E over F_{p^d} but is defined over a smaller extension field, usually F_{p^m} with $m < d$.

We use the extension field (F_{p^α}) from the base field F_p to bring the points to ensure the non-degeneracy because if both points will be from the base group then it will degenerate. If we go for higher values for d then it will be affecting the performance. Hence we can use the twist curve which will map to the higher values for d but work with the F_{p^m} means $m = 2$. The sextic curve (means degree of 6) is preferred that is F_{p^6} means $\alpha = d6$ however it is computationally not efficient. Hence, the curve can be twisted so that the computational efficiency will be achieved. The twist curve E' is isomorphic to the original curve or base curve E but defined over a smaller extension field F_{p^2} .

Let us see how to compute the $E'(F_{p^2})$:

- Let us take the elliptic curve equation $E = y^2 = x^3 + ax + b$, with $a = 0$, so we will be getting $y^2 = x^3 + b$. The value of b can be taken later.
- Now, the twisted curve $E' = y^2 = x^3 + \frac{b}{\xi}$; where where $\xi \in F_{p^m}$ is a non-residue (an element that is not a d^{th} power in F_{p^m}).
- The degree d corresponds to the order of the twist and it will be 6 if it is sextic twist and usually $m = 2$, so the twist curve is defined over \mathbb{F}_{p^2} . Here, non-residue means take all elements from the F_p , compute the 6^{th} power. If a number that is not in the list then that can be taken. For example, let us say that the value of p is 5 then elements are $\{0, 1, 2, 3, 4\}$. Take all elements and test it.

$$0^6 \text{ mod } 5 = 0$$

$$1^6 \text{ mod } 5 = 1$$

$$2^6 \text{ mod } 5 = 4$$

$$3^6 \text{ mod } 5 = 4$$

$$4^6 \text{ mod } 5 = 1$$

Here, if we see that 2 and 3 are non-residue (not in the sixth root) so we can pick any one of these values for ξ . **Important Note: for simplicity it is taken from F_p otherwise it has to be taken from F_{p^m} and test for the non-residue.**

- ξ must be chosen so that E' is not isomorphic to E over \mathbb{F}_{p^2} , but is isomorphic over \mathbb{F}_{p^6} . Hence, the non-residue is chosen.
- If we see, ξ is in denominator, we can bring to the numerator as ξ^{-1} and then compute the inverse of it as we did earlier and use that value.

- There exists an isomorphism over \mathbb{F}_{p^6} :

$$\psi : E'(\mathbb{F}_{p^2}) \rightarrow E(\mathbb{F}_{p^6})$$

given by:

$$\psi(x, y) = \left(\xi^{1/3}x, \xi^{1/2}y \right)$$

where $\xi^{1/3}$ is a cubic root of ξ in \mathbb{F}_{p^6} .

This means, if we take the point (x, y) from F_{p^2} then by multiplying respectively with $\xi^{1/3}$ and $\xi^{1/2}$, we will get the point from \mathbb{F}_{p^6} .

- Now the question is how to compute the $\xi^{1/3}$?
 - ξ , is carefully chosen so that it will have a cube root in F_{p^6} even if it does not in F_{p^2} . Means ξ should be a cube so that $\xi^{1/3}$ and $\xi^{1/2}$ can be computed. Here, we consider that $q = p^6$

- In case of $\xi^{1/3}$, we can write as $\xi^{\frac{(2q-1)}{3}}$. For example, we solve the congruence means to find x , which is the inverse of 3:

$$3x \equiv 1 \pmod{q-1}$$

We can solve the above using the Extended Euclidean Algorithm however to represent if we see for most values of q , the inverse of 3 modulo $q-1$ is given by:

$$3^{-1} \pmod{q-1} = \frac{2q-1}{3} \pmod{q}$$

Let us see, how the $q-1$ comes,

In a finite field \mathbb{F}_q^* , for any $a \in \mathbb{F}_q^*$ and integer n , we have:

$$a^n \pmod{q} = a^{n \pmod{q-1}} \pmod{q}$$

For example, let us take $F_7^* = \{1, 2, 3, 4, 5, 6\}$, where $q = 7$. Then we chose any value $a \in F_7^*$ and compute its power then we can see that will be equal to $a^{n \pmod{q-1}} \pmod{q}$. Here n can be any integer value. If we take $q = 7$, $n = 10$ and $a = 3$ then $n \pmod{q-1} = 10 \pmod{6} = 4$. Now $3^4 \pmod{7} = 81 \pmod{7} = 4$ and this will be equal to $a^n \pmod{7} = 3^{10} \pmod{7} = 59049 \pmod{7} = 4$. It shows both are equal because it is as per the Fermat's Little theorem.

- **Fermat's Little Theorem** states that
 - ⇒ If p is a prime number, then for any integer a , the number $a^p - a$ is an integer multiple of p or p divides $a^p - a$ (means $\frac{a^p - a}{p} = 0$). In modular arithmetic, this is expressed as $a^p \equiv a \pmod{p}$. For example, if $a = 2$ and $p = 7$, then $2^7 = 128$, and $128 - 2 = 126 = 7 \times 18$ is an integer multiple of 7.
 - ⇒ If a is not divisible by p (means $\frac{a}{p} \neq 0$), that is, if a is coprime to p , then Fermat's little theorem is equivalent to the statement that $a^{p-1} - 1$ is an integer multiple of p , $a^{p-1} \equiv 1 \pmod{p}$. For example, if $a = 2$ and $p = 7$, then $2^6 = 64$, and $64 - 1 = 63 = 7 \times 9$ is a multiple of 7.

- Now we will see how to compute the $\xi^{1/2}$?
We can use this equation $\xi^{\frac{q+1}{4}} \pmod{q}$ provided $q \equiv 3 \pmod{4}$ otherwise we need to go for complex Tonelli–Shanks algorithm.
- The reason behind choosing the twist is Computing group operations in F_{p^2} is much faster than in F_p^{12} or F_{p^6} . Here, the curve point is based on F_{p^2} however while computing ξ value and $\xi^{1/2}$ as well as $\xi^{1/3}$ we should consider F_{p^6} or $F_{p^{12}}$

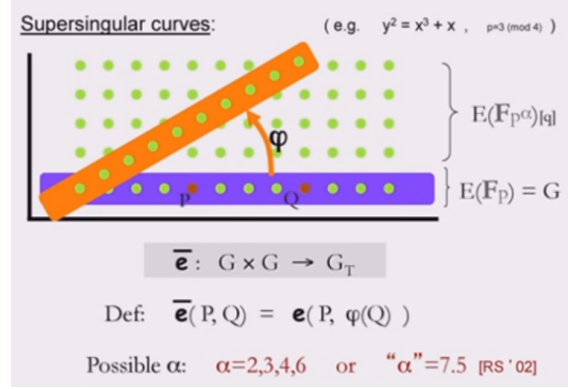


Figure 9: Symmetric Pairing [figure copied]

8.2 Symmetric Pairing

We do pairing by taking inputs from two groups. In case both the inputs (P and Q) are from the same group (G) then we say it is symmetric pairing. We should not misinterpret this as the linear mapping since we take the input from one group. Here we take two inputs two groups ($G_1 = G_2$) and map to the third group. The equation 28 shows the symmetric pairing.

$$e : G_1 \times G_1 \rightarrow G_T \quad (28)$$

If we choose both the points P and Q from the base group that is G_1 then we get always 1 ($e(P, Q) = 1$) that is degenerate and will not follow the bilinear property. Hence we need to go for a special curve which is supersingular curve with automorphism or endomorphism. If we take the point P from G_1 and Q from G_1 then we can move it to G_2 by doing φQ . Now the pairing will be like the following equation 29 and as show in figure 9.

$$e(P, Q) = e(P, \varphi(Q)) \quad (29)$$

Here φ refers to the scaling means raise each coordinate to $q^t h$ power in F_q . For example, if we have $Q = (x, y)$ and the field is F_{11} then $\varphi(Q) = (x^{q=11} \pmod{11}, y^{q=11} \pmod{11})$. The figure 9 shows that we choose two points P and Q from the base field which is derived over $E(F_p)$, then perform φQ will map the point to the extended field that is $E(F_{p^\alpha})[q]$, where q is torsion.

This is less secure or less efficient curves. The Weil pairing is symmetric. The symmetric pairing needs the supersingular curve, which may be less secure at higher security levels because of smaller embedding degrees, but pairing computations on them are often more efficient if smaller embedding degrees.

Embedding degree : The k is the embedding degree and that is part of $r|(p^k - 1)$. Here, we choose r first, usually as a large prime to guarantee security in the elliptic curve group. We can say r is the order of the subgroup and p is the prime of the based Field F_p . We get k from this $r|(p^k - 1)$. The reason for this embedding degree k is to determine that how secure the curve is against the Discrete Logarithmic Problem (DLP). For example, $k = 1, 2, 3, 4, 6$ for the supersingular curve, $k = 12$ for Barreto-Naehrig (BN) curves and $k = 16, 18, 24, 30$ for higher security levels.

Endomorphism : An endomorphism is a special type of homomorphism where the domain and codomain are the same object. For example, $f : X \rightarrow X$, the mapping of f by taking element from X (domain) to X (codomain) that is to itself. The supersingular curve has additional algebraic structure that is it has the endomorphism, which is the mapping. It takes the point from the base field and maps to some point in the extension field. This feature exist in the supersingular curve.

Group and Field : Here the curve points (x, y) are member of group G , however the coordinates x and y individually are from the finite field defined over p . Likewise, curve constants a and b also from F_p .

8.2.1 Supersingular Curve

An elliptic curve E defined over a finite field F_p is called supersingular if it has special algebraic properties that make it behave very differently from most curves that is called ordinary curves. In case of the symmetric pairing it is must to choose the supersingular curve otherwise the pairing of points from the same group will output 1 means degeneracy that violates the non-degeneracy property of bilinear pairing.

Let us see what makes the curve supersingular. Consider the following curve with finite field F with prime $p = 5$. The value of a and b in the curve is 0 and 1 respectively according to the simplified Weierstrass equation.

$$E : y^2 = x^3 + 1 \text{ over } F_5 \quad (30)$$

The number of points in this curve including the point of infinity is 6 that is $\#E(F_p) = 6$. An elliptic curve over F_p is supersingular if only if its trace of Frobenius is $t \equiv 0 \pmod{p}$. The trace of frobenius is $\#E(F_p) = p + 1 - t$. We can write it as follows.

$$t = p + 1 - \#E(F_p)$$

where

$$t = 5 + 1 - 6 = 0$$

This satisfies the equation $t \equiv 0 \pmod{p}$ that is $0 \equiv 0 \pmod{5}$. Hence the curve is super singular. It is worth to note that if the following holds then we can always get the supersingular curve.

$$p \equiv 2 \pmod{3} \Rightarrow p \pmod{3} = 2 \pmod{3} \quad (31)$$

In the above example, the value 5 is chosen for p and that satisfies the $5 \equiv 2 \pmod{3}$.

The drawback of the supersingular curve is it is not natural means very rare curves are supersingular. This is the reason we go mostly for the asymmetric pairing where we do not need supersingular curve.

Trace of Frobenius For an elliptic curve E defined over a finite field F_p , the trace of frobenius, usually denoted t , is a value that describes how far the number of points on the curve deviates from $p + 1$. This is useful for Point counting, supersingularity test and pairing construction.

8.3 Asymmetric Pairing

If both inputs that is P and Q from the different groups (G_1 and G_2) then we say it is asymmetric pairing. The equation 32 shows the asymmetric pairing where $G_1 \neq G_2$ and as show in figure 10. We can see from figure 10 that the G_2 is bigger because the G_2 is the extension field with more points.

$$e : G_1 \times G_2 \rightarrow G_T \quad (32)$$

This can give faster pairing computations with smaller keys. The Tate pairing and its optimal variant that is Ate pairing works mostly with asymmetric. In the asymmetric pairing,

- $G_1 \subseteq E(F_p)$ points on the elliptic curve base field.
- $G_2 \subseteq E'(F_p^q)$ or actually $E'(F_p^{2q})$ points on a twist of the curve over an extension field. We have discussed the twist curve already hence the points are from $E'(F_p^{2q})$ not from $E'(F_p^q)$
- $G_T \subseteq F_{p^a}^*$ is the target multiplicative group of the extension field ($F_{p^{12}}$). G_T is not a group of curve points, it is a subgroup of the multiplicative group of the field.

The twist E' is isomorphic to the original curve E over the larger extension field F_{p^q} but it can be defined over a smaller field $F_{p^{k/d}}$.

This smaller field $F_{p^{k/d}}$ is a subfield of F_p^k because $d|k$ (where d divides k) means

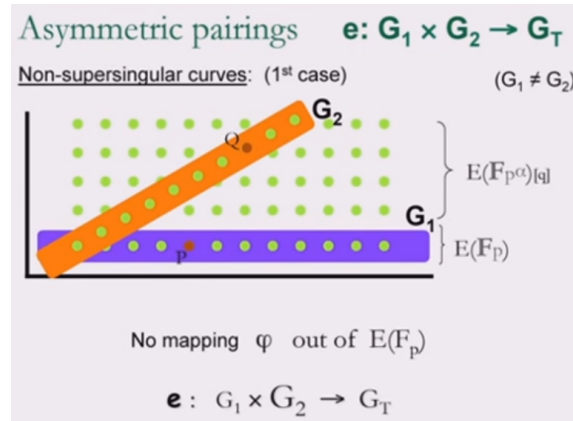


Figure 10: Asymmetric Pairing [figure copied]

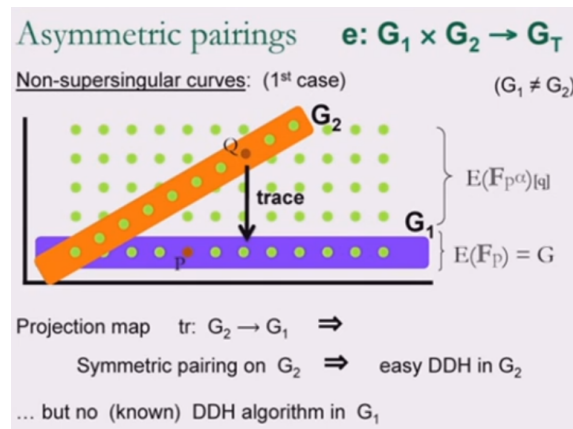


Figure 11: Asymmetric Pairing - Tracing [figure copied]

$$F_{p^{k/d}} \subseteq F_{p^k}$$

For an example, let us say $k = 12$ and we use the sextic twist (degree 6 twist), then $d = 6$ because $6|12$. Instead of F_p^{12} we use F_p^2 which is cheaper means efficient with less points. This is a key optimization in asymmetric pairings.

Why sextic twist not the 2 (quadratic), 3 (cubic), 4, 6 (sextic), etc.

- Efficient Arithmetic: F_p^2 (for the sextic twist) is significantly cheaper than F_p^{12}
- Embedding Degree Compatibility: Many pairing-friendly curves, like Barreto-Naehrig (BN) curves, have an embedding degree $k = 12$ and a degree 6 twist is naturally compatible.

Issue: Trace Map on an Elliptic Curve The trace map (also called the trace of Frobenius) is a group homomorphism related to how points on an elliptic curve over an extension field relate back to points over the base field.

If we choose the point P from G_1 and Q from G_2 , then the pairing is Decisional Diffie Hellman hard however there is a concept of trace that is given a point in G_2 can trace a point in G_1 make DDH easy. We can see from figure 11 that it get traced from G_2 to G_1 .

Here we take the point $Q = (x_Q, y_Q)$ from the extension field and by applying the Frobenius Endomorphism, we can another point that lives in the base field F_p . The sum of its Frobenius conjugates back to a point in F_p as follows, where π is the Frobenius Endomorphism.

$$(x_P, y_P) = \text{Tr}(Q) = Q + \pi(Q) + \pi^2(Q) + \dots + \pi^{k-1}(Q)$$

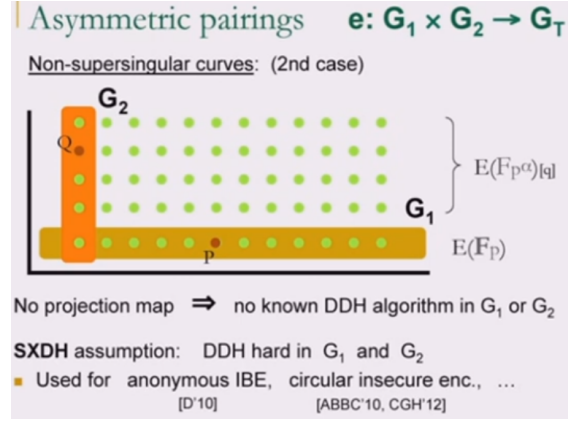


Figure 12: Asymmetric Pairing - No Tracing [figure copied]

Solution To destroy the tracemap, we should use the special subgroup and ensure that the tracemap will always give 1 as the result. This will be considered as SXDH assumption that means symmetric external diffie hellman problem. Here DDH is hard in both G_1 and G_2 . We have to choose points as shown in figure 12. To destroy the trace, we have to choose points $Q \in G_2$ such that

$$\text{Tr}(Q) = Q + \pi(Q) + \pi^2(Q) + \dots + \pi^{k-1}(Q) = \mathcal{O},$$

where π is the Frobenius endomorphism and \mathcal{O} is the identity element on the elliptic curve.

8.3.1 Summary of types of pairing

There are three types of pairing: Type 1, Type 2 and Type 3. The details are as follows:

Type 1 Pairing This is the symmetric pairing where the pairing is defined over the single group.

$$e : G_1 \times G_1 \rightarrow G_T$$

This is considered as simple since the points are from the same group however less efficient when compared with Type 2 and Type 3. Also, it needs the supersingular curve.

Type 2 Pairing This is the asymmetric pairing where the pairing is defined over two different groups.

$$e : G_1 \times G_2 \rightarrow G_T$$

However, there exists an efficiently computable homomorphism $\phi : G_2 \rightarrow G_1$ but there is no efficient computational homomorphism from G_1 to G_2 that is $\phi : G_1 \rightarrow G_2$. This has better efficiency than symmetric pairings but sometimes weaken the security assumptions because of efficiently computable homomorphism $\phi : G_2 \rightarrow G_1$. **Here, we use the term homomorphism not specifically as isomorphism because the mapping is not necessarily bijective (one-to-one: Any element in G_2 is mapped exactly to one element in G_1 and onto: Every element in G_1 is mapped by some element in G_2 . We can also say in case of bijective there exist an inverse, for example $W^{-1}(W(a)) = a$).** It is not bijective means no one-to-one and no onto thus no inverse possible and homomorphism only needs to preserve the group operation like $f(a + b) = f(a) + f(b)$.

In type 2 pairing, the pairing of two points from same group cannot be paired hence DDH is hard however, if we take a point from the base field ($P = (x_1, y_1)$) and extension field ($Q = (x_2, y_2)$) then we can map the extension field point (Q) to a point in the base field which is Q' ($Q' = (x'_2, y'_2)$) and solve the DDH problem since we have point from both the fields. Hence, we need to go for the Type 3 pairing.

Type 3 Pairing This is the asymmetric pairing where the pairing is defined over two different groups.

$$e : G_1 \times G_2 \rightarrow G_T$$

However, there is no efficiently computable homomorphism for G_1 to G_2 and G_2 to G_1 that is $\phi : G_2 \rightarrow G_1$ and $\psi : G_1 \rightarrow G_2$. The pairing is considered as most efficient and secure. This is used in Barreto–Naehrig (BN) curves. The figure 12 shows the Type 3 curve points of G_2 .

8.4 Group Homomorphism

A group homomorphism is a function between two groups that preserves the group operation. Suppose you have two groups that is (G, \cdot) - a group with any operation and (H, \times) — another group with multiplicative operation then a function $f : G \rightarrow H$ is a group homomorphism if for all elements $a, b \in G$ if it satisfies $f(a \cdot b) = f(a) \times f(b)$, which is in H that is $f(a) \times f(b)$ in H . In case of additive group that means if $(G, +)$ then $f(a + b) = f(a) + f(b)$ and in case (G, \times) then $f(a \times b) = f(a) \times f(b)$.

8.5 Consequence of Pairing

Here, we see the consequence of pairing with respect to the security.

8.5.1 Decisional Diffie-Hellman (DDH) problem

Using pairing, it is easy to solve the DDH problem that is given g^x, g^y, g^z and g all belongs to G , we can test $xy = z$ or not. We can see it in the equation 33 that it can be verified since this is the bilinear property.

$$e(g, g^z) \rightarrow e(g^x, g^y) \tag{33}$$

8.5.2 Discrete Log Problem (DLP)

Given g^x, g belongs to G we can transfer the discrete log problem from elliptic curve point to the finite field as shown in equation 34. If the G_T is small then Discrete Log can be solved otherwise complex.

$$e(g, g) \rightarrow e(g, g^a) \tag{34}$$

Here it moves the Discrete Log problem of the elliptic curve to the finite field because the pairing is done on the target field which is G_T . Hence the G_T should be large enough to make the DLP complex.

Computational Diffie-Hellman (CDH) problem: The CDH in the source group is hard eventhough it is not hard in the target group. We can see from the following that the target group gets the computed ab however getting in the source group is very hard.

$$e(g^a, g^b) \rightarrow e(g, g)^{ab}$$

8.5.3 Bilinear DDH (BDDH) Problem

As shown earlier, DDH is solvable, so we move on to BDDH. This will be hard by introducing another group. In DDH all g^x, g^y, g^z and g belongs to the same group. If we introduce another parameter h from another group (which is the bilinear) then it is complex. We consider, given h, g, g^x , and g^y from two groups checking $e(h, g)^z = 0$ that means $z = xy$ is very complex. However, trilinear can break the BDDH however making trilinear will be complex and that can be discussed in later documents. and

8.6 Miller's Algorithm

Miller's algorithm is a core part of computing bilinear pairings (like the Weil or Tate pairing or others) on elliptic curves. It was introduced by Victor Miller in 1984 (however published in 2004) and remains critical to efficient pairing-based cryptography. This is to solve the pairing function $e(P, Q)$. The initial f value in the equation will be 1. The values are computed as follows.

$$e(P^a, Q) = e(P, Q)^a = \left(f_{r,P}(Q)^{\frac{q^k-1}{r}} \right)^a = f_{r,P}(Q)^{a \cdot \frac{q^k-1}{r}}$$

The $f_{r,P}(Q)$ means, we define a function defined using P and evaluated at Q . We create a rational function using P and use the co-ordinates of $Q = (x_Q, y_Q)$ to solve it. The Miller's algorithm to solve the $f_{r,P}(Q)$ is given in algorithm 19. The algorithm start iterating from $n - 2$ because the bits are represented from from 0 to $n - 1$. However, we can see that $n - 1$ also dropped because the first Most Significant Bit (MSB) is considered by default so starts from $n - 2$. We have also seen this earlier in point doubling. Miller's algorithm provides an efficient way to compute these rational functions, using a double-and-add method similar to scalar multiplication on elliptic curves. The Miller's algorithm to compute the line function is given in algorithm 19. In all operation of the algorithm, the modulus operation has to be applied.

Why Pairing? The pairing (in some cases like in Tate or Weil pairing) involves evaluating certain rational functions associated with elliptic curve points. Evaluating the rational function naively is extremely slow, especially for cryptographically relevant curve sizes. Miller's algorithm does this efficiently using double-and-add method given in algorithm 19.

Algorithm 3 Miller's Algorithm for Computing $f_{r,P}(Q)$

Require: Scalar $r = (r_{n-1}, \dots, r_0)_2$, point P on elliptic curve E , point Q

Ensure: Value of $f_{r,P}(Q)$

```

1:  $R \leftarrow P$ 
2:  $f \leftarrow 1$ 
3: for  $i = n - 2$  to 0 do ▷ Doubling Step
4:   Compute slope  $\lambda \leftarrow \frac{3x_R^2 + a}{2y_R}$ 
5:   Compute  $x_{2R} \leftarrow \lambda^2 - 2x_R$ 
6:   Compute  $y_{2R} \leftarrow \lambda(x_R - x_{2R}) - y_R$ 
7:   Compute line function at  $Q$ :
           
$$g_{R,R}(Q) \leftarrow (y_Q - y_R - \lambda(x_Q - x_R))$$

8:    $f \leftarrow f^2 \cdot g_{R,R}(Q)$ 
9:    $R \leftarrow (x_{2R}, y_{2R})$ 
10: if  $r_i == 1$  then ▷ Addition Step
11:   Compute slope  $\lambda \leftarrow \frac{y_R - y_P}{x_R - x_P}$ 
12:   Compute  $x_{R+P} \leftarrow \lambda^2 - x_R - x_P$ 
13:   Compute  $y_{R+P} \leftarrow \lambda(x_R - x_{R+P}) - y_R$ 
14:   Compute line function at  $Q$ :
           
$$g_{R,P}(Q) \leftarrow (y_Q - y_R - \lambda(x_Q - x_R))$$

15:    $f \leftarrow f \cdot g_{R,P}(Q)$ 
16:    $R \leftarrow (x_{R+P}, y_{R+P})$ 
17: end if
18: end for
19: return  $f$ 

```

8.7 Weil Pairing

This is one of the symmetric pairing where both the points will be from the same group. However, it uses the supersingular curve to ensure the non-degeneracy property. Let E be an elliptic curve

over a finite field, and let r be a prime number that is the order of sub-group. $E[r]$ is the sub-group of r torsion points. This means we take a torsion point N and do rN so that we get the point of infinity \mathcal{O} . Now, let us take two distinct points P and Q

$$P, Q \in E[r]$$

The Weil pairing is defined as:

$$e_r(P, Q) = \frac{f_P(D_Q)}{f_Q(D_P)}$$

Where:

- f_P, f_Q are rational functions such that:

$$\operatorname{div}(f_P) = r(P) - r(\mathcal{O}), \quad \operatorname{div}(f_Q) = r(Q) - r(\mathcal{O})$$

- A rational function f on an elliptic curve E is a function that can be expressed as the ratio of two polynomials in the coordinates x and y defined on the curve that is

$$f = \frac{g(x,y)}{h(x,y)}; \text{ where } g \text{ and } h \text{ are polynomials.}$$

- A divisor is a formal way to describe where a function is zero (vanishes) or goes to infinity (has a pole)

For example, if a function f has a zero of order 2 at P and a pole of order 3 at Q , then the divisor of f is:

$$\operatorname{div}(f) = 2(P) - 3(Q)$$

This means:

- f *vanishes* at P with multiplicity or order 2.
- f *blows up* at Q with multiplicity or order 3.

It is important to note that vanish does not happen at $2P$, it happens at P itself however we need to see its order so its multiple or order that is 2. similarly for the pole that blows up or reach infinity at \mathcal{O} at its multiple or order 3. The Divisor keeps only these zeros and poles others if any not strong will be discarded.

The divisor $rP - r\mathcal{O}$ involves the point P with multiplicity r , and the point at infinity \mathcal{O} with multiplicity r . However, when working with functions f_P, f_Q , and their divisors, if you directly use $rP - r\mathcal{O}$ for both P and Q , their supports may overlap or coincide with other points $(P, Q, -P, Q, \mathcal{O})$ involved in the pairing calculation. Choosing $D_P = (P + R) - (R)$ "shifts" the divisor by R , so that the supports of D_P and D_Q are disjoint or at least do not intersect problematic points like $P, Q, \mathcal{O}, -P, -Q$.

- D_P, D_Q are divisors linearly equivalent to $(P) - (\mathcal{O})$, $(Q) - (\mathcal{O})$, and chosen to avoid the support of $\operatorname{div}(f_Q)$, $\operatorname{div}(f_P)$, respectively.

For example, choose a point $R \in E$, such that:

$$R \notin \{P, Q, \mathcal{O}, -P, -Q\}$$

Then set:

$$D_P = (P + R) - (R), \quad D_Q = (Q + R) - (R)$$

Hence, the value $e_r(P, Q) \in \mu_r \subseteq \mathbb{F}_{q^k}$ is a non-degenerate, bilinear map that outputs an r -th root of unity:

$$e_r(P, Q)^r = 1; \text{ this is equal to } z^r = 1 \text{ where } z \in \mu_r \subseteq \mathbb{F}_{q^k}$$

Let f_P be the rational function with divisor

$$\operatorname{div}(f_P) = r(P) - r(\mathcal{O})$$

Then evaluating f_P at the divisor

$$D_Q = (Q + R) - (R)$$

gives:

$$f_P(D_Q) = \frac{f_P(Q + R)}{f_P(R)}$$

Similarly, let f_Q be the rational function with divisor

$$\text{div}(f_Q) = r(Q) - r(\mathcal{O})$$

Then evaluating f_Q at the divisor

$$D_P = (P + R) - (R)$$

gives:

$$f_Q(D_P) = \frac{f_Q(P + R)}{f_Q(R)}$$

Therefore, the Weil pairing is computed as:

$$e_r(P, Q) = \frac{f_P(D_Q)}{f_Q(D_P)} = \frac{f_P(Q + R)}{f_P(R)} \cdot \frac{f_Q(R)}{f_Q(P + R)}$$

We should not think that $D_P = (P + R) - (R)$ as $D_P = P$, because $(P + R) - (R)$ is linearly equivalent to $(P) - (\mathcal{O})$, but it is not equal to the point P (so should not consider as point). Another important point is that the divisor equations will never be computed it is only to show that there is a need to take another point R to fulfill the non-degeneracy.

”Linear equivalent” can refer to two equivalent systems of linear equations with the same solution set, or a single equivalent linear equation that has the same solution as another, different-looking equation. Example: $x - 2 = 3$ and $x + 1 = 6$ are equivalent because both simplify to $x = 5$.

Now, the f_Q and f_P can be applied to the Miller’s algorithm to compute the value. The Miller’s algorithm will be called four times however if there is any reuse of the computed value possible then we can consider that the call will be only two times. If we see the Miller’s algorithm, it will take only two points however in case $f_P(Q + R)$ there are three points. The point Q and R will added with point addition and given as the input to Miller’s algorithm to confirm only two points as input.

Important Note In the Weil pairing equation, we do not see the parameter r however the Miller’s algorithm needs it. It is implicit that the r is present because we compute the subgroup based on r only.

8.8 Tate Pairing

Tate pairing is a bilinear map defined on elliptic curves over finite fields, used in cryptography for constructing identity-based encryption, signatures, and other protocols. This can be implemented based on symmetric as well as asymmetric pairing. Tate pairing is defined as:

$$e_r : E[r] \times \frac{E(\mathbb{F}_{q^k})}{rE(\mathbb{F}_{q^k})} \rightarrow \mu_r$$

or

$$e_r : E[r] \times E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k}) \rightarrow \mu_r$$

where:

- $E[r]$ is the group of r -torsion points (points P with $rP = \mathcal{O}$). Here r is the prime order of the subgroup.
- \mathbb{F}_{q^k} is an extension field where the pairing values live.
- μ_r is the group of r -th roots of unity in $\mathbb{F}_{q^k}^*$.
- k is the embedding degree, the smallest integer such that $r \mid q^k - 1$.

In more detail,

Term	Meaning
$E(\mathbb{F}_{q^k})$	Full elliptic curve group over \mathbb{F}_{q^k}
$rE(\mathbb{F}_{q^k})$	Subgroup of all r -multiples of points
$E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k})$	Group of cosets, i.e., equivalence classes

We take a point Q from the $E(\mathbb{F}_{p^k})$, and another point which is Q_1 that is rR where R is the point from the $E(\mathbb{F}_{p^k})$ and r is the order of that subgroup. While doing the operation, instead of Q , the Q_1 will be applied which is $Q + rR$. We do this to ensure the non-degeneracy.

If $Q \in rE(\mathbb{F}_{p^k})$, then for all $P \in E[r]$, the pairing is degeneracy:

$$e_r(P, Q) = 1$$

This is because, $rE(\mathbb{F}_{p^k})$ is the subgroup of all r -multiples of points in $E(\mathbb{F}_{p^k})$, and the Tate pairing becomes trivial (degenerate) on this subgroup. So, we should not pick Q from $rE(\mathbb{F}_{p^k})$ and use directly.

To ensure that the pairing is non-degenerate (i.e., $e_r(P, Q) \neq 1$ for some P and Q), we define the pairing on the quotient group by removing such Q that produces degeneracy:

$$e_r : E[r] \times (E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k})) \longrightarrow \mu_r$$

This means that the second argument to the pairing is not just any point $Q \in E(\mathbb{F}_{p^k})$, but rather a coset $[Q] \in E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k})$. Two points Q and Q' represent the same coset if:

$$Q' = Q + rR \quad \text{for some } R \in E(\mathbb{F}_{p^k})$$

Thus, the pairing is well-defined and does not depend on which representative of the coset is chosen. The Tate pairing is

$$e_r(P, Q) = f_P(Q)^{\frac{q^k-1}{r}},$$

where, $f_P(Q)$ means evaluating the function f_P at the point Q . The exponent $\frac{q^k-1}{r}$ ensures the value lies in the group μ_r that is target group. We know the value of all parameters from the above discussion hence we can call the Miller's algorithm and compute the value. The power $(\frac{q^k-1}{r})$ on the returned value will be computed to get the final pairing output that is $e_r(P, Q)$.

Coset: Let us say, we have a Group G and its subgroup $H \subseteq G$, then the coset is defined as: with a fixed element g , we compute the coset value that is $g \cdot h$ where $h \in H$.

Important Note: We can say that the Tate pairing is half of the Weil Pairing because we can see by comparing both the pairing techniques that Tate pairing implements inline function once, making it generally more efficient. In case of Weil pairing it is twice means it calls for both f_P and f_Q . Also, possibly Miller's algorithm will be called four times if the R not chosen properly in Weil Pairing otherwise it will re-use the partial computation.

What makes to do the same task in less use of Miller's algorithm in Tate? There is a possibility of ambiguity during running the Miller's algorithm. When we run Miller's algorithm that means we build $f_{n,P}$ by iteratively running line functions that arise from point doubling and addition steps. Each iteration computes a "partial line function" $l_{A,B}(Q)$, which is defined up to a multiplicative constant

$$l'_{A,B}(Q) = c \cdot l_{A,B'}(Q); \text{ where } c \in \mathbb{F}_{q^k}$$

This becomes the ambiguity. To overcome this, in case of Weil pairing, the computation is done twice. We see, the line function computation as follows.

$$\frac{f_{n,P}(Q)}{f_{n,Q}(P)}$$

However, it is as follows.

$$\frac{f_{n,P}(Q) \cdot c_P^n}{f_{n,Q}(P) \cdot c_Q^n} = \frac{f_{n,P}(Q)}{f_{n,Q}(P)} \cdot \left(\frac{c_P}{c_Q}\right)^n$$

Now, $\left(\frac{c_P}{c_Q}\right)^n$ this vanishes if $c_P = c_Q$ otherwise also it vanishes because of the n^{th} power. Here n is the torsion order of the subgroup. Recall, we use the $P \cdot r$ to form the subgroup, P is the torsion point or generator and $r \cdot P = \mathcal{O}$. We consider here $n = r$.

Another important question, if the n^{th} power is trivial then why do we need $\left(\frac{c_P}{c_Q}\right)^n$ and Weil pairing can call Miller's algorithm once. The answer is c_Q^n is not guaranteed to be trivial however the ratio will be trivial that means $c_Q^n \neq 1$ but $\left(\frac{c_P}{c_Q}\right)^n = 1$. Hence, the Miller's algorithm is called twice in Weil pairing thus increases the computation time.

In case of the Tate pairing, that ambiguity is handled in different way and removed. We see in the following that the c ambiguity occurs however that is removed with the help of the Fermat's theorem. Hence, one time running Miller's algorithm is sufficient.

$$\begin{aligned} (c f_{n,P}(Q))^{\frac{q^k-1}{n}} &= c^{\frac{q^k-1}{n}} f_{n,P}(Q)^{\frac{q^k-1}{n}} \\ &= f_{n,P}(Q)^{\frac{q^k-1}{n}} \end{aligned}$$

In this $c^{\frac{q^k-1}{n}}$, if we see, $c^{q^k-1} = 1$ then $1^{1/n} = 1$ [because anything raised to 1 is 1]. c^{q^k-1} . For example, as we have seen already $a^{p-1} \bmod p = 1 \bmod p$ then $2^{7-1} \bmod 7 = 64 \bmod 7 = 1 \bmod 7$. Hence, $c^{q^k-1} = 1$. Following including, c^n

$$\begin{aligned} (c^n f_{n,P}(Q))^{\frac{q^k-1}{n}} &= c^{n \frac{q^k-1}{n}} f_{n,P}(Q)^{\frac{q^k-1}{n}} \\ &= f_{n,P}(Q)^{\frac{q^k-1}{n}} \end{aligned}$$

In this $c^{n \frac{q^k-1}{n}}$, if we see, $c^{nq^k-1} = 1$ then $1^{n(1/n)} = 1^{n/n} = 1$ [because anything raised to 1 is 1]. For example, as we have seen already $a^{p-1} \bmod p = 1 \bmod p$ then $2^{7-1} \bmod 7 = 64 \bmod 7 = 1 \bmod 7$. Hence, $c^{q^k-1} = 1$. Hence, in one call of Miller's algorithm, the Tate pairing computes the inline function output and the ambiguity is removed. Hence, faster when compared with Weil Pairing.

Next question comes, when we say $c^{\frac{q^k-1}{n}}$ then cannot $f_{n,P}(Q)^{\frac{q^k-1}{n}} = 1$. It will not be because the value in $f_{n,P}(Q)$ is different means Raising to $\frac{q^k-1}{n}$ removes only the constant n^{th} power factor, not the actual Miller function value.

Summary: Raising to the exponent $(q^k - 1)/n$ removes the n -th power ambiguity c^n while keeping the actual pairing value intact in μ_n . Even c is not raised to n then it will not be 1.

8.9 Linearity and Bilinearity

Linearity is the property that multiplying a point by a scalar corresponds to exponentiation in the target group. For example, in $e(aP, Q) = e(P, Q)^a$, we do scalar multiplication (aP) in $e(aP, Q)$ but it is mapped to exponentiation in the target that is $e(P, Q)^a$. Similarly, the group addition map to the multiplication in the target group.

The bilinearity $e(aP, bQ) = e(P, Q)^{ab}$ can be broken down into two linearities:

$$\begin{aligned} e(aP, Q) &= e(P, Q)^a \\ e(P, bQ) &= e(P, Q)^b \end{aligned}$$

If we see, the linearity is applied only to one input that means either to P or Q , however in bilinearity this linearity works in both inputs that means for both P and Q as shown below.

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

9 BLS (Boneh–Lynn–Shacham) Signature

BLS is the signature algorithm and it is based on Type 3 asymmetric pairing. The step by step process of the BLS signature is as follows

- Step 1: Choose the curve and prime field. For example prime field $p = 23$ and the curve equation is $E : y^2 = x^3 + 1 \pmod{23}$
- Step 2: Generate Group G_1 by generating a generator point $P (g_1)$ at random and the group order n (number of points on the group G_1). We can write every other points as

$$R = k \times P \text{ where } k \in [0, n - 1]$$

- Step 3: Generate Group G_2 by generating a generator point let us say $Q (g_2)$. As seen earlier, this will be the extension field.
- Step 4: Generate private key $sk \in Z_n$
- Step 5: Compute the public key $pk = sk.Q \in G_2$ where Q is from G_2
- Step 6: Signing the message m , where Hash m deterministically onto a point in G_1
 $H : \{0, 1\} \rightarrow G_1, h = H(m)$
- Step 7: Compute the signature
 $\sigma = sk.h \in G_1$
- Step 7: Signature Verification: Given m, σ and pk , and known P and Q , verify the signature using pairing.

- First compute the hash of the received message $h = H(m) \in G_1$
- Verify using (Tate) pairing that $e(\sigma, Q) \stackrel{?}{=} e(h, pk)$; where Q is the generator of G_2 .
This works because of the following

$$e(\sigma, Q) = e(sk.h, Q) = e(h, sk.Q) = e(h, pk)$$

- Compute the pairing value as follows

$e(P, Q) = f_{r.P}(Q)^{(p^k-1)/r}$; where compute $f_{r.P}(Q)$ using the miller' algorithm. Here r is the order of the curve. In the step 2, n is considered as the order hence $r = n$. The power $(p^k - 1)/r$ will be computed on the Miller's algorithm returned value. In all operations, we need to apply the modulus function for example, $pk = sk.Q$ to be $pk = sk.Q \pmod{n}$

- Compute the same for both the sides. If both are equal that means the proof is valid and message not modified otherwise proof is invalid and message may be modified.

You can also see the summary of the BLS signature in the figure 13, where P as g_1 , Q as g_2 and sk as the private key. The aCDH stands for Augmented Computational Diffie-Hellman problem and this is different from CDH. In case of CDH, all elements are from the same group; however, in case of aCDH elements are from two groups.

9.1 Curve

The curves that will be used for the bilinear pairing are Barreto-Naehrig (BN) curves, BLS12 curves, MNT curve and so on. To see more on BN curves,

- BN curves are a family of pairing-friendly elliptic curves introduced by Paulo Barreto and Michael Naehrig in 2005.
- They have embedding degree $k = 12$, which means the pairing maps points from elliptic curve groups over a base field F_p to a finite field extension $F_{p^{12}}$. However, it will be F_{p^2} if we go with sextic.
- BN curves are usually given by a Weierstrass equation: $E : y^2 = x^3 + b$ over F_p with 256 bits or BN-254 with 254 bits and $b = 3$ mostly.

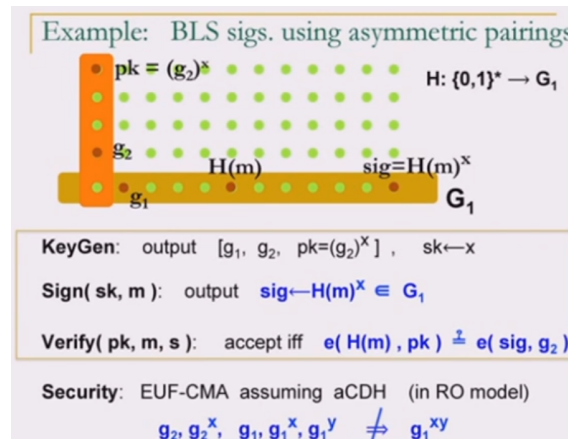


Figure 13: BLS Signature [figure copied]

9.2 How the hashing maps to group point

Simply compute the cryptographic hash (SHA256, SHA512, or any) of the message and take that value as x . Now compute, y of the elliptic curve equation $y^2 = x^3 + ax + b$. If the point is part of the curve then the point will be chosen otherwise increase x by 1 and try it. This is called Try-and-increment method (simplest, but slow). The other method is map to curve algorithm includes Simplified SWU (Shallue–van de Woestijne–Ulas) method: Used for many curves (including BN curves), Elligator 2: For certain curve forms and Icart’s function: For some curves. The BLS prefer the map to curve algorithm.

10 Zero-Knowledge

The zk-SNARK (zero-knowledge Succinct Non-Interactive ARgument of Knowledge) protocol is now largely explored for ensuring the privacy. In this section, we will discuss about the ZK-Snark along with introduction to the Zero-Knowledge.

10.1 Zero-Knowledge Proof

The Zero-Knowledge (ZK) proof is used in cryptography to prove the ownership of a secret without sharing the secret to the verifying party. The Zero-Knowledge proof involves two parties: prover and verifier (may be one or more). The Zero-Knowledge proof should satisfy the following properties:

- **Completeness:** If the prover statement is true then an honest verifier will be convinced of the statement.
- **Soundness:** If the prover statement is false, then no cheating prover can convince an honest verifier that it is true, except with some small probability.
- **Zero-knowledge:** If the prover statement is true, then no verifier learns anything other than the fact that the statement is true.

There are various protocols that implements the Zero-Knowledge Proof including Fiat–Shamir Protocol and Blum’s Protocol (1982).

10.2 Interactive Zero-Knowledge Protocol

The ZK Proof can be implemented using the Quadratic residue and this implementation is interactive means there should be multiple communication between the prover and verifier. Let us consider that for an integer N , there is a , $0 \leq a \leq N - 1$ is a quadratic residue $mod N$ if there is an r such that $a \equiv r^2 (mod N)$. With these, now perform the ZK Proof. We have two parties: one prover (Alice) and one verifier (Bob). The parameters that are known to the prover (Alice) are

N, a, r and known to the verifier (Bob) are N and a . Here the secret is r and Alice should prove to Bob that she know the secret r . The communication flow is as follows.

- Alice picks a random $t \in Z_N^*$ and send $y \equiv t^2 \pmod N$ to Bob.
- Bob picks random $b \in \{0, 1\}$ and send b to Alice
- Now, let us assume the case, Bob sends 0
 - Alice sets $z \equiv t \pmod N$ and send to Bob. This means now Alice is simply sending the t .
 - Bob check the equivalence that is received z when squared is equal to y or not. That is, if $z^2 \equiv t \pmod N$ then Bob accepts the proof otherwise reject it.
 - **Important Note: In case $b = 0$, even without knowing the secret Alice can prove to Bob**
- Now, let us assume the case, Bob sends 1
 - Alice sets $z \equiv rt \pmod N$ and send to Bob.
 - Bob check the equivalence that is received z when squared is equal to $y \times a$ or not. That is, if $z^2 \equiv ya \pmod N$ means $(rt)^2 \equiv ya \pmod N$. We know already that $a = r^2$ and $y = t^2$, so $(rt)^2 \equiv ya \pmod N \Rightarrow (rt)^2 \equiv (rt)^2 \pmod N$ so Bob accepts the proof otherwise reject it.
 - **Important Note: In case $b = 1$, Alice should know the secret then only can prove to Bob**

Issues: To ensure the soundness and completeness, Alice and Bob need to go for multiple interactions. In case of $b = 0$ then there is no guarantee that the Alice knows the secret means the probability is half that is 0.5 (referring random) however in case of $b = 1$, it guarantees the secret with Alice. However, if we see the number of interactions between Alice and Bob is more than one. In the Quadratic Residue based ZK-Proof algorithm, even in case of $b = 1$, there is a need for one cycle of interaction between Alice and Bob means Alice has to send the initial t , then Bob will send b and Alice send the proof z to Bob. Hence, minimum three interactions required between Alice and Bob. If Bob choose 0 then five interactions required to ensure the completeness and soundness.

These interactions are possible only when we have small set of verifiers and all are known to them. However, if we want to implement the ZK-Proof in cryptocurrency then it is highly complex because there will be so many verifiers and verifiers may not be known also. In such a scenario the interactive protocol is not applicable which means we cannot implement the quadratic based ZK-Proof in the cryptocurrency application. Hence, the zk-SNARK is preferred and implemented.

10.3 Vanishing Polynomial

The vanishing polynomial means that the polynomial will produce the output as 0 given a value from set. The details are as follows:

- Consider that Ω is a subset or subgroup of F_p where the subgroup size is k .
- Now, the vanishing polynomial is $Z_\Omega[X]$

$$Z_\Omega[X] = \prod_{a \in \Omega} (X - a)$$

degree of $Z_\Omega[X]$ is k . If $a \notin \Omega$ then $Z_\Omega[X] \neq 0$ so will not get vanished. This means if $a \in \Omega$ it will vanish (will produce output 0) otherwise not.

Let us see an example,

$$\text{Let } \mathbb{F}_p = \mathbb{F}_{17} \text{ and the subgroup or subset } \Omega = \{1, 2, 3\}.$$

Then the vanishing polynomial over the set Ω is:

$$Z_\Omega(X) = (X - 1)(X - 2)(X - 3)$$

Expanding it, we get:

$$Z_{\Omega}(X) = X^3 - 6X^2 + 11X - 6$$

Now, evaluate the polynomials based on the subset values that is $\Omega = \{1, 2, 3\}$:

$$Z_{\Omega}(1) = X^3 - 6X^2 + 11X - 6 = 1^3 - 6(1^2) + 11(1) - 6 = 1 - 6 + 11 - 6 = -12 + 12 = 0$$

$$Z_{\Omega}(2) = X^3 - 6X^2 + 11X - 6 = 2^3 - 6(2^2) + 11(2) - 6 = 8 - 24 + 22 - 6 = -30 + 30 = 0$$

$$Z_{\Omega}(3) = X^3 - 6X^2 + 11X - 6 = 3^3 - 6(3^2) + 11(3) - 6 = 27 - 54 + 33 - 6 = -60 + 60 = 0$$

It produces 0 for all the elements of Ω hence $Z_{\Omega}(X)$ is the vanishing polynomial for the value of Ω . For any other value $x \notin \Omega$:

$$Z_{\Omega}(x) \neq 0.$$

For example, we pick $x = 4$ which is not from Ω and it will not produce 0. We can see in the below.

$$Z_{\Omega}(4) = X^3 - 6X^2 + 11X - 6 = 4^3 - 6(4^2) + 11(4) - 6 = 64 - 96 + 44 - 6 = -102 + 108 = 6$$

In all the above computation *mod* 17 should be applied however we have not applied for easy presentation also we get the desired result which is less than 17.

Important Note: $\Omega = \{\omega^0, \omega^1, \omega^2, \dots, \omega^{k-1}\} \subseteq F_p$ (this is the multiplicative subset or subgroup of the finite field F_p^{\times} of size k), where ω is the primitive k -th root of unity means $\omega^k = 1$ then we can conclude that $Z_{\Omega}(X) = X^k - 1$. For example, let us consider that $p = 17$ so F_{17}^{\times} , $k = 4$ and $\omega = 4$ ($\omega^k = 1 = 4^4 = 256 \text{ mod } 17 = 1$) then $Z_{\Omega}(X) = X^k - 1 = Z_{\Omega}(4) = 4^4 - 1 = 256 - 1 = 255 \text{ mod } 17 = 0$

The conclusion is that, if any one given the polynomial $Z_{\Omega}(x)$ and a random element $x \in \Omega$ can compute and check the output is 0 or not. If the $X \notin \Omega$ (X not in Ω) then $Z_{\Omega}(X)$ will not vanish so help in identifying the change. For example, if some one tries to send the fake Ω value or incorrect polynomial instead of the actual polynomial $Z(X)$ then that can be easily identified since the polynomial will not vanish means will not produce the output 0.

10.3.1 Zero Test

Zero Test is one of the proof systems for the vanishing polynomial. It says that the polynomial f will output zero for all roots of unity inputs (that is Ω , we discussed in the previous section) given the vanishing polynomial Z divides f without remainder. We understand clearly from the below.

Let us consider that the prover has the polynomial f of degree d and the verifier has the commitment of f . The verifier verifies that the polynomial will be zero for the entire set Ω (output of the polynomial will be zero provided input from Ω) as discussed in the previous section. First, let us consider the following lemma (A lemma is a small theorem or proven statement whose main purpose is to help prove a bigger theorem.):

Lemma: The polynomial $f(X)$ is zero on all roots Ω (for all values of Ω) if and only if $f(X)$ is divisible by $Z_{\Omega}(X)$. We already know that $Z_{\Omega}(X) = X^k - 1$ and it is vanishing polynomial.

The process of proof is as follows:

- The prover computes a quotient polynomial which is $q(X)$. Now, we can say that $q(X) = \frac{f(X)}{Z_{\Omega}(X)}$. If $f(X)$ is 0 on any value of Ω then the division of $f(X)$ by a vanishing polynomial $Z_{\Omega}(X)$ will produce a quotient polynomial otherwise it may not produce clean polynomial (that may be a rational function with denominator). A rational function is a ratio of two polynomials. It may or may not have the denominator. In our requirement, it should not produce the output with denominator. If it produce with denominator, then it is not the polynomial.
- Now, the prover commits to the polynomial $q \in F_p^{\leq d}(X)$. Ask the verifier to validate the proof by sending the following:
 - The set Ω — the points where the polynomial should vanish.

- The size k of the subset/subgroup.
- The generator (primitive root of unity) ω .
- Receiving q , the verifier has to check whether q is really the quotient of $\frac{f(X)}{Z_\Omega(X)}$ ($q(x) = \frac{f(X)}{Z_\Omega(X)}$) or garbage.
- We can write $q(x) = \frac{f(X)}{Z_\Omega(X)}$ as $f(X) = q(X) \times Z_\Omega(X)$.
- Now to verify it, the verifier choose a random $r \in F_p$ ($r \xleftarrow{\$} \mathbb{F}_{17}$), here the $\$$ refers, the r is chosen uniformly and randomly. Uniform means all outcomes are equally likely and Random means that the choice is not deterministic means you do not know in advance what the value will be. We can also say that the Random: You cannot predict r ahead of time and Uniform: Every element of F_p has exactly the same probability $1/p$ of being chosen.) and send to the prover.
- For example, the occurrence of elements in the set $\{1, 2, 3, 4, 5\}$ are equal. The probability is $1/p$ for all elements, where p is the number of elements. If we see the set $\{1, 2, 2, 2, 5\}$, the occurrence is not equal because the element 2 is having the probability of $3/p = 3/5$. Hence, the $\{1, 2, 2, 2, 5\}$ set is not uniform.
- Now, receiving r , the prover computes $f(X = r) \Rightarrow f(r)$ and $q(X = r) \Rightarrow q(r)$ and send the result back to the verifier. That means $f(r)$ and $q(r)$ are sent to the verifier.
- The verifier can compute $Z_\Omega(r)$ from the prover sent Ω, ω, k and $F_p = F_{17}$.
- Now, verifier will accept if $f(r) = q(r) \times Z_\Omega(r)$ otherwise reject. Based on this $f(X) = q(X) \times Z_\Omega(X)$, $f(r)$ will be equal to $q(r) \times Z_\Omega(r)$ with high probability if no false modification in any computation.
- Here, when $f(r) = q(r) \times Z_\Omega(r)$ means that the $f(r)$ is divisible by the vanishing polynomial $Z_\Omega(r)$ then the verifier can conclude that the f will be zero on the entire set of ω . Hence, achieves the Zero Test.
- The prover gives $f(r)$ and $q(r)$ to the verifier. Now, verifier needs to compute the vanishing polynomial $Z_\Omega(r)$ to verify $f(r) = q(r) \times Z_\Omega(r)$. The computation of $Z_\Omega(r)$ should not be complex and should be efficient otherwise the verifier has to spend more time.

10.4 Commitment Scheme

It is a cryptographic mechanism and we can lock a value secretly so that:

- No one can see what the value is (hiding).
- The person who hided the value cannot change the value later (binding).

We can say this as a digital sealed envelope so that no body can modify the information present inside and no body can see the information inside until it get opened. We can see cryptographically as follows with two phases: commit and verify phase:

- **Commit Phase**

$$(C, r) \leftarrow \text{Commit}(m)$$

- m : the message or value to commit.
- r : a random binding value.
- C : is the commitment and shared with the verifier.
- *Commit*: is the commitment function. We can consider this as the cryptographic hash function.

Why the value r is required?

- a) If r is not there then attacker may try to brute force the hash function for possible message m and try to guess m . However, this is computationally very complex but if the message is small and size is known then possible to guess. Hence with large size r ensures hiding.
 - b) If we use r_1 and r_2 in binding then we can create two commitments for the same message.
 - c) If r is given by the verifier for an example, then prover cannot replace the commitment because different r will produce different commitment and will not match while verification. Let us say verifier gave r_1 for commitment but the prover cheated with r_2 then it will not match during verification because verifier will use r_1 along with message m to check. Hence, prover cannot cheat.
- **Verify Phase:** To reveal the commitment, the committer or prover discloses (m, r) and already C is shared. The verifier checks:

$$\text{Verify}(C, m, r) = \text{true}$$

The verifier takes the m and r , compute the hash (apply the commit function) then it produces C_v ($C_v = h(m, r)$, verifier computed hash). If $C = C_v$ then verification successful otherwise it fails.

Important Note: It is computationally infeasible to find distinct inputs $(m_1, r_1) \neq (m_2, r_2)$ such that:

$$\text{Hash}(m_1 \parallel r_1) = \text{Hash}(m_2 \parallel r_2)$$

assuming the hash function is collision-resistant (means not easy to find two messages that will collide with same hash value).

10.5 Polynomial Commitments

A commitment scheme is a cryptographic primitive that allows prover to commit to a chosen value, however, hidden from verifier, with the ability to reveal the committed value later to prove it; however, prover cannot change it in between means after the commitment till verification. A polynomial commitment allows one to commit to a polynomial $P(x)$ in such a way that:

- Binding: Once a prover commits to $P(x) = x^3 + x + 1$ cannot change it without being detected.
- Hiding: The commitment can hide the coefficients of the polynomial.
- Verification: The prover can prove the value of the polynomial at a specific point $x = a$ (means $P(a) = a^3 + a + 1 = y$) without revealing the entire polynomial, and the verifier can check the proof efficiently. Here, the proof is, given a , the polynomial will produce y and the polynomial is of given degree $\leq d$. The verifier is given the committed polynomial, a , y and degree d . If polynomial is changed that means if violated binding then that changed polynomial cannot produce y for a .

10.5.1 KZG (Kate-Zaverucha-Goldberg) polynomial commitment

The KZG is a polynomial commitment scheme considering three parties: Prover, Verifier and the trusted party (trusted setup). The trusted party create a commitment for a secret and give to the prover and verifier. However, the secret is the toxic waste (means random secret generated during the trusted setup and should be destroyed securely. If someone gets it then they can create fake proof and prove to the system) means after creating the commitment it should be destroyed securely. Let us see the process at all three parties.

Trusted Setup (Trusted Party)

- The trusted party knows the curve parameters (if elliptic curve is used) including the subgroup order, curve, base point(generator g) and p of F_p .

- The secret parameter or the toxic waste s to be computed following the prime field F_p means $s \in F_p$. We consider that $s = 3$
- The secret commitments will be computed by using the generator g that is the subgroup base point of the curve: $m_0 = g^{s^0}, m_1 = g^{s^1}, m_2 = g^{s^2}, m_3 = g^{s^3}, \dots, m_n = g^{s^n}$, where n is the order of the subgroup.
- The toxic waste will be securely destroyed and the commitments ($m = \{m_0, m_1, \dots, m_n\}$) will be shared with both the prover and verifier.
- **Important Note: Here, g^s means it is the scalar multiplication of the point by s . For example, let us say the g is a point of (x, y) coordinate and that is multiplied by s, s^1, s^2 and so on. Here the multiplication is the scalar multiplication including point addition and doubling as we already discussed in the elliptic curve example. In case, we implement this commitment in non-elliptic curve then g is the scalar value.**

Prover

- Prover knows the trusted setup commitments m and the curve parameters including the sub-group order, curve, base point(generator g).
- Create a polynomial for an example $P(x) = 2x^2 + 3x + 1$.
- Assume that the verifier has given an verification element that is a . This verification element is the value for x .
- Evaluate the polynomial by applying a . For example, consider the value of $a = 5$ then $P(5) = 2 \cdot 5^2 + 3 \cdot 5 + 1 = 2 \cdot 25 + 3 \cdot 5 + 1 = 50 + 15 + 1 = 66$. We consider this output as O .
- Now, prover has to compute the commitment. Prover take all the coefficients of the created polynomial ($P(x) = 2x^2 + 3x + 1$) as t means t_1, t_2, \dots . In our example, $t_2 = 2, t_1 = 3, t_0 = 1$ (where $P(x) = 2x^2 + 3x + 1$). We have to take in the reverse order means the highest power in the polynomial equation to be considered at last.
- Now, generate the commitment ($C = g^{P(s)}$): The secret parameter s is not known however the equivalent commitments m are known. Now let us compute the commitment as follows:

$$C = g^{P(s)} = m_0^{t_0} \cdot m_1^{t_1} \cdot m_2^{t_2}$$

- Now, compute the proof π . The π is computed as follows:

$$\pi = g^{Q(x)} \Rightarrow Q(x) = \frac{P(x) - P(a)}{x - a} = \frac{2x^2 + 3x + 1 - 66}{x - 5} = \frac{2x^2 + 3x - 65}{x - 5} = 2x + 13$$

Here in computing $g^{Q(x)}$, the new t values are required. As we computed polynomial $Q(x)$ already, t for this polynomial $2x + 13$ is $t_0 = 13$ and $t_1 = 2$. Now let us compute

$$\pi = g^{Q(x)} = m_0^{t_0} \cdot m_1^{t_1}$$

We can check the following considering the value of a is 5:

$$Q(x) = \frac{P(x) - P(a)}{x - a} \Rightarrow \frac{P(x) - P(5)}{x - 5} \Rightarrow Q(x) \cdot x - 5 = P(x) - P(a)$$

Where $P(x) - P(a)$ is $2x^2 + 3x - 65$ as per the earlier computation of $Q(x)$

$$(2x + 13) \cdot (x - 5) = 2x^2 - 10x + 13x - 65 = 2x^2 + 3x - 65$$

Both matches, so we can write,

$$Q(x) = \frac{P(x) - P(a)}{x - a} \Rightarrow Q(x) \cdot x - 5 = P(x) - P(a)$$

- Prover will send the π and C to the verifier.
- **Important Note: Here, $m_0^{t_0}$ means it is the scalar multiplication of the point by t_0 . For example, let us say the m_0 is a point of (x, y) coordinate and that is multiplied by scalar t_0 . Here the the multiplication is the scalar multiplication including point addition and doubling as we already discussed elliptic curve example.**

Polynomial Property

The $Q(s) = \frac{P(x)-P(a)}{x-a}$ is possible because of the Little Bézout theorem. The Little Bézout theorem states that for any given polynomial $\phi(x) \in Z_p[x]$ and any $a \in Z_p$, then

$$(x - a) \text{ divides } \phi(x) - \phi(a); \text{ where } \frac{\phi(x) - \phi(a)}{(x - a)} \text{ remainder is } 0$$

Hence $Q(s)$ is the quotient polynomial without any variable in the denominator or any negative exponents. Here $\phi \in Z_p[x]$ means $\phi(x)$ is a polynomial whose coefficients come from Z_p . However, x need not to be part of Z_p .

Example using Little Bézout's Theorem: Let us consider the polynomial

$$P(x) = x^3 + 2x^2 + 3x + 4$$

and choose $a = 1$.

Step 1: Compute $P(a)$

$$P(1) = 1^3 + 2 \cdot 1^2 + 3 \cdot 1 + 4 = 10$$

Step 2: From $P(x) - P(a)$

$$P(x) - P(1) = (x^3 + 2x^2 + 3x + 4) - 10 = x^3 + 2x^2 + 3x - 6$$

Step 3: Divide by $x - a = x - 1$ that is $Q(x) = \frac{P(x)-P(1)}{x-1}$. This can also be written as $P(x) - P(1) = (x - 1) \cdot Q(x)$. Here $Q(x)$ is the quotient of $\frac{P(x)-P(1)}{x-1}$.

We want the **quotient polynomial** $Q(x)$ such that

$$P(x) - P(1) = (x - 1) \cdot Q(x)$$

So, let us do the polynomial division,

Polynomial division using synthetic division (short-cut using only the coefficients instead of writing variables) method: Let us take the Coefficients of the dividend polynomial $P(x) - P(1) : x^3 + 2x^2 + 3x - 6 \Rightarrow 1, 2, 3, -6$ and a from divisor polynomial $x - 1$, where $a = 1$.

$$\begin{array}{r|rrrr} 1 & 1 & 2 & 3 & -6 \\ & & 1 & 3 & 6 \\ \hline & 1 & 3 & 6 & 0 \end{array}$$

The process of synthetic division is as above and see in the following:

- Let us write all the coefficients in the first line and keep the $a = 1$ value separately.
- Bring down the first coefficient as it is, means 1 will be brought down as it is.
- Multiply the number just brought down by $a = 1$ that is $1 \cdot 1 = 1$ and add this to the next coefficient that is 2. Now $1 + 2 = 3$ and this has to be written next in the result.
- Repeat the same process for next coefficients. We take the just written 3 and multiply with $a = 1$ so we get 3 and add with next coefficient so $3 + 3 = 6$. Likewise, we do for the next, take 6 multiply with $a = 1$ so we get 6 and add with next coefficient that -6 and we get 0. Now, we write 0 and it is over. We stop once we get 0.

Remainder = 0. Thus the quotient polynomial is

$$Q(x) = x^2 + 3x + 6$$

The quotient polynomial has the maximum degree as 2 because only three terms are there without zero in result. The first term with exponent of 2 (x^2), second term with exponent of 1 ($3x$) and third term with exponent of 0 (6) that means no x in that and only constant 6.

We will do the same division with straight method as follows: Let us take the dividend polynomial $P(x) - P(1) = x^3 + 2x^2 + 3x - 6$ and divisor polynomial is $x - 1$.

- Divide the first term of the dividend by the highest term of the divisor (meaning the one with the highest power of x , which in this case is x since we are considering $x - 1$). Consider the output as the quotient. ($x^3 \div x = x^2$, which is the first term of quotient).
- Multiply x^2 by dividend $x - 1$. We get $x^2(x - 1) = x^3 - x^2$
- Now apply $x^3 - x^2$ to the dividend polynomial for subtraction: Take first two terms ($x^3 + 2x^2$) of the dividend polynomial ($x^3 + 2x^2 + 3x - 6$) and subtract with $x^3 - x^2$. So $(x^3 + 2x^2) - (x^3 + x^2) = 3x^2$ (because $x^3 - x^3 = 0$ and $+2x^2 - (-x^2) = 2x^2 + x^2 = 3x^2$). Now the remainder polynomial is $3x^2 + 3x - 6$
- We have to repeat this process: Take the first term ($3x^2$) of the remainder polynomial ($3x^2 + 3x - 6$) and divide by the first term (x) of divisor that is $x - 1$. So, $3x^2 \div x = 3x$. Consider the output ($3x$) as the quotient. ($3x^2 \div x = 3x$, which is the second term of quotient, so now the quotient is $x^2 + 3x$)
- Now multiply $3x$ with the divisor that is $3x \cdot (x - 1) = 3x^2 - 3x$. We subtract the first two terms ($3x^2 + 3x$) of remainder dividend polynomial with this $3x^2 - 3x$. So, $3x^2 + 3x - (3x^2 - 3x) = 6x$ (because $3x^2 - 3x^2 = 0$ and $+3x - (-3x) = 3x + 3x = 6x$). Now the remainder polynomial is $6x - 6$ (including the pending term -6).
- We have to repeat this process: Let us take the first term ($6x$) of the remainder polynomial ($6x - 6$) and divide by the first term (x) of divisor that is $x - 1$. So, $6x \div x = 6$. Consider the output (6) as the quotient. ($6x \div x = 6$, which is the third term of quotient, so now the quotient is $x^2 + 3x + 6$)
- Now multiply 6 with the divisor that is $6 \cdot (x - 1) = 6x - 6$. We subtract the first two terms (or we can say the remaining terms because we have only two terms) $6x - 6$ of remainder polynomial with this $6x - 6$. So, $(6x - 6) - (6x - 6) = 0$ (because $6x - 6x = 0$ and $-6 - (-6) = -6 + 6 = 0$). Now there is no remainder hence we stop here.
- Finally, we got the quotient that is $x^2 + 3x + 6$ without any remainder (Remainder is 0). Hence, the $Q(x) = x^2 + 3x + 6$

Step 4: Verify Bézout's Theorem

We know already that

$$P(x) - P(1) = (x^3 + 2x^2 + 3x + 4) - 10 = x^3 + 2x^2 + 3x - 6$$

Now, by rewriting the equation $Q(x) = \frac{P(x)-P(1)}{x-1}$ as follows, we can say it is equal.

$$P(x) - P(1) = (x - 1)Q(x) = (x - 1)(x^2 + 3x + 6) = x^3 + 2x^2 + 3x - 6$$

Verifier

- Verifier knows the trusted setup commitments m and the curve parameters including the sub-group order, curve, base point (generator g). In addition, he/she knows the $P(a)$, π and C given by the prover.
- Now, the verifier will check the following with the help of bilinear pairing.

$$e(C, g) \stackrel{?}{=} e(\pi, g^s/g^a) \cdot e(g, g)^{P(a)}$$

Here, g^s is known to verifier because that come from the trusted party. Also, we can consider that a is known to verifier as a verifying component or it is generated and given by the verifier to the prover so verifier can compute g^a .

- Proof of Correctness: First let us take the LHS that is $e(C, g)$, where $C = g^{P(s)}$. Also, we can write $P(s) = Q(x)(s - a) + P(a)$ because $Q(x) = \frac{P(x)-P(a)}{x-a}$ and assume $x = s$ then $P(s) = P(x) = Q(x) \cdot (x - a) + P(a)$ or $P(s) = Q(s) \cdot (s - a) + P(a)$ hence

$$e(C, g) = e(g^{P(s)}, g) = e(g, g)^{P(s)} = e(g, g)^{Q(x)(s-a)+P(a)}$$

- We can write, $\frac{g^s}{g^a}$, as g^{s-a} if the base in both numerator and denominator are same and here it is g . For example, if $\frac{2^1}{2^3}$ then we can write as 2^{1-3} . so, we can write $e(\pi, g^s/g^a)$ as $e(\pi, g^{s-a})$.
- In case of RHS, $e(\pi, g^{s-a})$.

$$e(\pi, g^s/g^a) \cdot e(g, g)^{P(a)} = e(g^{Q(x)}, g^s/g^a) \cdot e(g, g)^{P(a)} = e(g^{Q(x)}, g^{s-a}) \cdot e(g, g)^{P(a)}$$

By applying the bilinearity, let us bring as follows

$$\begin{aligned} &= e(g, g)^{Q(x)(s-a)} \cdot e(g, g)^{P(a)} \\ &= e(g, g)^{Q(x)(s-a)+P(a)} \end{aligned}$$

Hence, LHS is equal to RHS

$$e(g, g)^{Q(x)(s-a)+P(a)} = e(g, g)^{Q(x)(s-a)+P(a)}$$

- If the equality holds that mean the verification is successful otherwise the proof will be rejected by the verifier.
- **The summary is, the prover wants to prove that he/she has a polynomial $P(x) = 2x^2 + 3x + 1$ and verifier wants to verify that the polynomial $P(x)$ evaluate to a certain value at a particular point a . That is getting achieved in the process.**

This polynomial commitment is not Zero-Knowledge because $g^{P(x)}$ is a deterministic function anyone knowing the C can map with different polynomial function. That is by having C , some one will try with possible polynomials whether it matches or not. If it matches then the polynomial is disclosed hence no zero-knowledge. It will be easy to get the matching polynomial if the field will be small that means p in Fp is small.

We can achieve the zero-knowledge by mixing the random r in C as $g^{P(s)} \cdot h^r$ instead of computing C as $g^{P(s)}$. Here, r is the random number and h is a function like g^i for another secret i (here r is that secret, so we can consider like $g^{P(s)} \cdot g^r$). Without knowing that random number, it is computationally complex to bring the matching polynomial.

10.6 Schwartz–Zippel Lemma

A non-zero polynomial has at least one non-zero coefficient in it. For example,

$$f(x) = 3x^2 + 1$$

is a non-zero polynomial (with 3 and 1 as coefficients) because not having all coefficients as 0. In contrast, the zero polynomial is the one where every coefficient is zero, so it is equal to zero for all values/inputs of the variable.

Even though, we say that a non-zero polynomial not produce 0 but there are possibilities that it can produce 0 as output for a small set of inputs. A non-zero polynomial can evaluate to 0 at some inputs that is for its roots (which is used to create the respective polynomial) as we have discussed earlier.

For example, let us consider that there is a finite set S and the values of this set will be applied to the value of x 's in the non-zero polynomial $P(x_1, x_2, x_3 \dots x_n)$ with degree d . Here, P is multi-variate polynomial (with more than one variable on the right hand side of the polynomial equation [eg. $P(x_1, x_2) = x_1^2 + x_1 \cdot x_2 + 2$]). In case, we want to have the uni-variate polynomial then it is only $P(x)$ where there will be only one variable on the right hand side [eg. $P(x) = x^2 + 1$].

The probability that the non-zero polynomial evaluates to zero for the inputs of S is atmost $\frac{d}{|S|}$. We can write as follows:

$$\Pr[P(r_1, \dots, r_n) = 0] \leq \frac{d}{|S|}$$

$r_1, r_2, r_3 \dots r_n \in S$ where r_1, r_2, \dots, r_n are selected at random independently and uniformly from S . Here independent refers that the r_2 is not chosen based on any other r means not having the

relationship. Uniform means that the occurrence of all r is equal for example in a set $\{1, 2, 3, 4, 5\}$, every number will be chosen uniformly with equal probability because chance of picking all 1,2,3,4 and 5 is $\frac{1}{5}$. If we take a set $\{1, 2, 2, 2, 5\}$ then probability of picking 1 is $\frac{1}{5}$ and 2 is $\frac{3}{5}$ so not uniform. In case of the uni-variate polynomial, randomly and uniformly one element r will be chosen.

When we try with inputs r in S then atmost for its roots d numbers, the polynomial equation will be evaluated to 0. So the overall chance of getting or evaluating the non-zero polynomial to 0 is $\leq \frac{d}{|S|}$ which is very less when we consider the larger set.

The Schwartz–Zippel Lemma states that the probability of a non-zero polynomial producing output 0 is $\leq \frac{d}{|S|}$.

Example:

Let us consider the polynomial,

$$P(x) = (x - 1)(x - 2) = x^2 - 3x + 2$$

and the degree is

$$d = 2.$$

Let

$$S = \{1, 2, 3, 4, 5\}$$

so the size is

$$|S| = 5.$$

The actual number of roots in S is 2 that is 1 and 2 (the number of roots equal to the degree). The fraction of inputs giving 0 is

$$\frac{2}{5}.$$

This matches the upper bound given by the Schwartz–Zippel lemma:

$$\Pr[P(r) = 0] \leq \frac{d}{|S|} = \frac{2}{5}$$

But consider the following polynomial with degree 2 and without root over the same set S :

$$Q(x) = x^2 + 1.$$

The actual fraction (we use the term fraction not the count or number of inputs because we do not get the exact integer when we compute probability so we say fraction since the lemma is based on probability) of inputs giving 0 is

$$0,$$

but the lemma still gives the upper bound:

$$\Pr[Q(r) = 0] \leq \frac{d}{|S|} = \frac{2}{5}$$

Thus, the Schwartz–Zippel lemma provides an upper bound, not an exact count of inputs that can produce zeros.

The zero and non-zero polynomial can also be called as identically zero and not identically zero respectively.

- A polynomial is identically zero if it equals or produces zero for every possible input.
- A polynomial is not identically zero if there is at least one input where it does not equal or produce zero.

Why Schwartz–Zippel lemma? In polynomial commitment schemes, the prover commits to a polynomial $P(x)$ and the verifier has to check it. The verification has to be performed by checking with all possible inputs for x and accepts if all inputs evaluate to 0. In general, verifier cannot check all points because the polynomial could be huge with many possible inputs (large set of inputs), so the verifier picks a random point r from the set and asks the prover to prove the polynomial's evaluation at r . The proof will be verified by the verifier on that point only accordingly verifier accepts or reject the proof. However, there is a following risk:

- The prover might try to cheat by using a different polynomial $Q(x) \neq P(x)$ that matches $P(x)$ at r and verifier may accept. **Issue: If the verifier checks and accepts based on only one input r , then the verifier accepts $Q(x)$ in place of $P(x)$ since both at the point r produce the output 0.**

The Schwartz-Zippel lemma guarantees that

- If $Q(x) - P(x)$ is a non-zero polynomial of degree d , then the probability that it evaluates to zero at a randomly chosen point r from a set S is bounded by

$$\Pr[Q(r) - P(r) = 0] \leq \frac{d}{|S|}.$$

It says that the probability of two different polynomials $P(x)$ and $Q(x)$ agreeing on a randomly chosen point can be made arbitrary low (very low) by choosing a sufficiently large set S .

The reason for mentioning $Q(x) - P(x)$ is that the verifier normally checks that whether $Q(x) = P(x)$ or not accordingly accept. In that case, we can say that $Q(x) - P(x) = 0$, if both $Q(x)$ and $P(x)$ are equal. Hence, mentioned as $Q(x) - P(x)$ otherwise, we can write as $Q(x)$ that may be equal to $P(x)$ or not equal to $P(x)$.

Conclusion: When the prover sends the commitments to the verifier, the verifier verifies it based on random point r . There is a possibility that even after the committed polynomial $P(x)$ modified to $Q(x)$ can produce the same output as $P(x)$ which is zero, the verifier accepts it. This is possible since two different polynomials can evaluate to zero (produce zero) at the same point r . However, as per the Schwartz-Zippel lemma, the probability of such occurrence is $\frac{d}{|S|}$, which is very low. Hence, the chance of cheating is very low. With the guarantee of this lemma, verifier can check with only one point instead of all points or inputs. This helps to ensure the succinctness in the applications (such as zk-SNARK), which implement the polynomial commitment scheme that is instead of checking with all possible inputs checking at one random point or input (r) is sufficient thus lightweight or less cost in verification.

10.6.1 Product Check

The prover has the polynomial and the verifier get the commitment of the polynomial to verify that the product of the polynomial with all values of a is 1. The polynomial that satisfies the zero test will not satisfy the product check since the zero test satisfied polynomial produce 0 as output for any values of a . If we take such polynomial then it fails the product check. Hence, we have to chose a polynomial f that satisfies the following condition to satisfy the product check:

$$\prod_{a \in \Omega} f(a) = 1 \text{ Here, } \prod_{a \in \Omega} f(a) \text{ is } f(1) \cdot f(\omega) \cdot f(\omega^2) \cdot \dots \cdot f(\omega^{k-1})$$

where

$$f(\omega^0) = f(1) \text{ and } f(\omega^1) = f(\omega)$$

so

$$a = \{\omega^0, \omega^1, \omega^2, \dots, \omega^{k-1}\}$$

To check this, let the prover interpolates the polynomial $t \in F_p^{\leq k}[X]$ with degree k . We can say that $t(1) = f(1)$ and $t(\omega^s) = \prod_{i=1}^s f(\omega^i)$, where $s = 1, 2, \dots, k-1$. We can interpolate the auxiliary

polynomial t in the following. Auxiliary polynomial is a polynomial constructed in addition to the main one, used to simplify a proof or derive contradictions. Instead of asking the verifier to compute all values of $f(a)$ (impossible in a commitment system), the prover constructs an auxiliary polynomial:

$$t(1) = f(1)$$

$$t(\omega) = f(1).f(\omega)$$

$$t(\omega^2) = f(1).f(\omega).f(\omega^2)$$

$$t(\omega^{k-1}) = \prod_{a \in \Omega} f(a) = 1$$

If the product is 1 then there is a recurrence relation (A recurrence relation is an equation that defines each term of a sequence using previous terms of the same sequence). We can see a pattern in the above that is each next value of t is the previous value multiplied by the next value of f . Hence,

$$t(\omega.x) = t(x).f(\omega.x) \text{ for all } x \in \Omega \text{ including } x = \omega^{k-1}$$

Here, consider x is the previous value for t and $\omega \cdot x$ is the new value so it holds the previous value output for $t(\omega \cdot x)$ that is $t(x).f(\omega.x)$. For more clarity, it is like the following.

$$t(x) = f(1).f(x)$$

$$t(\omega \cdot x) = f(1) \cdot f(x) \cdot f(\omega \cdot x) = t(x) \cdot f(\omega \cdot x)$$

Now, let us consider the following lemmas based on the above

- $t(\omega^{k-1}) = 1$
- $t(\omega.x) - (t(x).f(\omega.x)) = 0$ for all $x \in \Omega$ then $\prod_{a \in \Omega} f(a) = 1$

Here, the prover has to prove to the verifier that the above two lemma holds. Let us see these in the following:

- At first, the prover computes $t(X) \in F_p^{(\leq k)}$ and $t_1(X) = t(\omega.X) - t(X).f(\omega.X)$ and $t_1(X)$ should be zero on all elements of Ω and that has to be proved.
- The prover computes the quotient polynomial $q(x) = t_1(X)/(X^k - 1) \in F_p^{(\leq k)}$. We have already seen in vanishing polynomial section that the $Z_\Omega(X) = X^k - 1$.
- Now t and q along with public parameters like Ω are sent to the verifier.
- The verifier will compute the random $r \in F_p$ ($r \xleftarrow{\$} \mathbb{F}_{17}$) and send to the prover and ask the following:
 - $t(\omega^{k-1})$. The Ω containing ω^{k-1} is sent to the verifier already.
 - $t(r)$
 - $t(\omega \cdot r)$
 - $q(r)$
 - $f(\omega \cdot r)$
- The prover computes all the above and send to the verifier.
- Now, the verifier checks the equality holds or not to accept according to the given two lemmas:

$$t(\omega^{k-1}) \stackrel{?}{=} 1$$

$$t(\omega \cdot r) - t(r)f(\omega \cdot r) \stackrel{?}{=} q(r).(r^k - 1)$$

- If the equality holds, the verifier accepts the proof otherwise reject it.
- In case of $t(\omega^{k-1})$, the prover will send the output as 1 and that will be equal to 1 always. But how the verifier will come to know whether it is computed or not. We will see in the following how the verifier can verify it.
 - The prover will send the output of $t(\omega^{k-1})$ that is 1
 - Also, send the commitment opening proof π .
 - Using, the commitment opening proof it will be checked whether it equals 1 or not.
- If we go for the KZG polynomial commitment, we can do the the following to verify it:
 - Let $P(X) = t(X)$ be a polynomial over a finite field \mathbb{F}_p and the commitment is

$C_t = g^{t(s)}$; Refer the 10.5.1 for knowing how to commit the polynomial using the secret s

where:

- * g is a generator of a pairing-friendly group G_1 ,
- * s is the secret trapdoor in the beginning that come from the trusted set up. Actually s will not be given directly to the prover since it is the secret. Refer the trusted setup in the section 10.5.1 of KZG to understand how the commitment of s will be generated and provided. The verifier already received $m = g^s$ set from the trusted part.
- * $t(s)$ is the evaluation of the polynomial at s . Again this cannot be computed directly since s is not known to the prover. Refer the trusted setup in the section 10.5.1 for computing the $g^{t(s)}$
- To prove that

$$t(\omega^{k-1}) = 1,$$

the prover computes a quotient polynomial $q(X)$ such that:

$$t(X) - 1 = q(X) \cdot (X - \omega^{k-1})$$

and sends the opening proof:

$$\pi = g^{q(s)}$$

Here, we use the term s in place of X since the proof is committed using the secret s . You may refer section 10.5.1, how to compute this $g^{q(s)}$.

- The verification is as follows based on pairing:
The verifier checks the equality:

$$e(C_t - 1 \cdot g_1, g_2) \stackrel{?}{=} e(\pi, s - \omega^{k-1} g_2)$$

Important Note: When it is written s , it is g^{s^1} given by the trusted setup to both the verifier and prover.

LHS:

Using bilinear property, we can write

$$e(C_t - 1 \cdot g_1, g_2) = e(g_1^{C_t-1}, g_2)$$

RHS:

$$e(\pi, s - \omega^{k-1} g_2) = e(g^{q(s)}, (s - \omega^{k-1})g_2)$$

Using bilinearity of the pairing:

$$e\left(g^{q(s)}, (s - \omega^{k-1})g_2\right) = e(g^{q(s)}, g_2)^{s - \omega^{k-1}} = e\left(g^{q(s)(s - \omega^{k-1})}, g_2\right)$$

In the above, we are moving the product or exponent $(s - \omega^{k-1})$ of group G_2 to G_1 . It allows in the pairing that exponents can be swapped or moved whether $G_1 = G_2$ or $G_1 \neq G_2$. Now recall that the quotient polynomial satisfies

$$t(s) - 1 = q(s)(s - \omega^{k-1})$$

Substituting this identity gives:

$$e\left(g^{q(s)(s - \omega^{k-1})}, g_2\right) = e\left(g^{t(s)-1}, g_2\right)$$

We know that the $C_t = g^{t(s)}$, therefore,

$$e\left(g^{t(s)-1}, g_2\right) = e(g_1^{C_t-1}, g_2)$$

Hence LHS is equal to RHS.

where:

- * C_t is the polynomial commitment,
- * π is the opening proof,
- * $t(\omega^{k-1}) = 1$ is the claimed evaluation,
- * g_1, g_2 are public generators of Group 1 and 2 (may be $G_1 = G_2$ or $G_1 \neq G_2$).

Hence LHS is equal to RHS.

- If the equality holds, the verifier is convinced that the committed polynomial really evaluates to 1 at ω^{k-1} .

Important Note: In all our discussion that the random r is given by the verifier therefore it is interactive. To make it Non-Interactive, we can use the Fiat-Shamir's Transformation. We discuss the Fiat-Shamir's Transformation in section 10.12.5.

10.7 zk-SNARK

The zk-SNARK (zero-knowledge Succinct Non-Interactive ARGument of Knowledge) is the non-interactive protocol ensuring the following properties in addition to the soundness and completeness.

- Zero-knowledge: A verifier does not learn anything beyond the fact that the statement is true.
- Succinct: The size of the proof needs to be small enough to be verified in milliseconds. This is referring to lightweight.
- Non-Interactive: The prover sends only one set of information to the verifier, no further back and forth communication between the prover and verifier.
- ARGument: A dishonest or malicious prover cannot generate a fake proof (for a false statement) in polynomial time, unless they can break a hard computational assumption (like factoring, discrete log, or elliptic curve hardness). [This is like Prevents lying about false statements means trying to prove a false statement]
- Knowledge: The prover cannot generate the proof without the witness (the private input needed to prove the statement). [This property prevents faking a proof without the secret.]

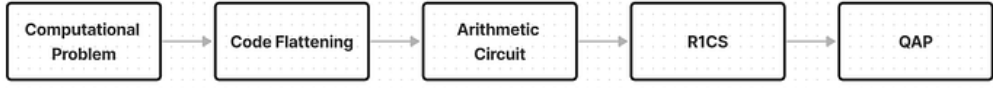


Figure 14: zk-SNARK process [Copied]

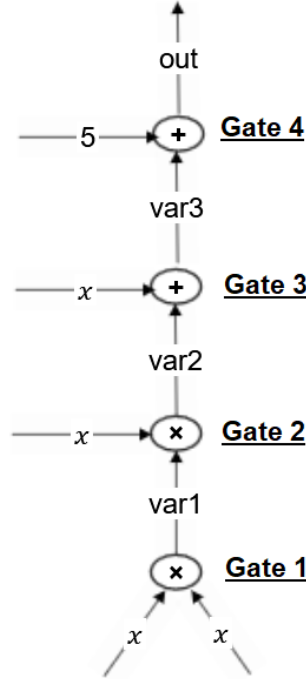


Figure 15: Arithmetic circuit [Copied from Risen Crypto]

10.8 Groth-16 Implementation Process

There are multiple implementations for the zk-SNARK that exist, including **Groth16** (2016) and **Halo** / **Halo2**. In this discussion, we will discuss **Groth16** by Jens Groth. The process of Zk-SNARK is shown in Figure 14, which includes code flattening, Rank-1 Constraint System (R1CS), and Quadratic Arithmetic Polynomial. We see the process of zk-SNARK with an example.

Let us consider the following equation 35 as the input along with the witness 3.

$$x^3 + x + 5 = 35 \tag{35}$$

- Step 1: As a first step, the above equation needs to be flattened, that means we convert the equation to the Three Address Code (TAC) in quadruple form, including assignment (=), operator (*, +), variables, constants. The following is the flattening of the equation 35.

$$\begin{aligned} var1 &= x * x \\ var2 &= var1 * x \\ var3 &= var2 + x \\ out &= var3 + 5 \end{aligned}$$

- Step 2: Next, we can present the flattened code as the arithmetic circuit. Every operation here refers to the Gate. In the taken example equation, we have total 04 gates (with multiplication and additional operation), 5 variables and one constant (5). We can visualize it from Figure 15.
- Step 3: Using the flattened code and circuit diagram, we can create the following solution vector.

$$S = [1, out, x, var1, var2, var3]$$

where, $x = 3$, $var1 = x \times x = 3 \times 3 = 9$, $var2 = var1 \times 3 = 9 = times3 = 27$, $var3 = var2 + 3 = 27 + 3 = 30$ and $out = var3 + 5 = 30 + 5 = 35$. In S , 1 refers the constant to be included for example the 5 in our original equation will be embedded with that at 1×5 . Hence, the following

$$S = [1, 35, 3, 9, 27, 30]$$

- Step 4: Now, we can convert each gate into the form of vector by including all parameters of the flattened code: $var1, var2, var3, constant, output, variable$. We can have the order as solution equation. When we take a gate then we can generate three vectors including two arguments [l and r] and the final output or intermediate output that is the variable. The following is vectors of the Gate 1.

$$\begin{aligned} l &= [0, 0, 1, 0, 0, 0] \\ r &= [0, 0, 1, 0, 0, 0] \\ o &= [0, 0, 0, 1, 0, 0] \end{aligned} \tag{36}$$

Let us match the solution vector with the value in equation 36. If we take the left side variable of the Gate 1, then we can see the mapping as follows, where NA is not applicable in the Gate 1.

$$\begin{aligned} l &= [1, out, x, var1, var2, var3] \\ l &= [NA, NA, x, NA, NA, NA] \\ l &= [0, 0, 1, 0, 0, 0] \end{aligned} \tag{37}$$

$$\begin{aligned} r &= [1, out, x, var1, var2, var3] \\ r &= [NA, NA, x, NA, NA, NA] \\ r &= [0, 0, 1, 0, 0, 0] \end{aligned} \tag{38}$$

$$\begin{aligned} O &= [1, out, x, var1, var2, var3] \\ 0 &= [NA, NA, NA, var1, NA, NA] \\ 0 &= [0, 0, 0, 1, 0, 0] \end{aligned} \tag{39}$$

- Step 5: Any solution vector which is correct will satisfy $(l \cdot S) * (r \cdot S) = (o \cdot S)$. In this case,

$$\begin{aligned} (l = [0, 0, 1, 0, 0, 0]) \cdot (s = [1, 35, 3, 9, 27, 30]) & * (r = [0, 0, 1, 0, 0, 0]) \cdot (s = [1, 35, 3, 9, 27, 30]) = \\ & ([0, 0, 3, 0, 0, 0]) * ([0, 0, 3, 0, 0, 0]) \Rightarrow \text{Gate 1 Output} = [0, 0, 9, 0, 0, 0] \\ (s = [1, 35, 3, 9, 27, 30]) & * (o = [0, 0, 0, 1, 0, 0]) \Rightarrow \text{Output} = [0, 0, 9, 0, 0, 0] \end{aligned}$$

Hence, ensures the $(l \cdot S) * (r \cdot S) = (o \cdot S)$.

- Step 6: Now, let us try for the Gate 3 which is having addition. While doing the addition, the process includes multiplication also that can be seen from the following. So, the addition in Gate 3 is considered as $var2 + x * 1$ and accordingly compute the l with $var2 + x$, r with 1 and o .

$$\begin{aligned} l &= [1, out, x, var1, var2, var3] \\ l &= [NA, NA, x, NA, var2, NA] \\ l &= [0, 0, 1, 0, 1, 0] \end{aligned} \tag{40}$$

$$\begin{aligned} r &= [1, out, x, var1, var2, var3] \\ r &= [1, NA, NA, NA, NA, NA] \\ r &= [1, 0, 0, 0, 0, 0] \end{aligned} \tag{41}$$

$$\begin{aligned}
o &= [1, out, x, var1, var2, var3] \\
0 &= [NA, NA, NA, NA, NA, var3] \\
0 &= [0, 0, 0, 0, 0, 1]
\end{aligned} \tag{42}$$

- Step 7: Let us compute for the Gate 4, which is the final gate producing the final output.

$$\begin{aligned}
l &= [1, out, x, var1, var2, var3] \\
l &= [NA, NA, x, NA, NA, var3] \\
l &= [5, 0, 0, 0, 0, 1]
\end{aligned} \tag{43}$$

$$\begin{aligned}
r &= [1, out, x, var1, var2, var3] \\
r &= [1, NA, NA, NA, NA, NA] \\
r &= [1, 0, 0, 0, 0, 0]
\end{aligned} \tag{44}$$

$$\begin{aligned}
o &= [1, out, x, var1, var2, var3] \\
0 &= [NA, out, NA, NA, NA, NA] \\
0 &= [0, 1, 0, 0, 0, 0]
\end{aligned} \tag{45}$$

- Step 8: Now we can create the R1CS (Rank 1 Constrained System) as follows using the computed l , r and s of all gates. We will be getting 3 R1CS matrix respectively for the left input, right input and the intermediate output or final output.

$$L = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad O = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The reason for using the R1CS matrix is that it is better representation to bridge the program logic to the polynomial equation. We can present every operation (add, multiply) in a single rank-1 constraint thus ensures the compact representation. It is also efficient for storage and evaluation thus helps to succint.

- Step 8: Now, we construct the polynomial from the R1CS matrix to bring QAP (Quadratic Arithmetic Program) form. We take each column from the three matrix and form the polynomial equation.
- Step 10: In this case, we have only one cartesian value may be x or y if we take it from the matrix column but not the polynomial equation. Therefore we need the lagranges interpolation to create the polynomial. Given $n + 1$ points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ with all x_i distinct, the Lagrange interpolation gives the polynomial $P(x)$. This can make the polynomial that can touch The way of computing the polynomial using the lagranges interpolation and co-ordinates is shown in the following:

$$P(x) = \sum_{i=0}^n y_i \cdot \ell_i(x),$$

where the Lagrange basis polynomials $\ell_i(x)$ are defined as follows:

$$\ell_i(x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

We can see from the equation that when $j \neq i$ then the respective computation will not be applied because if $i = j$ then it is going to refer the same x ($x_i = x_j$) value and become 0 ($x_i - x_j$) hence we get infinity therefore we should ignore the $\frac{x-x_j}{x_i-x_j}$ if $i = j$. For example, given 3 points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, the polynomial is

$$P(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

- Step 11: The number of polynomials and each polynomial degree depends on the length or number of elements of the solution vector and the number of gates used in the circuit diagram. The solution vector contains 6 elements, so we can construct 6 polynomials. As we have seen, each gate contributes 1 point to each polynomial. We have 4 gates here, we get 4 points per polynomial that means in total 6 polynomial of maximum degree 3. If we see in the R1CS matrix we have 6 columns and four rows for each L, R and O . So, we are going to use each column for creating a polynomial using the lagranges interpolation therefor we will get 6 polynomials with maximum three degree. This has to be done for all L, R and O so finally we will be getting 18 polynomial equations.
- Step 12: Now, let us take the first column of l which is 0,0,0,5 and consider for cartesian value y . The $x \in 1, 2, 3, 4$ however We can take other values for $x \in 1, 7, 9, 11$ but to be consistent across all computations. Now the co-ordinates are $(1, 0), (2, 0), (3, 0), (4, 5)$. As we know that given $n + 1$ co-ordinates, the lagranges interpolation will generate n^{th} degree polynomial equation. Hence, we will be getting the 3^{rd} degree polynomial.

1: Lagrange Basis Polynomials

For 4 points, the Lagrange basis polynomials are:

$$\begin{aligned} \ell_0(x) &= \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}, & \ell_1(x) &= \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)}, \\ \ell_2(x) &= \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)}, & \ell_3(x) &= \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)}. \end{aligned}$$

2: Plug in y_i values to compute polynomial $P(x)$

$$P(x) = 0 \cdot \ell_0(x) + 0 \cdot \ell_1(x) + 0 \cdot \ell_2(x) + 5 \cdot \ell_3(x)$$

$$P(x) = 5 \cdot \ell_3(x) = 5 \cdot \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)}$$

3: Simplify Denominator

$$(4-1)(4-2)(4-3) = 3 \cdot 2 \cdot 1 = 6$$

$$P(x) = \frac{5}{6}(x-1)(x-2)(x-3)$$

4: Expand the polynomial

$$(x-1)(x-2)(x-3) = (x-1)((x-2)(x-3)) = (x-1)(x^2 - 5x + 6)$$

$$= x(x^2 - 5x + 6) - 1(x^2 - 5x + 6) = x^3 - 5x^2 + 6x - x^2 + 5x - 6 = x^3 - 6x^2 + 11x - 6$$

Multiply by $\frac{5}{6}$:

$$P(x) = \frac{5}{6}x^3 - 5x^2 + \frac{55}{6}x - 5$$

5: Final Lagrange Interpolation Polynomial

$$P(x) = \frac{5}{6}x^3 - 5x^2 + \frac{55}{6}x - 5$$

The final equation is having the fraction and we cannot use like this in our implementation. Hence we will compute the inverse. Here for 6, we will compute the inverse ($6 \times j \equiv 1 \pmod{n}$), where j is the inverse and $p = 641$ as considered in <https://risencryptio.github.io>) as we discussed earlier using the Extended Euclidian Algorithm. Hence, the final equation is as follows:

$$P(x) = 5(6)^{-1}x^3 - 5x^2 + 55(6)^{-1}x - 5$$

$$P(x) \equiv 535x^3 + 636x^2 + 116x + 636 \pmod{641}$$

- Step 13: After the lagranges interpolation, we will be getting 18 polynomials, where $L(x) = L_i(x) = L_1(x), L_2(x) \dots L_6(x)$ contains 6 polynomial equations, $R(x) = R_i(x) = R_1(x), R_2(x) \dots R_6(x)$ contains 6 polynomial equations and $O(x) = O_i(x) = O_1(x), O_2(x) \dots O_6(x)$ contains 6 polynomial equations. Now the computed polynomials can be represented in the form of matrix. While forming the matrix, we should consider the first term coefficient of the polynomial as the last column element. The L_m that is only the $L(x)$'s coefficients is as follows.

$$L_m = \begin{bmatrix} 636 & 116 & 636 & 535 \\ 0 & 0 & 0 & 0 \\ 8 & 416 & 5 & 213 \\ 635 & 330 & 637 & 321 \\ 4 & 634 & 324 & 320 \\ 640 & 536 & 640 & 107 \end{bmatrix}$$

For example, the first row of L_m is based on $P(x) \equiv 535x^3 + 636x^2 + 116x + 636 \pmod{641}$. The reason for taking 641 is that our field order that is $F_p = F_{641}$.

- Step 14: Likewise we will create for the R_m from $R(x)$ and O_m from $O(x)$. Now the QAP is ready.
- Step 15: Next, we do the dot product of each $L(x)$, $R(x)$ and $O(x)$ with the solution vector S . We know that the S is also having 6 elements. The output will be a polynomial means will be getting three polynomial one for each $L(x)$, $R(x)$ and $O(x)$. Let us name these output polynomials as L_x , R_x and O_x . **Important Note: Here, we are attaching the witness S into the equation means we are creating proof including the witness**
- Dot Product: We can also say this as the Scalar product. Let us say, given two vectors to the dot product function we will be getting the scalar output not the vector output. For example,

$$\mathbf{a} = [2, 3, 4] \quad \text{and} \quad \mathbf{b} = [5, 6, 2],$$

then we compute the dot product by transposing b or a . The dot product needs one input to be row vector and another to be column vector. The transpose refers to the row vector to the column vector or column vector to the row vector.

$$\mathbf{a}^T \cdot \mathbf{b} = (2)(5) + (3)(6) + (4)(2) = 10 + 18 + 8 = 36$$

In case, the dot product to be performed on the vector and a matrix then no transpose required since matrix has both the column and row vector since satisfy the requirement.

- Step 16: As of now, we go three polynomials that is L_x , R_x and O_x . Now let us reduce it to one polynomial by performing the following operation. We take the dot product on the L_x , R_x and O_x with solution vector S and compute another polynomial T .

$$T = (S \cdot L_x) * (\cdot R_x) - \cdot O_x$$

- Step 17: We know very well that $(l \cdot S) * (r \cdot S) = (o \cdot S)$ from the earlier discussion in Step 5. We can also say that $(l \cdot S) * (r \cdot S) - (o \cdot S) = 0$ [this is like $2 + 3 = 5$ and $2 + 3 - 5 = 0$]. Hence T polynomial will be zero, if we give the value of x to the polynomial as $x = 1, x = 2, x = 3, x = 4$ not for others. The reason it will work for only these values ($x = 1, x = 2, x = 3, x = 4$), we computed the polynomial through lagranges interpolation using these values as x in Step 12. If T is zero at $x = 1, x = 2, x = 3, x = 4$ then it means these are all roots of T and that should be because the polynomial equations were constructed base on these values 1, 2, 3, 4 for x .
- Step 18: Now, let us create a new polynomial that will be known to both the verifier and prover. The polynomial Z is as follows.

$$Z = (x - 1)(x - 2)(x - 3)(x - 4) = x^4 - 10x^3 + 35x^2 - 50x + 24$$

If T is divided by Z , then it will be perfectly divisible and will leave no remainder since the roots are $x = 1, x = 2, x = 3, x = 4$ and it is part of Z . Hence, we can say that Z is vanishing polynomial since it is going to vanish in all its point. For example, if we take $x = 1$, then $Z(1) - (1-1)(1-2)(1-3)(1-4) = 0 \cdot (-1) \cdot (-2) \cdot (-3) = 0$, hence it vanishes and same for 2, 3 and 4. Also, we can say there must exist a polynomial $H(x)$ such that $T(x) = H(x) \cdot Z(x)$ because $H(x) = \frac{T(x)}{Z(x)}$. Here $H(x)$ will be zero if $T(x)$ divides $Z(x)$ otherwise not. This means, if the correct $T(x)$ and $Z(x)$ will be given as input then $H(x)$ will be zero otherwise non-zero. The polynomial equation $H(x) = \frac{T(x)}{Z(x)} = 139x^2 + 1762x + 13030$ can be computed using the polynomial long division method (this process is not discussed in this material).

- Step 18: Now, the following polynomial equations are available:

$$L(x)$$

$$R(x)$$

$$O(x)$$

$$H(x)$$

$$Z(x)$$

$$T(x)$$

We know

$$T(x) = H(x) \cdot Z(x)$$

$$O(x) = L(x) \cdot R(x)$$

$$T(x) = L(x) \cdot R(x) - O(x)$$

We can write

$$H(x) \cdot Z(x) = L(x) \cdot R(x) - O(x)$$

$$H(x) \cdot Z(x) + O(x) = L(x) \cdot R(x)$$

$$Q(x) = H(x) \cdot Z(x) + O(x)$$

The prover has to prove that $L \cdot R = Q$ and for that commitments need to generated by the prover. We are having $L(x)$, $R(x)$ and $Q(x)$ in the form of polynomial equation but this has to be the group point since we are going to apply pairing.

- Step 19: Now, the prover moving to a trusted setup, where we will be generating a secret s , This should be hidden not disclosed to any other party.
- Step 20: Let us pick a generator point $P_1(g_1)$ and $P_2(g_2)$ from two groups G_1 and G_2 that we have discussed in the pairing chapter.
- Step 21: We know our polynomial equation $L(x)$ and that is in the following form:

$$L(x) = l_0 + l_1x + l_2x^2 + \dots + l_dx^d.$$

To understand clearly, let us take this $529x^3 + 359x^2 + 354x + 43$ polynomial and write with respect to l .

$$\begin{aligned} l_0 &= 43 \\ l_1 &= 354 \\ l_2 &= 359 \\ l_3 &= 529 \end{aligned}$$

Here, the value of $d = 3$. Recall in Step 13, we presented the matrix that includes the polynomials where the first term comes at the last column.

- Step 22: Now, let us compute the commitments as follows.

$$[L]_1 = \sum_{i=0}^d l_i \cdot [s^i]_1 = l_0 g_1 + l_1 g_1^s + l_2 g_1^{s^2} + l_3 g_1^{s^3} + \dots$$

In case of first row value of L , it is as follows:

$$= 43g_1 + 354g_1^s + 359g_1^{s^2} + 529g_1^{s^3}$$

$$R(x) = r_0 + r_1 x + r_2 x^2 + \dots + r_d x^d$$

$$[R]_2 = \sum_{i=0}^d r_i \cdot [s^i]_2 = r_0 g_2 + r_1 g_2^s + r_2 g_2^{s^2} + r_3 g_2^{s^3} + \dots$$

$$Q(x) = q_0 + q_1 x + q_2 x^2 + \dots + q_d x^d$$

$$[Q]_1 = \sum_{i=0}^d q_i \cdot [s^i]_1 = q_0 g_1 + q_1 g_1^s + q_2 g_1^{s^2} + q_3 g_1^{s^3} + \dots$$

Important Note: Here g_1^s means $s \cdot g_1$. We can say that if g_1 (P_1) is a point from the group G_1 then it is $s \cdot g_1$ or $s \cdot P$. The g_1 is from group G_1 , g_2 from group G_2 , if we see $e : G_1 \cdot G_2 \rightarrow G_T$.

- So now prover has to prove that $L \cdot R = Q$. The prover computed $[L]_1$, $[R]_1$, and $[Q]_1$ and this will be given to the verifier. When we write $[L]_1$ means $[L]$ is the commitment of L and 1 in $[L]_1$ refers that it is from the group G_1 . Similarly for the R (which is from group G_2) and Q (which is from group G_1).
- Verifier can then verify if $L \cdot R \stackrel{?}{=} Q$ by computing the pairing and very as follows:

$$e([L]_1, [R]_2) \stackrel{?}{=} e([Q]_1, G_2)$$

The G_2 in $e([Q]_1, G_2)$ is the generator g_2 or Q (means a curve point). We discussed pairing computation in the pairing section. To show both sides are equal:

$$\begin{aligned} e([L]_1, [R]_2) &= e([Q]_1, G_2) \\ e(G_1, G_2)^{LR} &= e(G_1, G_2)^Q \\ e(G_1, G_2)^{LR} &= e(G_1, G_2)^{LR}, \text{ where } Q = LR \end{aligned}$$

10.8.1 Security Issues

The discussed implementation process has the following issues:

- The first issue is on soundness property. Even though, it ensures the completeness, the soundness is not ensured because the dishonest prover can choose random k_1 and k_2 and prove to the verifier. Let us say that

$$\begin{aligned} L1 &= k_1 * g_1 \\ R2 &= k_2 * g_2 \\ C1 &= k_1 * k_2 * g_1 \end{aligned}$$

then equation $e(k_1 f_1, k_2 * g_2) = e(k_1 * k_2 * g_2)$ will get satisfied because it is simply checking whether both are equal or not nothing beyond it means no method to ensure that parameters are derived based on the witness and actual input equations or not.

- The first 2 elements of the Solution vector are known to both Prover and Verifier. Let us consider that as the public inputs called **Instance**. The other 4 elements are known only to Prover and not to the verifier so that is the private inputs and as called **Witness**. The issue here is: even Prover do not generate the proof using random values, the Verifier cannot verify that these were generated using the Public Inputs/Instance which is known to him/her. The question here is, according to the equation $x^3 + x + 5 = 35$, the verifier do not have any provision to verify whether value 35 was used as *out* in the computation that is in the solution vector S .
- The verifier cannot check whether the prover used the known QAP in case QAP is known to the verifier.

10.8.2 Randomized Proof in zk-SNARKs

To fix the issues, random elements (α, β, γ & δ) will be generated during the trusted set up and added with the polynomials. We have three parties: Trusted Party (or Trusted Multi-Party), Prover and Verifier. We will discuss the task of each parties.

Trusted Party in Trusted Setup

The trapdoor elements ($\alpha, \beta, \gamma, \delta, \tau \in F_p$) are generated by a trusted party or trusted multi-parties during the trusted setup. No one else know these values including the prover and verifier. Now using these trapdoor parameters, the commitments will be computed as follows:

$$\begin{aligned} [\alpha]_1 &= \alpha \cdot G_1 \\ [\alpha]_2 &= \alpha \cdot G_2 \\ [\beta]_1 &= \beta \cdot G_1 \\ [\beta]_2 &= \beta \cdot G_2 \\ [\gamma]_1 &= \gamma \cdot G_1 \\ [\gamma]_2 &= \gamma \cdot G_2 \\ [\delta]_1 &= \delta \cdot G_1 \\ [\delta]_2 &= \delta \cdot G_2 \end{aligned}$$

Here, G_1 and G_2 refers Group 1 and 2 respectively. In the above equation, the generator points (g) from the respective group will be applied in case G_1 or G_2 .

Also the Common Reference String (CRS) is computed as follows for both the groups (G_1 and G_2) with the help τ and group generator point.

$$\begin{aligned} CRS_1 &= \tau^0 \cdot g_1, \tau^1 \cdot g_1, \tau^2 \cdot g_1, \dots \\ CRS_2 &= \tau^0 \cdot g_2, \tau^1 \cdot g_2, \tau^2 \cdot g_2, \dots \end{aligned}$$

Note: This can be computed according to the maximum degree of polynomial we want to commit. For example, if you want to commit for 4 degree polynomial in G_1 then $CRS_1 = \{\tau^0 \cdot g_1, \tau^1 \cdot g_1, \tau^2 \cdot g_1, \tau^3 \cdot g_1\}$ and if you want to commit for 3 degree polynomial in G_2 then $CRS_2 = \{\tau^0 \cdot g_2, \tau^1 \cdot g_2, \tau^2 \cdot g_2\}$.

Public Z In the protocol, the polynomial equation Z is known to all including the trusted party or multi-parties. In that case, all the Z coefficients will be chosen and the common reference string will be computed as follows:

$$\begin{aligned} CRSZ_1 &= \tau^0 \cdot z_0 \cdot g_1, \tau^1 \cdot z_1 \cdot g_1, \tau^2 \cdot z_2 \cdot g_1, \dots \\ CRSZ_2 &= \tau^0 \cdot z_0 \cdot g_2, \tau^1 \cdot z_1 \cdot g_2, \tau^2 \cdot z_2 \cdot g_2, \dots \end{aligned}$$

Also, the prover computes the following after getting the L , R and O from the prover.

$$\begin{aligned} CRS_{pub} &= \frac{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}}{\gamma} = \left(\frac{\beta L_i + \alpha R_i + O_i}{\gamma} \cdot G_1 \right)_{i=0}^{l-1} \\ CRS_{pri} &= \frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta} = \left(\frac{\beta L_i + \alpha R_i + O_i}{\delta} \cdot G_1 \right)_{i=l}^{m-1} \end{aligned}$$

Here, m is the total number of elements in the solution vector including l public inputs and $m - l$ private or secret inputs. These are shared by the prover to the trusted set up. **Because of CRS_{pub} and CRS_{pri} , for every circuit the trusted setup has to be called.**

$$CRS_{HZ} = \left(\frac{\tau^i Z(\tau)}{\delta} \cdot G_1 \right)_{i=0}^{n-2}$$

The computed commitments and the CRSs will be given to both the prover and verifier. The trapdoor values may be deleted by the trusted parties to ensure the confidentiality.

Prover

We have already seen upto the Quadratic Arithmetic Program (QAP) polynomial construction. We have also seen that L_x was computed by taking the dot product between the solution vector S and $L(x)$. We have also discussed that the first two parameters value (1 and output 35) are public and remaining are private. Now, will compute the the L_x in two parts: one (L_{xpr}) with private value and another (L_{xpu}) with the public value as follows:

$$L_{xpu} = \sum_{i=0}^{c-1} L(x)_i \cdot S_i; \text{ where } c \text{ is number of public elements. In our example case, it is two values: 1 \& 35.}$$

$$L_{xpr} = \sum_{i=c}^n L(x)_i \cdot S_i; \text{ where } n \text{ is number of elements in } S. \text{ In our example case, it is six.}$$

Similarly, it will be computed for R and O . We will be having $R_{xpu}, R_{xpr}, O_{xpu}, O_{xpr}$.

When the prover generates the proof, he/she generate two random elements $r_1 \in F_p$ and $r_2 \in F_p$ to randomize the proof. Prover then create new polynomials (A and B) considering two inputs L and R as well as the random commitments received from the trusted party that is computed during trusted setup. Hereafter, we consider the $\alpha, \beta, \gamma, \delta$ and τ as its commitments that is $[\alpha], [\beta], [\gamma], [\delta]$

and $[\tau]$. Whether it will be $[\delta]_1$ or $[\delta]_2$ is based on Group generator ($g_1 \in G_1$ or $g_2 \in G_2$) we use in the equation.

$$A = L + \alpha + r_1\delta \quad (46)$$

$$B = R + \beta + r_2\delta \quad (47)$$

Multiplying A and B :

$$AB = \alpha\beta + \alpha R + \alpha r_2\delta + \beta L + LR + r_2\delta L + r_1\beta\delta + r_1\delta R + r_1r_2\delta^2$$

We already know from Step 18 of the implementation process section 10.8, that $LR = O + HZ$.

$$HZ = \left\{ \frac{\tau^i Z(\tau)}{\delta} G_1 \right\}_{i=0}^{n-2}$$

and we know from Step 18 of Section 10.8, Z is $(x-1)(x-2)(x-3)(x-4) = x^4 - 10x^3 + 35x^2 - 50x + 24$. Hence, we can write as follows, and $k_1 = 1, k_2 = 2, k_3 = 3, k_4 = 4$.

$$Z(x) = \prod_{i=1}^n (x - k_i)$$

Now, let us compute the HZ as follows:

At first assume:

$$H(x) = 139x^2 + 480x + 210$$

Now replace $H(x)$ in the equation. Also, consider x as τ for Z .

$$Z(\tau) \cdot H(x) = Z(\tau) \cdot 139x^2 + Z(\tau) \cdot 480x + Z(\tau) \cdot 210$$

The commitment of $Z(\tau) \cdot H(x)$ is that is by replacing the x :

$$[HZ]_1 = Z(\tau) \cdot 139\tau^2 \cdot G_1 + Z(\tau) \cdot 480\tau \cdot G_1 + Z(\tau) \cdot 210 \cdot G_1$$

Rearranging it then we will get the following:

$$[HZ]_1 = 139 \cdot (\tau^2 \cdot Z(\tau) \cdot G_1) + 480 \cdot (\tau \cdot Z(\tau) \cdot G_1) + 210 \cdot (Z(\tau) \cdot G_1)$$

The τ value is not known to the prover, hence it is not possible to commit it. However, those committed values are there with the prover that are sent during the trusted setup by the trusted party. Now, prover has the values $\cdot(\tau^2 \cdot Z(\tau) \cdot G_1), \cdot(\tau \cdot Z(\tau) \cdot G_1), \cdot(Z(\tau) \cdot G_1$ from the Common Reference String (CRS) and that will be used.

IMPORTANT NOTE: The $[HZ]$ is computed and shared by the trusted set up or trusted party. Here it is shown under the prover part to know how it is getting computed.

Now, substitute the following that are very well known in AB :

$$LR = O + HZ, \quad L_x = L_{xpu} + L_{xpr}, \quad R_x = R_{xpu} + R_{xpr}, \quad O_x = O_{xpu} + O_{xpr}$$

We consider here $L = L_x = L_{xpu} + L_{xpr}$ and we know from section 10.8 step 18, $L \cdot R = O + HZ$ ($L(x) \cdot R(x) = O(x) + HZ(x)$). we get the following and all the values are known:

$$AB = \alpha\beta + \alpha(R_{xpu} + R_{xpr}) + r_2\alpha\delta + \beta(L_{xpu} + L_{xpr}) + (O_{xpu} + O_{xpr}) + HZ + r_2\delta L + r_1\beta\delta + r_1\delta R + r_1r_2\delta^2$$

Let us rearrange the terms to bring simplified version:

$$AB = \alpha\beta + (\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}) + (\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ) + r_2\alpha\delta + r_2\delta L + r_1\beta\delta + r_1\delta R + r_1r_2\delta^2$$

Let us consider the following I and J . Here I is based on group G_1 and J is based on group G_2 . This means the commitments $[\alpha]_1, [\beta]_1, [\gamma]_1$ will be applied for I and $[\delta]_1, [\beta]_1, [\delta]_1$ will be applied for J .

$$I = \frac{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}}{\gamma}$$

$$J = \frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta}$$

Now, we apply I and J in AB , will get the following:

$$AB = \alpha\beta + I \cdot \gamma + (J + r_2\alpha + r_2L + r_1\beta + r_1R + r_1r_2\delta) \cdot \delta$$

$$AB = \alpha\beta + I \cdot \gamma + (J + r_2(L + \alpha) + r_1(R + \beta) + r_1r_2\delta) \cdot \delta$$

Now substitute $L + \alpha = A - r_1\delta$ and $R + \beta = B - r_2\delta$ in AB . We know that $A = L + \alpha + r_1\delta$, can be written as $L + \alpha = A - r_1\delta$ and $B = R + \beta + r_2\delta$ can be written as $R + \beta = B - r_2\delta$.

$$\begin{aligned} AB &= \alpha\beta + I \cdot \gamma + (J + r_2(A - r_1\delta) + r_1(B - r_2\delta) + r_1r_2\delta) \cdot \delta \\ &= \alpha\beta + I \cdot \gamma + (J + r_2A - r_1r_2\delta + r_1B - r_1r_2\delta + r_1r_2\delta) \cdot \delta \\ &= \alpha\beta + I \cdot \gamma + (J + r_2A + r_1B - r_1r_2\delta) \cdot \delta \end{aligned}$$

Further, we simplify by considering the following.

$$C = J + r_2A + r_1B - r_1r_2\delta$$

Now, the final equation to prove to the verifier is:

$$AB = \alpha\beta + I \cdot \gamma + C \cdot \delta$$

where

$$I = \frac{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}}{\gamma}$$

$$C = \frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta} + r_2A + r_1B - r_1r_2\delta \quad (48)$$

The prover computed the commitments: $[A]_1, [B]_1, [C]_1$ and sends them to the verifier. The $[A]_1$ is from equation 46, $[B]_1$ is from equation 47 and $[C]_1$ from equation 48. The A become $[A]_1$ when scalar multiplication on the group G_1 ($[A]_1 = A \cdot G_1$ or $[A]_1 = [L]_1 + [\alpha]_1 + r_1[\delta]_1$). To note, $[\alpha]_1$ and $[\delta]_1$ are given in the trusted setup, prover does not know α and δ only known are the commitments $[\alpha]_1$ and $[\delta]_1$. So, wherever the prover equation has α then it is automatically considered as $[\alpha]_1$ however the Group G_1 or G_2 (that is $[\alpha]_1$ or $[\alpha]_2$) is based on the group requirement. For example, in case of A then it is $[A]_1$ and in case of B then it is $[B]_2$.

Verifier

The verifier received the commitments: $[\alpha]_1, [\alpha]_2, [\beta]_1, [\beta]_2, [\gamma]_1, [\gamma]_2, [\delta]_1, [\delta]_2$ from the trusted party or from the prover. The verifier received $[A]_1, [B]_1, [C]_1$ from the prover. Also verifier knows the public parameter (**instance**) of the solution, hence he/she can compute $I = \frac{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}}{\gamma}$ that is L_{xpu} , R_{xpu} and O_{xpu} are public and γ ($[\gamma]_2$) received from the trusted party/prover. So, the verifier has the following:

$$[I]_1, [\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2$$

Using the bilinear property: $e(a \cdot b, c \cdot d) = e(a, c) \cdot e(b, d)$, the verifier can check the following:

$$e([A]_1, [B]_2) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) \cdot e([I]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2) \quad (49)$$

If the equation 49 satisfies then the proof is valid and accepted by the receiver.

How the equation 49 satisfies? Left Hand Side (LHS):

$$\begin{aligned} e([A]_1, [B]_2) &= e(A = L + \alpha + r_1\delta, B = R + \beta + r_2\delta) \\ &= e(L + \alpha + r_1\delta, R + \beta + r_2\delta) \\ &= e(L + \alpha + r_1\delta, R + \beta + r_2\delta) \end{aligned}$$

Let us expand with the help of bilinear property

$$= e(L, R) \cdot e(L, \beta) \cdot e(L, r_2\delta) \cdot e(\alpha, R) \cdot e(\alpha, \beta) \cdot e(\alpha, r_2\delta) \cdot e(r_1\delta, R) \cdot e(r_1\delta, \beta) \cdot e(r_1\delta, r_2\delta)$$

Right Hand Side (RHS):

$$e([\alpha]_1, [\beta]_2)$$

This is straight forward, so let us move on to next.

$$e([I]_1, [\gamma]_2) = e(G_1, G_2)^{[I]_1[\gamma]_2} = e(G_1, G_2)^{\beta L_{pu} + \alpha R_{pu} + O_{pu}}$$

Now, we can see from the above, that $I \cdot \gamma$ has to be applied. So, we can see from the following the equivalent of $I \cdot \gamma$ and apply to the pairing. Hence, we get the equivalent pairing $e(L_{pu}, [\beta]_2) e([\alpha]_1, R_{pu}) e(O_{pu}, [1]_2)$.

$$I = \frac{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}}{\gamma} = I \cdot \gamma = \beta L_{xpu} + \alpha R_{xpu} + O_{xpu}$$

$$e([I]_1, [\gamma]_2) = e(G_1, G_2)^{I \cdot \gamma} = e(G_1, G_2)^{\beta L_{xpu} + \alpha R_{xpu} + O_{xpu}} = e(L_{xpu}, [\beta]_2) e([\alpha]_1, R_{xpu}) e(O_{xpu}, [1]_2)$$

The above is possible with the help of bilinear property as in the following two equations.

$$\begin{aligned} e(G_1 P, G_2 Q) &= e(G_1, G_2)^{PQ} = e(G_1 Q, G_2 P) \\ e(\alpha P + \beta Q + \gamma R, [1]_2) &= e(P, [1]_2)^\alpha \cdot e(Q, [1]_2)^\beta \cdot e(R, [1]_2)^\gamma = e(P, \alpha) \cdot e(Q, \beta) \cdot e(R, \gamma) \end{aligned}$$

Similar to the $e([I]_1, [\gamma]_2)$, next we can do for the $e([C]_1, [\delta]_2)$.

$$\begin{aligned} e([C]_1, [\delta]_2) &= e(G_1, G_2)^{C \cdot \delta} \\ &= e([L]_1, [R]_2) \cdot e([L]_1, [\beta]_2) \cdot e([\alpha]_1, [R]_2) \cdot e([L]_1, [r_2\delta]_2) \cdot e([\alpha]_1, [r_2\delta]_2) \cdot e([r_1\delta]_1, [R]_2) \\ &\quad \cdot e([r_1\delta]_1, [\beta]_2) \cdot e([1]_1, [1]_2)^{r_1 r_2 \delta^2} \end{aligned}$$

where,

$$C = \frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta} + r_2 A + r_1 B - r_1 r_2 \delta$$

Expansion of $e([C]_1, [\delta]_2)$

We start by replacing C_1 :

$$e([C]_1, [\delta]_2) = e\left(\left[\frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta} + r_2 A + r_1 B - r_1 r_2 \delta\right]_1, [\delta]_2\right).$$

We can write the above as follows using the bilinearity property:

$$\begin{aligned} e([C]_1, [\delta]_2) &= e\left(\frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta}, [\delta]_2\right) \cdot e(r_2 A, [\delta]_2) \cdot e(r_1 B, [\delta]_2) \cdot e(-r_1 r_2, [\delta]_2) \\ &= e\left(\frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta}, [\delta]_2\right) = e(G_1, [1]_2)^{\frac{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}{\delta}, [\delta]_2} \end{aligned}$$

Now, the δ cancels, so we get the following:

$$e(G_1, [1]_2)^{\beta L_{xpr} + \alpha R_{xpr} + O_{xpr} + HZ}$$

We know that $LR = O + HZ$ and we have O_{xpu} from I and that is mentioned in I expansion as $e(O_{xpu}, [1]_2)$. we can add the $O_{xpu} + O_{xpr} = O$ because in AB , we can see $I \cdot \gamma + C \cdot \delta$. Now we can exclude $e(O_{xpu}, [1]_2)$ from I , since it is substituted in C .

$$e(P_1 + P_2, Q) = e(P_1, Q) \cdot e(P_2, Q)$$

hence we can write as follows:

$$e(G_1, [1]_2)^{\beta L_{xpr} + \alpha R_{xpr} + LR} = e([L]_1, [R]_2) \cdot e([L_{xpr}]_1, [\beta]_2) \cdot e([\alpha]_1, [R_{xpr}]_2)$$

Note: We assume, the G_1 is there to represent the base point of the group.

$$e([C]_1, [\delta]_2) = e([L]_1, [R]_2) \cdot e([L_{xpr}]_1, [\beta]_2) \cdot e([\alpha]_1, [R_{xpr}]_2) \cdot e([r_2A]_1, [\delta]_2) \cdot e([r_1B]_1, [\delta]_2) \cdot e([-r_1r_2\delta]_1, [\delta]_2) \quad (50)$$

Expand $e([r_2A]_1, [\delta]_2)$

We know that $A = [L + \alpha + r_1\delta]_1$, so expand as follows:

$$\begin{aligned} e([r_2A]_1, [\delta]_2) &= e([r_2(L + \alpha + r_1\delta)]_1, [\delta]_2) \\ &= e([L]_1, [r_2\delta]_2) e([\alpha]_1, [r_2\delta]_2) e([r_1\delta]_1, [r_2\delta]_2). \end{aligned} \quad (51)$$

Expand $e([r_1B]_1, [\delta]_2)$

Similarly, We know that $B = [R + \beta + r_2\delta]_2$, so expand as follows:

$$\begin{aligned} e([r_1B]_1, [\delta]_2) &= e([r_1(R + \beta + r_2\delta)]_1, [\delta]_2) \\ &= e([r_1\delta]_1, [R]_2) e([r_1\delta]_1, [\beta]_2) e([r_1\delta]_1, [r_2\delta]_2). \end{aligned} \quad (52)$$

Last term

The final randomization term expands directly as follows:

$$e([-r_1r_2\delta]_1, [\delta]_2) = e([1]_1, [1]_2)^{-r_1r_2\delta^2}.$$

Combine all terms

Substituting equations (51) and (52) into (50), get the final as follows:

$$e([C]_1, [\delta]_2) = e([L]_1, [R]_2) \cdot e([L_{xpr}]_1, [\beta]_2) \cdot e([\alpha]_1, [R_{xpr}]_2) \cdot e([L]_1, [r_2\delta]_2) \quad (53)$$

$$e([\alpha]_1, [r_2\delta]_2) e([r_1\delta]_1, [R]_2) e([r_1\delta]_1, [\beta]_2) e([r_1\delta]_1, [r_2\delta]_2)^2 e([1]_1, [1]_2)^{-r_1r_2\delta^2} \quad (54)$$

Simplifying the last two terms:

$$e([r_1\delta]_1, [r_2\delta]_2)^2 \cdot e([1]_1, [1]_2)^{-r_1r_2\delta^2} = e([1]_1, [1]_2)^{r_1r_2\delta^2},$$

We get the final compact form:

$$e([C]_1, [\delta]_2) = e([L]_1, [R]_2) \cdot e([L_{xpr}]_1, [\beta]_2) \cdot e([\alpha]_1, [R_{xpr}]_2) \cdot e([L]_1, [r_2\delta]_2) \quad (55)$$

$$e([\alpha]_1, [r_2\delta]_2) e([r_1\delta]_1, [R]_2) e([r_1\delta]_1, [\beta]_2) e([1]_1, [1]_2)^{r_1r_2\delta^2}. \quad (56)$$

Now, let us see the LHS and RHS of $e([A]_1, [B]_2) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) \cdot e([I]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2)$.

RHS =

$$\begin{aligned}
& e([\alpha]_1, [\beta]_2) \cdot e([I]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2) \\
= & e([\alpha]_1, [\beta]_2) e(L_{\text{pu}}, [\beta]_2) e([\alpha]_1, R_{\text{pu}}) e([L]_1, [R]_2) e([L_{\text{xpr}}]_1, [\beta]_2) e([\alpha]_1, [R_{\text{xpr}}]_2) e([L]_1, [r_2\delta]_2) \\
& e([\alpha]_1, [r_2\delta]_2) e([r_1\delta]_1, [R]_2) e([r_1\delta]_1, [\beta]_2) e([1]_1, [1]_2)^{r_1 r_2 \delta^2}
\end{aligned}$$

We can write,

$$\begin{aligned}
& ([L_{\text{xpr}}]_1, [\beta]_2) \cdot e(L_{\text{pu}}, [\beta]_2) = e(L, [\beta]_2) \\
& e([\alpha]_1, [R_{\text{xpr}}]_2) \cdot e([\alpha]_1, R_{\text{pu}}) = e([\alpha]_1, R)
\end{aligned}$$

So, Right Hand Side (RHS) =

$$\begin{aligned}
& e([\alpha]_1, [\beta]_2) e([L]_1, [R]_2) e([L]_1, [\beta]_2) e([\alpha]_1, [R]_2) e([L]_1, [r_2\delta]_2) e([\alpha]_1, [r_2\delta]_2) \\
& e([r_1\delta]_1, [R]_2) e([r_1\delta]_1, [\beta]_2) e([1]_1, [1]_2)^{r_1 r_2 \delta^2} \tag{57}
\end{aligned}$$

LHS =

$$e(L, R) \cdot e(L, \beta) \cdot e(L, r_2\delta) \cdot e(\alpha, R) \cdot e(\alpha, \beta) \cdot e(\alpha, r_2\delta) \cdot e(r_1\delta, R) \cdot e(r_1\delta, \beta) \cdot e(r_1\delta, r_2\delta) \tag{58}$$

We can see the mapping in table 3, hence $e([A]_1, [B]_2) = e([\alpha]_1, [\beta]_2) \cdot e([I]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2)$, if there is no modification.

RHS term	Corresponding LHS term
$e([\alpha]_1, [\beta]_2)$	$e(\alpha, \beta)$
$e([L]_1, [R]_2)$	$e(L, R)$
$e([L]_1, [\beta]_2)$	$e(L, \beta)$
$e([\alpha]_1, [R]_2)$	$e(\alpha, R)$
$e([L]_1, [r_2\delta]_2)$	$e(L, r_2\delta)$
$e([\alpha]_1, [r_2\delta]_2)$	$e(\alpha, r_2\delta)$
$e([r_1\delta]_1, [R]_2)$	$e(r_1\delta, R)$
$e([r_1\delta]_1, [\beta]_2)$	$e(r_1\delta, \beta)$
$e([1]_1, [1]_2)^{r_1 r_2 \delta^2}$	$e(r_1\delta, r_2\delta) = e([1]_1, [1]_2)^{r_1 r_2 \delta^2}$

Table 3: Mapping of RHS expanded terms to LHS

Note: As already mentioned that \square_1 and \square_2 is based on the group G_1 or G_2 get used. In the final proof verification equation 49, the $[\gamma]_2$ is used which is computed using the $g \in G_2$ ($g \cdot \gamma$) and $e(G_1, G_2)$, so the second argument is from the second group that is G_2 . Here, the g is the point in the group and $g \cdot \gamma$ is the scalar multiplication of point with addition and doubling discussed in Elliptic Curve Cryptography Section. In the mapping that is in table 3, we can see RHS is having \square but LHS is not having but it should be however not given in the discussion and proof (we should assume that it is there with respect to the Group (G_1) or (G_2)).

Important Note: In every operation the modulus should be considered since it is based on the Finite Field.

10.8.3 Solution to Issues in Randomized Proof

The issues discussed in section 10.8.1 are addressed in the Randomized proof. The fixes are as follows:

- First issue is that, dishonest prover can generate random k_1 and k_2 and get the proof verified. In the Randomized version, the commitment is not only generated by the prover, also generated by the verifier that is I get generated using the public value. Hence, the prover cannot get any random successfully verified.

- The commitments L , R and O formed by doing the dot product with the solution vector S . However no-way to confirm it. In the randomized version, the parameters α and β creates the linear dependency in computing $A \cdot B$ also these two parameters are involved in computing A and B using the L and R .
- The verifier cannot check whether the prover had used the same public input (i.e. the out) or not. In the randomized version, it can be checked by the verifier because the equation I has the public input divided by γ (only commitment is known to the prover not the original γ value). Hence, if the Prover used different public inputs in computing L, R or O then it will not be balanced out by the part computed by the Verifier in bilinear pairing.
- There are multiple elements (like $\alpha, \beta, \gamma, \delta$ and instances) that are public and someone can try to get the witness by doing the brute force however it is not possible since the prover generated random r_1 and r_2 is used in the proof generation and that is not known to anyone except the prover.

10.9 PLONK

The Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge (PLONK) allows us to use the trusted setup only once for multiple circuits. The word "Oecumenical" (sometimes spelled "ecumenical") referring to the universal that means we can apply for multiple circuits. The trusted setup will be called for the toxic waste generation and its commitment to support a degree n . The prover can use the toxic waste commitments in multiple circuits that is for multiple circuit polynomials less than or equal to degree n . In case of the Groth16, we need to call trusted set up for every arithmetic circuit that is of degree equal to or less than or greater than n . The PLONK need not to call more than once if the circuit polynomial degree is n or less than n . The PLONK IOP (Interactive Oracle Polynomial) can be combined with different polynomial commitment schemes, like KZG (based on pairing) named Aztec or JellyFish, bullet-proof (no pairing and no trusted setup but based on elliptic curve however it is slow at verifer side) named Halo2 and FRI (hashing and no trusted setup) named plonky2. Here, we will be discussing about the PLONK that uses the KZG polynomial commitment scheme.

10.9.1 Lipton's Trick for Permutation Checking

The PLONK uses the permutation check so let us understand the efficient trick for permutation check that is the Lipton's Trick. Let us consider that we have two sequences A and B both are having the same set of numbers or values however it is permuted. The requirement is, how shall we prove that both are same except the arrangements without knowing the entire numbers or values. To verify this, without knowing all numbers or values, we can use the Lipton's Trick (1989). For example, we have a sequence

$$A = (a_1, a_2, \dots, a_n)$$

and another sequence B , which is the permutation of A .

$$B = (b_1, b_2, \dots, b_n).$$

Using these two sequences, define the polynomials as follows.

$$P_A(x) = \prod_{i=1}^n (x - a_i) \quad \text{and} \quad P_B(x) = \prod_{i=1}^n (x - b_i).$$

For example, let us consider $A = \{1, 2, 3\}$. Now, we construct the $P_A(x)$ ad follows:

$$P_A(x) = \prod_{i=1}^n (x - a_i)$$

$$P_A(x) = (x - 1)(x - 2)(x - 3)$$

$$P_A(x) = (x^2 - 2x - x + 2)(x - 3)$$

$$P_A(x) = (x^2 - 3x + 2)(x - 3)$$

$$P_A(x) = x^3 - 3x^2 + 2x - 3x^2 + 9x - 6$$

$$P_A(x) = x^3 - 6x^2 + 11x - 6$$

The coefficients of $P_A(x)$ are $(1, -6, 11, -6)$ and it is three degree polynomial since the maximum or higher exponent is 3.

Similarly, let us do for $P_B(x)$, where $B = \{3, 1, 2\}$.

$$P_B(x) = \prod_{i=1}^n (x - b_i).$$

$$P_B(x) = (x - 3)(x - 1)(x - 2)$$

$$P_B(x) = (x^2 - x - 3x + 3)(x - 2)$$

$$P_B(x) = (x^2 - 4x + 3)(x - 2)$$

$$P_B(x) = x^3 - 4x^2 + 3x - 2x^2 + 8x - 6$$

$$P_B(x) = x^3 - 6x^2 + 11x - 6$$

The coefficients of $P_B(x)$ are $(1, -6, 11, -6)$. Now, we can say that

$$P_A(x) = P_B(x)$$

given two sequences A and B are permutations of each other.

Instead of doing the complex polynomial computation in the above, we can check equality probabilistically by choosing a random r from a large finite field and compute

$$P_A(r) = \prod_{i=1}^n (r - a_i) \quad \text{and} \quad P_B(r) = \prod_{i=1}^n (r - b_i).$$

Now, let us consider the random values $r = 5 \in F_p$, where p is 11 for example. Now, let us compute the $P_A(r)$.

$$P_A(r) = (5 - 1)(5 - 2)(5 - 3)$$

$$P_A(r) = (4)(3)(2)$$

$$P_A(r) = (12)(2)$$

$$P_A(r) = 24$$

Similarly, let us do for $P_B(r)$

$$P_B(r) = (5 - 3)(5 - 1)(5 - 2)$$

$$P_B(r) = (2)(4)(3)$$

$$P_B(r) = (8)(3)$$

$$P_B(r) = 24$$

If $P_A(r) \neq P_B(r)$, the sequences are definitely not permutations; if $P_A(r) = P_B(r)$, they are permutations with high probability. **Important Note: Why we say this as probabilistic not the exact? Because there is a chance that even though numbers are not same, we may get the same result. However, if we try with more than one r values (r_1, r_2, \dots) then accuracy or confidentiality increases and we can say deterministically whether different or equal.**

10.9.2 Prescribed Permutation Check

In the above, we checked that the two vectors are permuted but equal however here we will see the special check that is the prescribed permutation check.

- Let us consider that $W : \Omega \rightarrow \Omega$ is a permutation of Ω if for all $i \in k : W(\omega^i) = \omega^j$ is a bijection.
- Consider $k = 3$ and $W(\omega^0) = \omega^2, W(\omega^1) = \omega^0, W(\omega^2) = \omega^1$.
- Let, f, g are the polynomials in $F_p^{(\leq d)}[X]$. Now the verifier has f, g, W .
- Now the prover wants to prove that $f(y) = g(W(y))$ for all $y \in \Omega$.
- Here the $g(\Omega)$ is same as $f(\Omega)$, permuted by the prescribed W .
- Can we use the zero test to prove that $f(y) - g(W(y)) = 0$? It is not possible because f is the k -degree polynomial however $g(W(y))$ is of k^2 degree. It is k^2 because g is d -degree polynomial (where $d > k$) and W is k -degree polynomial. If we compose W with g then it is polynomial of degree k^2 . The prover need to manipulate the k^2 degree polynomial which will be computationally heavy. We need the linear time method.
- Why we say that the zero test can be used? Because if the two polynomials are same except the permutation then the subtraction of both will give zero since the output will be equal.
- How do we bring it to linear time? We consider an observation, if $(W(a), f(a))_{a \in \Omega}$ is a permutation of $(a, g(a))_{a \in \Omega}$ then $f(y) = g(W(y))$ for all $y \in \Omega$.

$$(W(a), f(a))_{a \in \Omega} \Leftrightarrow (a, g(a))_{a \in \Omega}$$

- The above observation gets true and we see in the following example with $k = 3$:

– We consider the following for $W(a)$

$$W(\omega^0) = \omega^2, W(\omega^1) = \omega^0, W(\omega^2) = \omega^1$$

then Right Tuple, means $(a, g(a))_{a \in \Omega}$ is as follows. We simply replace the a with ω

$$(\omega^0, g(\omega^0)), (\omega^1, g(\omega^1)), (\omega^2, g(\omega^2))$$

- In case of the Left Tuple that is $(W(a), f(a))_{a \in \Omega}$, we know that $W(\omega^0) = \omega^2, W(\omega^1) = \omega^0, W(\omega^2) = \omega^1$. Hence we can write as follows:

$$(\omega^2, f(\omega^0)), (\omega^0, f(\omega^1)), (\omega^1, f(\omega^2))$$

- In the above, if we see that $(\omega^2, f(\omega^0)) = (\omega^2, g(\omega^2))$ because the $g(\omega^2)$ is $g(W(\omega^0))$ as per the consideration that $W(\omega^0) = \omega^2$. Also, we know that the $f(y) = g(W(y))$. Hence, $f(\omega^0) = g(W(\omega^0)) = g(\omega^2)$, where $W(\omega^0) = \omega^2$.
- Likewise, $(\omega^0, f(\omega^1)) = (\omega^0, g(\omega^0))$ because the $g(\omega^0)$ is $g(W(\omega^1))$ as per the consideration $W(\omega^1) = \omega^0$.
- Also, $(\omega^1, f(\omega^2)) = (\omega^1, g(\omega^1))$ because the $g(\omega^1)$ is $g(W(\omega^2))$ as per the consideration $W(\omega^2) = \omega^1$.

- Here, to check the similarity, we have to check the pairs.
- For the proof, we can create the bivariate polynomial. The bivariate polynomial means we use two variables. In case of the univariate polynomial, we only use x on the right hand side and in bivariate polynomial, we use two variables for example, x and y on the right hand side.
- We write the following bivariate polynomials with degree k for Left and Right tuple:

$$f(X, Y) = \prod_{a \in \Omega} (X - Y) \cdot W(a) - f(a)$$

$$g(X, Y) = \prod_{a \in \Omega} (X - Y) \cdot a - g(a)$$

The proof is

$$f(X, Y) = g(X, Y) \Leftrightarrow (W(a), f(a))_{a \in \Omega} \text{ is permutation of } (a, g(a))_{a \in \Omega}$$

- To prove this, let us consider that the prover has f, g, W and the verifier has the commitment of f, g, W . Now, the verifier sends the random r and s to the prover to send the committed proof to verify. The reason for sending the two random numbers is we have bivariate polynomials and we need value for both X and Y .
- We can prove the following using the product check.

$$f(r, s) = g(r, s)$$

- The product check is

$$\prod_{a \in \Omega} \frac{r - s \cdot W(a) - g(a)}{r - s \cdot a - g(a)} = 1$$

In the above, we have two random variables r and s and in addition a value also required to pass as input to the univariate polynomial.

Now the question is how the verifier verifies it. Now let, us consider that the Z is

$$Z = \prod_{a \in \Omega} \frac{r - s \cdot W(a) - g(a)}{r - s \cdot a - g(a)} = 1$$

Taking the r and s , the prover computes Z and send it. Now, it is possible that prover will simply send $Z = 1$ without any computation. How, the verifier will come to know about it? The answer is, the verifier takes the following from the prover:

- commitments C_f, C_g, C_W
- $f(a)$
- $g(a)$
- $W(a)$
- Z
- $a \in \Omega$, where Ω is the publicly known set.

Using these values, the verifier will compute and check whether the sent $Z = 1$ is correct or not.

Important Consideration: The PLONK does not use the pair (r, s) instead it uses the (α, γ) however both are same. The difference to understand is that the (r, s) is getting used in the bivariate polynomial instead of that we can use the (α, γ) in univariate polynomial as follows:

$$H(X) = (W(X), f(X)) \mapsto f(X) + \beta \cdot W(X) + \gamma$$

$$G(X) = (X, g(X)) \mapsto g(X) + \beta \cdot X + \gamma$$

Here, both $H(X)$ and $G(Y)$ are the univariate polynomial. There is no difference between $(r = Y, s)$ [where $X = a$] and (α, γ) . Only difference is how we are going to handle. Here r and s are considered as symbolic that means not fixed (that is just conceptual point) however in case of (α, γ) , it is fixed value. If it is with (r, s) then

$$H(X, Y) = Y - s \cdot W(X) - f(X)$$

$$G(X, Y) = Y - s \cdot X - g(X)$$

Here both $H(X, Y)$ and $G(X, Y)$ are the bivariate polynomial

10.9.3 Implementation Process

Now, let us go through the implementation process of the PLONK with KZG polynomial commitment. The proof here is given the public circuit C and the public input x , the prover knows secret witness w by showing that $C(x, w)$ outputs 0 or respective value. The process includes three parties: Trusted set up, Prover and Verifier.

The proof verifications is as follows:

- The polynomial encodes the right input that means the circuit is given the right public and secret inputs.
- Every gate in the circuit is implemented correctly.
- Wiring is implemented correctly: Here wiring means, the same number that is given as input in multiple gates. That has to be checked.
- the output of the last gate in the circuit is 0 (even though it is given as 0, it should be what the actual value is).

The process is as follows

- First choose a polynomial and create the arithmetic circuit as we have discussed in the figure 15. Let us consider the polynomial equation $(x_1 + x_2)(x_2 + w)$, where $x_1 = 5, x_2 = 6, w = 1$. Here, x_1 and x_2 values are public and the w value is the secret or witness. The secret will not be shared with any one.
- The figure 16 shows the circuit for the polynomial $(x_1 + x_2)(x_2 + w)$. The circuit contains three gates (two addition gate and one multiplication gate). Given the input to each variable finally produces the output 77. Now, the circuit C and the x_1 and x_2 values are public so known to the verifier but w is not known. Without sharing w , the prover can prove to the verifier that the hidden input w along with x_1 and x_2 values to the circuit produces 77 as the output.
- We construct the trace table with left inputs, right inputs, and outputs, which is called arithmetization. The table 5 is the traces of the circuit C .
- Let us encode the trace as a polynomial with the following consideration:
 - $|C|$ - total number of gates in circuits $\Rightarrow 3$
 - $|I|$ - total number of public and private (witness) inputs $\Rightarrow 2 + 1 = 3$
 - We consider degree $d \leq 3|C| + |I| \Rightarrow 12$. It is $3|C|$ because each gate includes two inputs and one output.
 - We define a variable Ω , where $\Omega := \{\omega^0, \omega^1, \omega^2, \omega^3 \dots \omega^{d-1}\}$

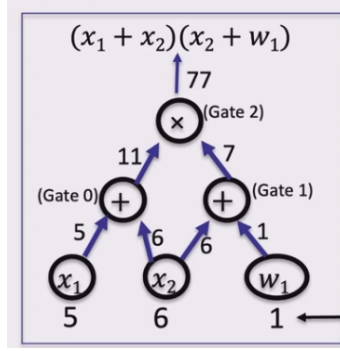


Figure 16: Arithmetic circuit [Copied from Prof. Dan Boneh Lecture]

INPUTS	Publicly known input = 5	Publicly known input = 6	Witness or secret = 1
For gates	Left input	Right input	Output
GATE 0	5	6	11
GATE 1	6	1	7
GATE 2	11	7	77

Table 4: PLONK Trace Table

- We can use Lipston’s Trick (1989) for the permutation proof.
- The prover interpolates the polynomial $T \in F_p^{(\leq d)}[X]$ using Ω (which will be used as the points as we have discussed earlier in the Groth16). The $F_p^{(\leq d)}[X]$ indicates that the polynomial T should bound on F_p that means the coefficients should not be more than $p - 1$ because of finite field, the polynomial should be of degree $\leq d$ with variable as X .
- The polynomial T should encode all inputs and the wires (that is the left input, right input and the output of the gates).
- Now, we can write the following table 5 based on public inputs, witness and the gate. We have in total 12 points. Even though gate 0 and gate 1 inputs reflects the public (x_1 and x_2) and private (w) inputs we consider as different.

INPUTS	$T(w^{-1}) = 5$	$T(w^{-2}) = 6$	$T(w^{-3}) = 1$
GATE 0	$T(w^0) = 5$	$T(w^1) = 6$	$T(w^2) = 11$
GATE 1	$T(w^3) = 6$	$T(w^4) = 1$	$T(w^5) = 7$
GATE 2	$T(w^6) = 11$	$T(w^7) = 7$	$T(w^8) = 77$

Table 5: PLONK Trace Table

- The degree of the polynomial that will be generated from these points will be 11. As we discussed in the Lagrange interpolation that is given n points, we can generate the polynomial of $n - 1$ degree. Here, the points are 12, so we get the polynomial of degree 11.
- **Important Note:** As we discussed in the Groth16, from the trace points (gate’s left inputs, right inputs and outputs) we generate the polynomial using the Lagrange interpolation however, we do not need the polynomial only the coefficients are sufficient.
- To generate the polynomial, we can use the Lagrange interpolation which takes $O(d^2)$ however the Fast Fourier Transform (FFT) takes only $O(d \log(d))$ time to find the coefficient. Here d is the degree of the polynomial. Since, the FFT is efficient, we can use it to find the coefficients of the polynomial.
- Let us use the FFT and compute the coefficients in the following:

- The FFT algorithm works efficiently (using the Cooley–Tukey FFT algorithm) when the domain size or input size N is a power of two, because it recursively splits the computation in half at each stage. In our example, the number of points are 12 which is not power of 2. Hence, we can add the padding that means we will add 0's to fulfill the power of two. For example, we have the following points:

$$[5, 6, 1, 5, 6, 11, 6, 1, 7, 11, 7, 77]$$

and pad it to 16 points to bring the power of two.

$$[5, 6, 1, 5, 6, 11, 6, 1, 7, 11, 7, 77, 0, 0, 0, 0]$$

- **Important Note: We need only the inputs and outputs of gates as we discussed in Groth16. However, here we consider the inputs (both private and public inputs) separately also because Permutation argument depends on full wiring, not just gates - a dishonest prover could supply inconsistent witness values at the start to cheat.**
- We need to find the primitive 16-th root of unity v such that $v^{16} \equiv 1 \pmod{17}$ no smaller power (means any positive exponent less than 16 excluding 0) gives 1. For example, if we compute $v^{14} \pmod{17}$ then it will not give 1 only $v^{16} \pmod{17}$ will give 1. We can take 3 so that we can get $v^{16} \equiv 1 \pmod{17} = 3^{16} \equiv 1 \pmod{17}$.
- FFT (Direct DFT formula) over F_{17} is as follows:

$$X_k = \sum_{n=0}^{15} x_n \cdot v^{nk} \pmod{17}, \quad k = 0, 1, \dots, 15 \quad (59)$$

Here x refers to the numbers in the vector $[5, 6, 1, 5, 6, 11, 6, 1, 7, 11, 7, 9, 0, 0, 0, 0]$, v is the primitive 16^{th} root, N is the total number of points. We can see from the above equation 59 that, for every X_k , we have to compute the process 16 times hence $O(n^2)$.

- We can use the Cooley–Tukey FFT algorithm to compute the coefficients and this needs the power of two. Hence, we considered 16 in our case with padding 0's and it needs only $n \log n$ time.
- We should reduce all numbers to modulo 17 because it is Finite Field F_{17} . One value in our set that is 77 is higher so we reduce it by computing the $77 \pmod{17} = 9$. Hence, our new vector is as follows.

$$[5, 6, 1, 5, 6, 11, 6, 1, 7, 11, 7, 9, 0, 0, 0, 0]$$

- The Cooley–Tukey FFT algorithm works based on divide and conquer method. It repeatedly divide the DFT into smaller DFTs of half the size. So, we divide into two as even and odd.

$$x_{even} = [5, 1, 6, 6, 7, 7, 0, 0]$$

$$x_{odd} = [6, 5, 11, 1, 11, 9, 0, 0]$$

- Further, split as follows:

$$x_{even,even} = [5, 6, 7, 0]$$

$$x_{even,odd} = [1, 6, 7, 0]$$

$$x_{odd,odd} = [5, 1, 9, 0]$$

$$x_{odd,even} = [6, 11, 11, 0]$$

- Each 4-point FFT consists of two 2-point butterflies. Compute Base 2-point FFTs based on pairs (a,b), as follows:

$$A_0 = (a + b) \pmod{17}$$

$$A_1 = (a - b) \text{ mod } 17$$

For example in first pair [5, 6], the value $a = 5$ and $b = 6$. Now, we get

Input	A_0	A_1
[5,6]	11	16
[1,6]	7	12
[7,0]	7	7
[7,0]	7	7
[6,11]	0	12
[5,1]	6	4
[9,0]	9	9

We computed 2-point FFT and now for each split, compute 4-point FFT using the following formula:

$$F_0 = E_0 + O_0, \quad F_1 = E_1 + v_4 O_1, \quad F_2 = E_0 - O_0, \quad F_3 = E_1 - v_4 O_1,$$

with $v_4 = 13 \text{ mod } 17$ means we compute $v^4 = 3^4 = 81 \text{ mod } 17 = 13$. If we see in the above, v_4 is not used in the F_0 and F_2 because it will be 1, so not used. To understand it more clearly, we can say that we are going for v^4 because of 4-point FFT that means for the 16-points, we go for the 4-points so $16/4 = 4$. When we apply the v^4 in F_0, F_1, F_2, F_3 , we consider the index so F_0 and F_2 are 0-index and F_1 and F_3 are the 1st index. Now, in case of 0-index, we write $v^{4k} = v^{4(0)}$, where k is the index and in case of 1-index, we write $v^{4k} = v^{4(1)}$. Hence, $v^{4(0)} = v^0$ is 1 so not included in the equation however, $v^{4(1)} = v^4$ will give different value so included.

Important Note: Here v_4 is the twiddle factor. In case of the earlier formula: $A_1 = (a + b) \text{ mod } 17$ and $A_1 = (a - b) \text{ mod } 17$, the twiddle factor is not used because each pair is already at the base size. The twiddle factor is used when combining smaller DFTs into a larger DFT.

– For first set,

$$x_{\text{even,even}} = [5, 6, 7, 0]$$

Referring the computation in the above table, let us compute the following

$$\text{Even 2-point FFT of } [5, 6] = [A_0 = 11, A_1 = 16]$$

$$\text{Odd 2-point FFT of } [7, 0] = [A_0 = 7, A_1 = 7]$$

$$F_0 = E_0 + O_0 = 11 + 7 = 18 \equiv 1 \text{ mod } 17$$

$$F_1 = E_1 + v_4 O_1 = 16 + 13 \cdot 7 = 16 + 91 = 107 \equiv 5 \text{ mod } 17$$

$$F_2 = E_0 - O_0 = 11 - 7 = 4 \text{ mod } 17$$

$$F_3 = E_1 - v_4 O_1 = 16 - 91 = -75 \equiv 16 \text{ mod } 17$$

$$x_{\text{even,even}} \text{ FFT} = [1, 5, 4, 16]$$

– For second set,

$$x_{\text{even,even}} = [1, 6, 7, 0]$$

Referring the computation in the above table, let us compute the following

$$\text{Even 2-point FFT} = [7, 12]$$

$$\text{Odd 2-point FFT} = [7, 7]$$

$$F_0 = 7 + 7 = 14 \text{ mod } 17$$

$$F_1 = 12 + 13 \cdot 7 = 12 + 91 = 103 \equiv 1 \text{ mod } 17$$

$$F_2 = 7 - 7 = 0 \text{ mod } 17$$

$$F_3 = 12 - 91 = -79 \equiv 12 \text{ mod } 17$$

$$x_{\text{even,odd}} \text{ FFT} = [14, 1, 0, 12]$$

– For Third Set

$$x_{odd,odd} = [5, 1, 9, 0]$$

Referring the computation in the above table, let us compute the following

$$\text{Even 2-point FFT} = [6, 4]$$

$$\text{Odd 2-point FFT} = [9, 9]$$

$$F_0 = 6 + 9 = 15 \pmod{17}$$

$$F_1 = 4 + 13 \cdot 9 = 4 + 117 = 121 \equiv 2 \pmod{17}$$

$$F_2 = 6 - 9 = -3 \equiv 14 \pmod{17}$$

$$F_3 = 4 - 117 = -113 \equiv 4 \pmod{17}$$

$$x_{odd,odd} \text{ FFT} = [15, 2, 14, 4]$$

– For fourth Set

$$x_{odd,even} = [6, 11, 11, 0]$$

Referring the computation in the above table, let us compute the following

$$\text{Even 2-point FFT} = [0, 6]$$

$$\text{Odd 2-point FFT} = [11, 11]$$

$$F_0 = 0 + 11 = 11 \pmod{17}$$

$$F_1 = 6 + 13 \cdot 11 = 6 + 143 = 149 \equiv 13 \pmod{17}$$

$$F_2 = 0 - 11 = -11 \equiv 6 \pmod{17}$$

$$F_3 = 6 - 143 = -137 \equiv 11 \pmod{17}$$

$$x_{odd,even} \text{ FFT} = [11, 13, 6, 11]$$

– Now let us combine even 4 points to 8 points: for combining two four points, that is combining $x_{even,even}$ FFT and $x_{even,odd}$ FFT with $v_8 = v^2 = 9$ means we compute $v^2 = 3^2 = 9 \pmod{17} = 9$. Here v^2 taken because combining two 4-points to bring 8-points so $v^{16/8} = v^2$. We know that the 16 is the size or total number of points in the initial vector including the paddings.

$$E = [1, 5, 4, 16], \quad O = [14, 1, 0, 12]$$

$$\text{Now } v_8 \text{ powers} = [v_8^0, v_8^1, v_8^2, v_8^3] \pmod{17} = [9^0 \pmod{17}, 9^1 \pmod{17}, 9^2 \pmod{17}, 9^3 \pmod{17}] = [1, 9, 13, 15]$$

We know already, that

$$E'_k = E_k + v_k^4 \cdot O_k \pmod{17}$$

$$E'_{k+4} = E_k - v_k^4 \cdot O_k \pmod{17}$$

Using the above formula, compute the following.

$$k = 0 : X_0 = 1 + 1 \cdot 14 \pmod{17} = 15, \quad X_4 = 1 - 14 \pmod{17} = 4$$

$$k = 1 : X_1 = 5 + 9 \cdot 1 \pmod{17} = 14, \quad X_5 = 5 - 9 \pmod{17} = 13$$

$$k = 2 : X_2 = 4 + 13 \cdot 0 \pmod{17} = 4, \quad X_6 = 4 - 0 \pmod{17} = 4$$

$$k = 3 : X_3 = 16 + 15 \cdot 12 \pmod{17} = 16 + 180 = 10 \quad X_7 = 16 - 180 \pmod{17} = 3$$

$$\text{Even 8-point FFT} = [15, 14, 4, 10, 4, 13, 4, 3]$$

– Now let us combine odd 4 points to 8 points: for combining two four points, that is combining $x_{odd,even}$ FFT and $x_{odd,odd}$ FFT with $v_8 = v^2 = 16$ means we compute $v^2 = 3^2 = 9 \pmod{17} = 9$. Here v^2 taken because combining two 4 points.

$$\text{Now } v_8 \text{ powers} = [v_8^0, v_8^1, v_8^2, v_8^3] \pmod{17} = [9^0 \pmod{17}, 9^1 \pmod{17}, 9^2 \pmod{17}, 9^3 \pmod{17}] = [1, 9, 13, 15]$$

The points are

$$x_{odd,even} = E = [11, 13, 6, 11], \quad x_{odd,odd} = O = [15, 2, 14, 4]$$

We know already, that

$$\begin{aligned} E'_k &= E_k + v_k^4 \cdot O_k \pmod{17} \\ E'_{k+4} &= E_k - v_k^4 \cdot O_k \pmod{17} \end{aligned}$$

Using the above formula, compute the following.

$$\begin{aligned} k = 0 : \quad X_0 &= 11 + 15 = 26 \equiv 9, & X_4 &= 11 - 15 = -4 \equiv 13 \\ k = 1 : \quad X_1 &= 13 + 9 \cdot 2 = 13 + 18 = 14, & X_5 &= 13 - 18 = -5 \equiv 12 \\ k = 2 : \quad X_2 &= 6 + 13 \cdot 14 = 6 + 182 = 12, & X_6 &= 6 - 182 = -176 \equiv 12 \\ k = 3 : \quad X_3 &= 11 + 15 \cdot 4 = 11 + 60 = 13, & X_7 &= 11 - 60 = -49 \equiv 2 \end{aligned}$$

$$\text{Odd 8-point FFT} = [9, 14, 12, 13, 13, 12, 12, 2]$$

- Now combine the 8 points to 16 points FFT. **Important Note: When we go to combine the two 8 FFT then the v should be v_{16} . We know that when we combine two 2 FFT to one 4 FFT then we compute $v_4 = v^4 = v^{16/4}$, likewise when we combine two 4 FFT to one 8 FFT then we compute $v_8 = v^2 = v^{16/8}$, therefore when we combine two 8 FFT to 16 FFT then we compute $v_{16} = v^1 = v^{16/16} = 3^1 = 3$.**
- Now v_{16} powers = $[v_{16}^0, v_{16}^1, v_{16}^2, v_{16}^3, \dots, v_{16}^7] \pmod{17}$
 $= [3^0 \pmod{17}, 3^1 \pmod{17}, 3^2 \pmod{17}, 3^3 \pmod{17}, \dots, 3^7 \pmod{17}] = [1, 3, 9, 10, 13, 5, 15, 11]$
- The points are

$$x_{\text{even}} = E = [15, 14, 4, 10, 4, 13, 4, 3], \quad x_{\text{odd}} = O = [9, 14, 12, 13, 13, 12, 12, 2]$$

- We know already, that

$$\begin{aligned} E'_k &= E_k + v_k^{16} \cdot O_k \pmod{17} \\ E'_{k+8} &= E_k - v_k^{16} \cdot O_k \pmod{17} \end{aligned}$$

Using the above formula, compute the following.

For $k = 0$:

$$X_0 = 15 + 1 \cdot 9 = 24 \pmod{17} = 7, \quad X_8 = 15 - 1 \cdot 9 = 6$$

For $k = 1$:

$$X_1 = 14 + 3 \cdot 14 = 56 \pmod{17} = 5, \quad X_9 = 14 - 42 = -28 \pmod{17} = 6$$

For $k = 2$:

$$X_2 = 4 + 9 \cdot 12 = 112 \pmod{17} = 10, \quad X_{10} = 4 - 108 = -104 \pmod{17} = 15$$

For $k = 3$:

$$X_3 = 10 + 10 \cdot 13 = 140 \pmod{17} = 4, \quad X_{11} = 10 - 130 = -120 \pmod{17} = 16$$

For $k = 4$:

$$X_4 = 4 + 13 \cdot 13 = 173 \pmod{17} = 3, \quad X_{12} = 4 - 169 = -165 \pmod{17} = 5$$

For $k = 5$:

$$X_5 = 13 + 5 \cdot 12 = 73 \pmod{17} = 5, \quad X_{13} = 13 - 60 = -47 \pmod{17} = 4$$

For $k = 6$:

$$X_6 = 4 + 15 \cdot 12 = 184 \pmod{17} = 14, \quad X_{14} = 4 - 180 = -176 \pmod{17} = 11$$

For $k = 7$:

$$X_7 = 3 + 11 \cdot 2 = 25 \pmod{17} = 8, \quad X_{15} = 3 - 22 = -19 \pmod{17} = 15$$

The output is

$$X = [7, 5, 10, 4, 3, 5, 14, 8, 6, 6, 15, 16, 5, 4, 11, 15]$$

- Finally the coefficients of the polynomial $T(x)$ required for the next computation is as follows.

$$X = [7, 5, 10, 4, 3, 5, 14, 8, 6, 6, 15, 16, 5, 4, 11, 15]$$

- The Polynomial from the FFT coefficient $X = [7, 5, 10, 4, 3, 5, 14, 8, 6, 6, 15, 16, 5, 4, 11, 15]$ is as follows:

$$T(X) = 7+5X+10X^2+4X^3+3X^4+5X^5+14X^6+8X^7+6X^8+6X^9+15X^{10}+16X^{11}+5X^{12}+4X^{13}+11X^{14}+15X^{15}$$

- **We initially had 12 points in the vector so the polynomial that will be produced should have the degree of 11. However, due the FFT, we have padded 4 0's and solved. Now, we got 16 coefficients and this will have more than degree 11. How will the proof works then? The answer is, the extra four points are because of the padding 0. The padding 0 will not change the polynomial it is just a trick. The produced coefficients will be like produced from the 12 points. This will be clear when we do the inverse FFT.**

The inverse FFT formula (mod 17) is:

$$x_k = \frac{1}{16} \sum_{j=0}^{15} X_j \cdot (v_{16}^{-k})^j \pmod{17}, \quad k = 0, \dots, 15$$

In the above, x is the new coefficients that we will bring from the extended coefficients (X). We can write $(v_{16}^{-k})^j$ as $(v_{16}^{-1 \cdot k})^j$. We already discussed about computing the inverse. In this case $v_{16} = 3$ as we already seen that. The inverse of 3 is 6. Now, the value 6 can be used for v_{16}^{-1} and compute the vectors.

We can compute the modulo inverse using the Extended Euclidean Algorithm. After applying the inverse FFT, we get the recovered coefficients (including the actual and padded with 0's) as follows:

$$x_{\text{recovered}} = [5, 6, 1, 5, 6, 11, 6, 1, 7, 11, 7, 9, 0, 0, 0, 0]$$

The inverse result shows that the padding will not impact the computation.

- **The above computation result may mismatch because of the computational mistake however the methodology is followed.**
- **Important Note: The term PLONK includes Lagrange's in the name however we are replacing it with FFT. How will it be PLONK now? The answer is: Any polynomial of degree $< n$ can be written in two bases:**

- Monomial basis:

$$f(X) = a_0 + a_1X + a_2X^2 + \dots + a_{n-1}X^{n-1}$$

- Lagrange basis at roots of unity (ω_i): Given n distinct points

$$\Omega = \{\omega_0, \omega_1, \dots, \omega_{n-1}\}$$

and a polynomial $f(X)$ of degree $< n$, the Lagrange basis representation is:

$$f(X) = \sum_{i=0}^{n-1} f(\omega_i) \cdot \ell_i(X)$$

where $\ell_i(X)$ are the Lagrange polynomials satisfying:

$$\ell_i(\omega_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

means only the matching ($i = j$) is applied and others are rejected while computing the $f(X)$.

- In PLONK, polynomials are committed and evaluated in the Lagrange basis at the evaluation domain $\Omega = \{\omega_0, \dots, \omega_{n-1}\}$. Even though, it is not using the lagrange interpolation for constructing or reconstructing the polynomial from the given points, the commitments are based on the Lagrange basis (Lagrange basis is the set of polynomials and these form a basis for all polynomials of degree $< n$).

- **Advantage:** The $f(\omega_i)$ are already the evaluations at the points and no need to compute $f(\omega_i)$ from coefficients, which saves interpolation cost. This means, we have the computed set of polynomials to bring the interpolated polynomial output (we need only addition of set of polynomials $(f(\omega_1) + f(\omega_2) + \dots + f(\omega_n))$ and multiplicative with 1 or 0 that is $l_i(\omega_j)$) however in case of monomial basis we have to compute the entire polynomial including the multiplication of coefficients and the addition of all terms thus computationally costly.

- Now, let us commit the polynomial $T(X)$ using the KZG polynomial commitment scheme.
- We have to consider that the trusted party computes the toxic waste t and compute the following reference string:

$$(g^{t^i}) = (g^{t^0}), (g^{t^1}), (g^{t^2}), (g^{t^3}) \dots (g^{t^{d-1}})$$

We can consider this as

$$m = (g^{t^i})$$

$$m_0 = (g^{t^0}), m_1 = (g^{t^1}), m_2 = (g^{t^2}), m_3 = (g^{t^3}) \dots, m_{d-1} = (g^{t^{d-1}})$$

The g is the generator and known to all parties.

- **Important Note:** We have considered d as 11 in our example, however if we see the FFT produced polynomial, it has more than $d = 11$ that is it has 16 coefficients. Two possibilities: 1) We may repeat the m_{d-1} for the remaining coefficients or 2) The prover may give reference string considering the power of two (that is like, either 2^4 or 2^5 or 2^{64}) so that it will map with the FFT produced coefficients.
- We know that the Plonk does not need to have different toxic waste for every circuit if the degree is less than or equal to the given. For example, if the reference string computed by the prover for degree d then for any circuit which has the polynomial less than or equal to degree d can use it.
- We consider the g as 3 and the toxic waste as $t = 3$. The KZG commitment T on the $T(X)$ is as follows:

$$T = \sum_{i=0}^{15} m_i^{T(X)_i} \pmod{17}$$

Just for an example, to compute the first two element values,

$$m_0 = g^{t^0} = 3^{3^0} = 3^1 = 3 \pmod{17}$$

$$T_0 = m_0^{T(X)_0} = 3^7 = 2187 = \pmod{17} = 11$$

For the next,

$$m_1 = g^{t^1} = 3^{3^1} = 3^3 = 27 \pmod{17} = 10$$

$$T_1 = m_1^{T(X)_1} = 10^5 = 100000 = \pmod{17} = 6$$

We do like this for all elements in the $T(X) = [7, 5, 10, 4, 3, 5, 14, 8, 6, 6, 15, 16, 5, 4, 11, 15]$ and it will produce the following output.

$$T = 3$$

- Now prover committed T to the polynomial $T(x)$ and send to the verifier. The committed T is a constant commitment and it just a single group element. In addition, the verifier will get some more parameters that we will see in the following test.
- Now, verifier will check one by one:
- Test 1: Check whether T encodes the correct input or not:

- Now the prover and verifier interpolates a polynomial $S(X) \in F_p^{(\leq |I_x|)}[X]$ that encodes the x -inputs (public inputs) to the circuit, where $j = 1, 2, \dots, |I_x|$ and we can write the polynomial as $S(\omega^{-j})$. We have already seen from table 5 that the public inputs are $\omega^{-j} \Rightarrow \omega^{-1} = 5$ and $\omega^{-2} = 6$. The $|I_x|$ is number of public inputs chosen earlier. In our example, it is two.
- The verifier need to ensure that the $S(\omega^{-1}) = 5$ and $S(\omega^{-2}) = 6$ while computing the circuit. How to confirm that? We know that $\Omega_{inp} = \{\omega^{-1}, \omega^{-2}\}$ and $|I_x|$ is 2.
- We have already considered that the

$$\omega = 3 \in \mathbb{F}_{17}$$

Next, we compute the modular inverse and we already discussed about computing the modular inverse

$$\omega^{-1} = 3^{-1} \pmod{17} = 6$$

$$\omega^{-2} = (\omega^{-1})^2 = 6^2 = 36 \equiv 2 \pmod{17}.$$

$$\Omega_{inp} = \{6, 2\}.$$

- We have two points $S(\omega^{-1}) = 5$ and $S(\omega^{-2}) = 6$
- Now let us use the Lagrange's interpolation to compute the polynomial. Since, the input size is less so it will not be very complex to compute the polynomial using the Lagrange's interpolation. We may also use the FFT, if required.
- For two points (x_1, y_1) and (x_2, y_2) , the lagrange's interpolation is as follows:

$$S(X) = y_1 \cdot \frac{X - x_2}{x_1 - x_2} + y_2 \cdot \frac{X - x_1}{x_2 - x_1}$$

- Take the inputs for the points and compute

$$x_1 = 6, \quad y_1 = 5, \quad x_2 = 2, \quad y_2 = 6$$

$$S(X) = 5 \cdot \frac{X - 2}{6 - 2} + 6 \cdot \frac{X - 6}{2 - 6}$$

$$S(X) = 5 \cdot \frac{X - 2}{4} + 6 \cdot \frac{X - 6}{-4}$$

- Now, we have to bring the denominator to the numerator as follows.

$$S(X) = 5 \cdot (X - 2) \cdot 4^{-1} + 6 \cdot (X - 6) \cdot -4^{-1}$$

- Let us compute the modular inverse. We can use the Extended Euclidean algorithm for this.

$$4^{-1} \equiv 13 \pmod{17}, \quad (-4)^{-1} \equiv 4 \pmod{17}$$

So:

$$S(X) = 5 \cdot 13(X - 2) + 6 \cdot 4(X - 6)$$

- Expand and reduce mod 17 because the values should be less than 17 since the finite field $F_p = F_{17}$. No value should exceed 17 and it should be the member of the finite field.

$$S(X) = (65X - 130) + (24X - 144) = 89X - 274$$

Now reduce each coefficient mod 17:

$$89 \pmod{17} = 4, \quad -274 \pmod{17} = -274 + 289 = 15$$

So:

$$S(X) = 4X + 15 \quad \text{in } \mathbb{F}_{17}$$

- Now, let us verify that the $S(\omega^{-1}) = 5$ and $S(\omega^{-2}) = 6$ holds or not for the values 6 and 2 from Ω_{inp} :

$$S(6) = 4 \cdot 6 + 15 = 24 + 15 = 39 \equiv 5 \pmod{17}$$

$$S(2) = 4 \cdot 2 + 15 = 8 + 15 = 23 \equiv 6 \pmod{17}$$

- The interpolated $S(X)$ at the verifier side is as follows:

$$S(X) = 4X + 15 \quad \text{over } \mathbb{F}_{17}$$

- We know that the Ω includes the public and private inputs along with the circuit inputs and outputs. We know that the $\Omega = [\omega^0, \omega^1, \omega^2, \dots, \omega^{15}]$. In this set, we will be taking only the first two values that is the public input. Now, from the given public inputs $\Omega_{inp} = \{\omega^{-1}, \omega^{-2}, \dots, \omega^{-|I_x|}\} \subseteq \Omega$.
- Using the Zero-test, the following will be tested to ensure that the T encodes the correct public inputs.

$$T(y) - S(y) = 0 \text{ for all } y \in \Omega_{inp}$$

We give the input to the polynomials $T(y)$ and $S(y)$ as y that is from the public input list Ω_{inp} and check whether the zero test holds or not. We already discussed about the zero test. If $T(y) - S(y) = 0$ then the T encodes all public inputs.

- For example, if we give the y as 5 for the $S(y)$ then it will produce 5 and the same will be produced by the $T(y)$, so become zero and the check satisfied.
- Here, the verifier get the $T(y)$ from the prover and compute the $S(y)$ by the known y and public parameters Ω .
- **Important Question: The committed polynomial is just an element then how the verifier can compute $T(y)$?. Given y by the verifier, the prover computes and send the $T(y)$ to the verifier that is a single value.**
- We can check it through the pairing as follows:

$$e(T - g^{T(y)}, g) \stackrel{?}{=} e(\pi_y, g^{t-y})$$

- We recall the KZG Commitments, given a polynomial $T(X)$:

$$T(X) = \sum_{i=0}^{n-1} a_i X^i$$

Here, a is the coefficients of the polynomial. The commitment is:

$$T = g^{T(t)} = g^{\sum_{i=0}^{n-1} a_i t^i}$$

The above commitment T is computed in the earlier steps and given to the verifier by prover.

- Now let us see the proof at y . We know from the KZG discussion, that we can derive the quotient polynomial as follows:

$$Q(X) = \frac{T(X) - T(y)}{X - y}, \quad \pi_y = g^{Q(t)}$$

In KZG commitments, the trusted setup gives powers of $g, g^t, g^{t^2}, g^{t^3}, \dots, g^{t^n}$, hence it is given as $g^{Q(t)}$

- By the definition of $Q(X)$, as we have seen earlier:

$$T(X) - T(y) = Q(X) \cdot (X - y)$$

Substitute $X = t$:

$$T(t) - T(y) = Q(t) \cdot (t - y)$$

Exponentiating with the generator g :

$$g^{T(t)-T(y)} = g^{Q(t) \cdot (t-y)} = (g^{Q(t)})^{t-y} = \pi_y^{t-y}$$

where $\pi = g^{Q(t)}$ as per our earlier discussions.

– Pairing Check

$$e(T - g^{T(y)}, g) \stackrel{?}{=} e(\pi_y, g^{t-y})$$

$$e(T/g^{T(y)}, g) \stackrel{?}{=} e(\pi_y, g^{t-y})$$

$$e(T \cdot g^{-T(y)}, g) \stackrel{?}{=} e(\pi_y, g^{t-y})$$

We know that the commitment T is $g^{T(t)}$, so

$$* \text{ LHS: } T/g^{T(y)} = T \cdot g^{-T(y)} = g^{T(t)} \cdot g^{-T(y)} = g^{T(t)-T(y)}.$$

We know that $T(t) - T(y) = Q(t) \cdot (t - y)$

So, we can write as follows:

$$g^{Q(t) \cdot (t-y)}$$

Now it is,

$$\text{LHS} = e(g^{Q(t) \cdot (t-y)}, g)$$

* RHS: We can write $e(\pi_y, g^{t-y})$ as follows as per the pairing properties:

$$e(\pi_y, g^{t-y}) = e(\pi_y^{t-y}, g)$$

We know that π_y is $g^{Q(t)}$, so we can write the equation as follows:

$$\text{RHS} = e(\pi_y^{t-y}, g) = e(g^{Q(t) \cdot (t-y)}, g)$$

– Now, LHS is equal to RHS so the proof will work if there is no modification in the inputs or polynomials or commitments.

– This ensures that the inputs are properly encoded in the circuit.

- Test 2: Next, let us (verifier) check whether all the gates are evaluated properly or not. We want to ensure that the addition gates are evaluated for the addition and multiplication gates are evaluated for the multiplication.

– Let us have the selective polynomial $S(X) \in F_p^{\leq d}[X]$ such that $l = 0, 1, \dots, |C| - 1$. Here, $|C|$ is the number of gates in the circuit.

* $S(\omega^{3l}) = 1$ if gate number l is an addition gate. This means, the polynomial S on the point ω^{3l} , will produce 1 as the output.

* $S(\omega^{3l}) = 0$ if gate number l is an multiplication gate. This means, the polynomial S on the point ω^{3l} , will produce 0 as the output.

– In case of our example, there are two addition gates and one multiplication gate. The figure 17 shows the selective polynomial output for each gate. That is in case of the addition gate, output is 1 and in case of multiplication gate, output is 0.

– The selective polynomial is just an encoding of the gates not any input.

inputs:	5, 6, 1	$S(X)$	
Gate 0 (ω^0):	5, 6, 11	1	(+)
Gate 1 (ω^3):	6, 1, 7	1	(+)
Gate 2 (ω^6):	11, 7, 77	0	(×)

Figure 17: Trace Table with Selective output[Copied from Prof. Dan Boneh Lecture]

- The prover wants to prove the following to the verifier.

$$S(y) \cdot [T(y) + T(\omega y)] + (1 - S(y)) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

for all $y \in \Omega_{gates} = 1, \omega^3, \omega^6, \omega^9, \dots, \omega^{3|C|-1}$. Here the value 3 in $3|C|$ comes because every circuit has two inputs and one output. So, three variables for each gate. The variable l refers the gate numbers.

- In the following, $T(y)$ is the left input and $T(\omega y)$ is the right input.

$$S(y) \cdot [T(y) + T(\omega y)] + (1 - S(y)) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

If we take the gate as Addition then $S(y) = 1$, then $(1 - S(y)) = 0$, so what happens is that

$$(1) \cdot [T(y) + T(\omega y)] + (1 - 1) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

$$(1) \cdot [T(y) + T(\omega y)] + (0) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

$$(1) \cdot [T(y) + T(\omega y)] = T(\omega^2 y)$$

$$[T(y) + T(\omega y)] = T(\omega^2 y)$$

We left with $[T(y) + T(\omega y)]$ that means with left and right input. The addition of the two inputs will be $T(\omega^2 y)$ as per the circuit and the trace table. Hence, the following holds.

$$T(\omega^2 y) = T(\omega^2 y)$$

If we take the gate as Addition then $S(y) = 0$, then $(1 - S(y)) = 1$, so what happens is that

$$(0) \cdot [T(y) + T(\omega y)] + (1 - 0) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

$$(1) \cdot T(y) \cdot T(\omega y) = T(\omega^2 y)$$

$$T(y) \cdot T(\omega y) = T(\omega^2 y)$$

We left with $[T(y) \cdot T(\omega y)]$ that means with left and right input. The multiplication of the two inputs will be $T(\omega^2 y)$ as per the circuit and the trace table. Hence, the following holds.

$$T(\omega^2 y) = T(\omega^2 y)$$

- Remember that $T(y)$ is the left input wire, $T(\omega y)$ is the right input wire and $T(\omega^2 y)$ is the output wire.
- **How the verifier knows $S(X)$? The $S(X)$ is completely public and predetermined once the circuit is known.**
- The verifier also gets the $(\omega^0, 1), (\omega^3, 0), (\omega^6, 1)$ and can interpolate the polynomial $S(X)$ as follows.

- * We have three gates: two addition gate and one multiplication gate. So, $(1, 1, 0)$.
- * The verifier knows $\omega = 3$ and also verifier knows $(\omega^0, 1) = (3^0, 1), (\omega^3, 1) = (3^3, 1), (\omega^6, 0) = (3^6, 0)$. Hence, the points are $(1, 1), (10, 1), (15, 0)$. Here, $3^6 = 729 \bmod 17 = 15$ and $\omega^3 = 3^3 = 27 \bmod 17 = 10$.
- * Using these points, the verifier can compute the selective polynomial using the Lagrange's Interpolation or FFT. The following is the $S(X)$ based on Lagrange's Interpolation.

$$S(X) = 8X^2 + 14X + 13$$

Refer to our earlier discussion in Groth16 for the lagrange's interpolation and compute the above polynomial. The coefficients are $\{8, 14, 13\}$. Now this has to be committed using the m_0, m_1, m_2 as we seen in the previous check.

$$S(t) = m_2^8 + m_1^{14} + m_0^{13}$$

From, toxic secret $t = 3$ and publicly known generator $g = 3$, the trusted party computes and give the following to both prover and verifier.

$$m_0 = g^{t^0} = 3^{3^0} = 3$$

$$m_1 = g^{t^1} = 3^{3^1} = 27 \bmod 17 = 10$$

$$m_2 = g^{t^2} = 3^{3^2} = 3^9 = 19683 \bmod 17 = 14$$

Now, apply this to the coefficients of $S(X)$ to get the commitment.

$$m_2^8 = 14^8 = 1475789056 \bmod 17 = 16$$

$$m_1^{14} = 10^{14} = 1e + 14 \bmod 17 = 8$$

$$m_0^{13} = 3^{13} = 1594323 \bmod 17 = 12$$

Now the, the commitment of the selective polynomial according to the equation

$$C_S = \prod_{i=0}^d (gt^i)^{s_i}$$

is

$$C_S = 16 \cdot 8 \cdot 12 = 1536 \bmod 17 = 6$$

Similar way, the prover also computes the C_S .

- Here, the verifier chooses y and send to the prover. Now, prover computes $T(y), T(\omega y)$ and $T(\omega^2 y)$.
- Now, the verifier having the computed $S(X)$ and its commitment C_S and prover given $T(y), T(\omega y)$ and $T(\omega^2 y)$, he/she can verify whether the following holds or not.

$$S(y) \cdot [T(y) + T(\omega y)] + (1 - S(y)) \cdot T(y) \cdot T(\omega y) \stackrel{?}{=} T(\omega^2 y)$$

with the Zero-Test

$$S(y) \cdot [T(y) + T(\omega y)] + (1 - S(y)) \cdot T(y) \cdot T(\omega y) - T(\omega^2 y) \stackrel{?}{=} 0$$

If holds then the proof is accepted otherwise rejected.

- This verification can also done using the pairing as follows:

$$\text{For Left input: } e\left(\frac{C_S}{g^{T(y)}}, g\right) \stackrel{?}{=} e(\pi_y, g^{t-y})$$

$$\text{For Right input: } e\left(\frac{C_S}{g^{T(\omega y)}}, g\right) \stackrel{?}{=} e(\pi_{\omega y}, g^{t-\omega y})$$

$$\text{For Output: } e\left(\frac{C_S}{g^{T(\omega^2 y)}}, g\right) \stackrel{?}{=} e(\pi_{\omega^2 y}, g^{t-\omega^2 y})$$

We already discussed about the pairing mapping in first test that is **Check Whether T encodes the correct input or not**. You may refer it.

- We can also say the verifier will first check the pairing equivalence to ensure that the sent $T(y)$, $T(\omega y)$ and $T(\omega^2 y)$ are right or not and then check the following condition holds or not.

$$S(y) \cdot [T(y) + T(\omega y)] + (1 - S(y)) \cdot T(y) \cdot T(\omega y) \stackrel{?}{=} T(\omega^2 y)$$

- Test 3: Check whether Wiring is implemented correctly: Here wiring means, the same number (may be the output of another gate) that is given as input in multiple gates.
 - Considering our example, the repeated inputs in the circuit are as follows:

$$6 = T(\omega^{-2}) = T(\omega^1) = T(\omega^3)$$

This means, the same value 6 is applied for $T(\omega^{-2})$, $T(\omega^1)$ and $T(\omega^3)$. Likewise for the following:

$$5 = T(\omega^{-1}) = T(\omega^0)$$

$$11 = T(\omega^2) = T(\omega^6)$$

$$1 = T(\omega^{-3}) = T(\omega^4)$$

- Now, let us define a polynomial $W : \Omega \rightarrow \Omega$ that implements the rotation. For example:

$$W(\omega^{-2}, \omega^1, \omega^3) = (\omega^1, \omega^3, \omega^{-2})$$

likewise

$$W(\omega^{-1}, \omega^0) = (\omega^0, \omega^{-1})$$

and so on for all the wires.

- Lemma: For all $y \in \Omega$, $T(y) = T(W(y))$ then all wires satisfied. This test we can do using the prescribed permutation check that we have discussed already.
- The verifier will send (r, s) pair or the (α, γ) pair to the prover. Then the prover will compute the Z and send to the verifier along with the following:

- * commitments C_f, C_g, C_W

- * $f(a)$

- * $g(a)$

- * $W(a)$

- * Z

- * $a \in \Omega$, where Ω is the publicly known set. We already discussed how to compute the f , g and W in the prescribed permutation check section.

- * **Important Note: How to compute the $W(a)$? We have seen the $f(a), g(a)$ in the prescribed permutation check but what about the $W(a)$? The answer is,**

We know that $W(\omega^0) = \omega^2, W(\omega^1) = \omega^0, W(\omega^2) = \omega^1$ and take ω as a then it is $W(a^0) = a^2, W(a^1) = a^0, W(a^2) = a^1$.

Now consider these points a_0, a_1, a_2 and apply the lagrange's interpolation and compute the polynomial. Later, compute the commitments as we discussed in the KZG commitment.

Likewise the polynomial $f(\cdot)$: left input and $g(\cdot)$: right input can be computed. For example, f-values: $f(\omega_0) = 5, f(\omega_1) = 7, f(\omega_2) = 3$ then we can use these points and compute the polynomial f using the lagrange's interpolation. We know that from W , we get g so we can compute the polynomial.

- * $f(X)$ is a polynomial encoding the left input wire values for each gate.

- * $g(X)$ is a polynomial encoding the right input wire values for each gate.

- * W is a permutation that tells how values in f should match values in g (or across other wires) to enforce copy constraints.

On receiving the above, the verifier will compute the following and check whether the received Z is equal to the computed Z . If both matches then accepted as the wires are computed properly otherwise there is some modification hence get rejected.

$$Z = \prod_{a \in \Omega} \frac{r - s \cdot W(a) - g(a)}{r - s \cdot a - g(a)} = 1$$

- In case of the pairing, we have to perform the correctness of polynomial commitment along with the permutation and product check:

$$e(C - g^{f(r)}, g) = e(\pi, g \cdot (r - X))$$

where:

- * C is the commitment to the polynomial $f(X)$ sent by the prover,
- * r is the evaluation point chosen by the verifier,
- * $f(r)$ is the claimed evaluation of the polynomial at r ,
- * π is the opening proof provided by the prover. We have already seen how to compute the π value.
- * g is the generator of the pairing groups. If group $G_1 \neq G_2$, then there will be two generators: g_1 and g_2 (For example, the g_1 and g_2 will be applied as $e(g_1, g_2)$). If group $G_1 = G_2$, then there will be only one generator g .

This pairing check ensures that the committed polynomial actually evaluates to $f(r)$ at the point r . Note that the permutation and product check itself is done over field elements and does not require pairings; pairings are only needed to verify the correctness of polynomial commitments.

- **The permutation check is required that if a value is used in multiple gates, we need to make sure that the same value appears in all wires without disclosing the individual wires.**
- Now the final test that is the output of the final gate is 0 or the expected value. In our case, it should produce the output as 77 for the private (1) and public inputs (5, 6).
 - We know $|C|$ is the number of gates in the circuit and the last gate is $|C| - 1$.
 - The polynomial $T(X)$ encodes all wire values (left inputs and right inputs).
 - The final output corresponds to the last wire. The last wire is at $\omega^{3|C|-1}$.
 - The verifier knows the commitment C_T of the polynomial $T(X)$.
 - The verifier chooses the evaluation point corresponding to the output wire:

$$X_{\text{out}} = \omega^{|C|-1}$$

The value $\omega^{|C|-1}$ is known to the verifier since $\Omega = \{\omega^1, \omega^2, \dots, \omega^{|C|-1}\}$ is public.

- The verifier asks the prover to open $T(X)$ at X_{out} , i.e., provide $T(X_{\text{out}})$ along with the opening proof.
- The verifier checks:

$$T(X_{\text{out}}) \stackrel{?}{=} 0$$

According to our example,

$$T(X_{\text{out}}) \stackrel{?}{=} 77$$

If this holds, the final output of the circuit is correct. This does not need the pairing check. If $T(X)$ is committed using KZG commitments, the verifier checks the opening proof at the point X_{out} using a pairing as follows:

$$e(C_T - g^{T(X_{\text{out}})}, g) = e(\pi, g \cdot (X_{\text{out}} - X))$$

This verifier can check this equality as we have done earlier. However, only for verifying the commitment not the final output check. This pairing check is required otherwise prover will simply give any output value and claim as the actual output.

- Finally, all checks are done. If all passes then the prover successfully prove to the verifier.
- The above discussions are on the interactive mode that is every time, the verifier has to send y . To make it, non-interactive, the prover can use the Fiat-Shamir's transformation and compute y using the public parameters and send it to the verifier. The verifier can also compute y using the public parameters and use it for all verifications. For example, the y can be computed as follows:

$$y = \text{Hash}(C_T || C_S || \text{circuit description}) \text{ mod } 17$$

Here C_T is the commitment of the circuit polynomial $T(X)$ and C_S is the commitment of the selective polynomial in the case of verifying whether all gates are computed properly or not (that is for the second test). The *circuit description* can be like how many addition and multiplication gates. However, the same parameters has to be communicated to the verifier so that the verifier will also compute y . Like this, required y 's can be computed.

- We have computed as g^t or g^s or g^a . However, if we consider the elliptic curve then it should be the multiplication of scalar value with the curve points using the point addition and point doubling. For example, if G is the base point of the sub group then it should be $G \cdot t$ or $G \cdot s$ or $G \cdot a$.

10.10 Affine Coordinates

A point on an elliptic curve is represented as a simple pair or we can say in 2 Dimenison:

$$(x, y)$$

Now, if we consider two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the elliptic curve $y^2 = x^3 + ax + b$ The slope λ is defined as:

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2, \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2. \end{cases}$$

Then the new point $P_3 = (x_3, y_3)$ resulting from addition is:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1.$$

Important Note: The division in λ , e.g. $\frac{y_2 - y_1}{x_2 - x_1}$, is expensive in modular arithmetic, especially modulo a large prime (while computing the modulo inverse it will be computationally heavy, $x/y \text{ mod } p = x \cdot y^{-1} \text{ mod } p$).

10.11 Projective Coordinates

A point is represented in three coordinates. We can represent the above elliptic curve point in projective coordinate as follows:

$(X : Y : Z)$ When $Z = 1$, the point is already in "normalized" form, and $(X : Y : Z) = (x, y, 1)$

When $Z \neq 1$, the point is not normalized and we have to divide by Z to get the actual affine coordinates $x = X/Z, y = Y/Z$.

A point in projective coordinates is represented as

$$P = (X : Y : Z)$$

corresponding to affine coordinates

$$(x, y) = \left(\frac{X}{Z}, \frac{Y}{Z} \right).$$

The elliptic curve equation in projective coordinates becomes

$$Y^2 Z = X^3 + aXZ^2 + bZ^3.$$

Point Doubling in Projective Coordinates

For a point $P = (X_1, Y_1, Z_1)$, the doubling formula is:

$$\begin{aligned}A &= X_1^2, \\B &= Y_1^2, \\C &= B^2, \\D &= 2((X_1 + B)^2 - A - C), \\E &= 3A + aZ_1^4, \\F &= E^2, \\X_3 &= F - 2D, \\Y_3 &= E(D - X_3) - 8C, \\Z_3 &= 2Y_1Z_1.\end{aligned}$$

Point Addition in Projective Coordinates

For points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$, the addition formula is:

$$\begin{aligned}U_1 &= X_1Z_2, & U_2 &= X_2Z_1, \\S_1 &= Y_1Z_2, & S_2 &= Y_2Z_1, \\H &= U_2 - U_1, & R &= S_2 - S_1, \\X_3 &= R^2 - H^3 - 2U_1H^2, \\Y_3 &= R(U_1H^2 - X_3) - S_1H^3, \\Z_3 &= HZ_1Z_2.\end{aligned}$$

Note: No division is needed; only multiplications, squarings, and additions means no need to compute the inverse and without it we get the equivalent output. In all the computations, we need to apply the modulo. You can try for $(x_1, y_1) = (5, 1)$, $(x_2, y_2) = (6, 3)$ and $Z = 1$. You can verify your results with our point addition and doubling example in elliptic curve example section 5.7.

In the implementation of zk-SNARK, it will be represented in the form of project coordinate to make the process fast.

10.12 zk-STARK

The zero-knowledge Scalable Transparent Argument of Knowledge (zk-STARK) is the quantum secure algorithm since the proof is based on hashing. The feature of zk-STARK is as follows:

- Scalable: Can handle large computations efficiently.
- Transparent: No trusted setup is required.
- Argument of Knowledge: The prover demonstrates they know a valid witness (refers to argument of knowledge) without revealing it (referring to Zero-Knowledge).
- In addition, it ensures the soundness and completeness properties.

The zk-STARK uses the Fast Reed-Solomon, Merkle Tree and Fiat-Shamir's Transformation. We discuss each concept first and then will move on to the zk-STARK implementation.

10.12.1 Fast Reed-Solomon Interactive Oracle Proof (FRI)

The is somewhat similar to the Reed-Solomon code but the full vector or data will not be revealed. We see the process of FRI based on prover and verifier side.

Prover Computation based on Interaction

- Step 1: Let prover choose a finite field F_p . For example $F_7 = \{0, 1, 2, 3, 4, 5, 6\}$.
- Step 2: Choose the domain points that is the input points ($D = \{0, 1, 2, 3\}$). The length $n = 4$.
- Step 3: Consider that, prover wants to claim that f is an evaluation of a polynomial of degree ≤ 1 ($d = 1$) over the input points (D).
- Step 3.1: Choose a polynomial $P(x)$ with degree d and evaluate the $P(x)$ for every value in D ($\forall \in D$). The output is the f . We assume that we have a polynomial $P(x)$ and we apply the values of $D = \{0, 1, 2, 3\}$ as x and take the output as f . For example, let us say the $P(x) = x \pmod{7}$ and first we take $x = 0$ from the D and apply then $P(0) = 0 \pmod{7} = 0$, now $f = [0]$, next take $x = 2$ from D then $P(2) = 2 \pmod{7} = 2$, now $f = [0, 2]$. Next take $x = 3$ from D then $P(3) = 3 \pmod{7} = 3$, now $f = [0, 2, 3]$ and finally take $x = 1$ from D then $P(1) = 1 \pmod{7} = 1$, now $f = [0, 2, 3, 1]$
- Step 4: We consider f as a vector. For example $f = [2, 3, 0, 1]$.
- Step 5: Now, split f into even and odd vectors. In our example:
 - Step 5.1: Even indices: $f_{\text{even}} = [f_0, f_2] = [2, 0]$
 - Step 5.2: Odd indices: $f_{\text{odd}} = [f_1, f_3] = [3, 1]$
- Step 6: Verifier picks random challenge: $r_1 = 2 \in F_7$ and give it to the prover.
- Step 7: Now, prover uses the r_1 and computes the proof in rounds:
- Step 8: First Round

- For each index i ,

$$f^{(1)}[i] = f_{\text{even}}[i] + r_1 \cdot f_{\text{odd}}[i] \pmod{7}.$$

- Using the above, compute the following for each i . Here i is 2 since we have two values in both f_{even} and f_{odd} :

$$f^{(1)}[0] = 2 + 2 \times 3 = 2 + 6 = 8 \equiv 1 \pmod{7},$$

$$f^{(1)}[1] = 0 + 2 \times 1 = 0 + 2 = 2 \pmod{7}.$$

- From $f = [0, 2, 3, 1]$ or $f^{(0)} = [0, 2, 3, 1]$, we came to the first round vector:

$$f^{(1)} = [1, 2].$$

- Step 9: We have given in step 3 that the prover want to evaluate f with degree ≤ 1 however here the vector has two values. That is Vector length = 2, still larger than degree bound $d = 1$. So, now let us go for the second round

- Step 9.1 1: Again, we have to Split even/odd based on index:

$$f_{\text{even}} = [1], \quad f_{\text{odd}} = [2].$$

- Step 9.2: Verifier picks random challenge for this round:

$$r_2 = 3.$$

- Step 9.3: Like in first round, let us do the folding as follows:

$$f^{(2)}[0] = 1 + 3 \times 2 = 1 + 6 = 7 \equiv 0 \pmod{7}.$$

- Second Round Vector

$$f^{(2)} = [0].$$

- Length = 1, trivially low-degree that is equal to d (which is 1). Hence no need to further reduce it.

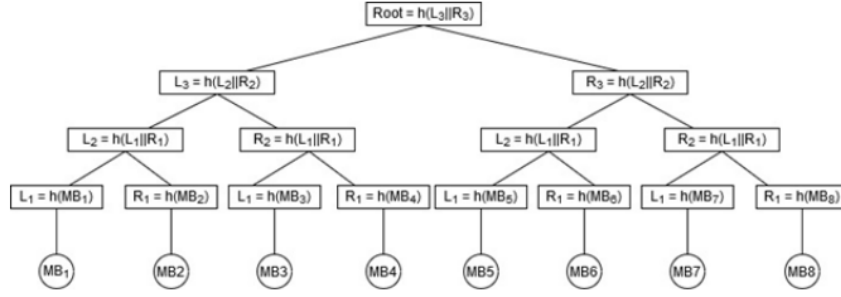


Figure 18: Merkle Tree

Verifier check based on Interactive Oracle Proof (IOP)

The verifier knows the F_p as well as vectors (initial vector, first and second round vector given by the prover). The initial round vector is $[2, 3, 0, 1]$, first round vector is $[1, 2]$ and second or final vector is $[0]$. Now, the verifier picks a random point in the final vector (only one here, that is 0) and follows preimages $[1, 2]$ through the folding steps and verify the proof. The verifier in the example given two random values r_1 and r_2 (refer step 7 and 9.2) to the prover. Now the verifier checks as follows:

- Check the consistency in first round, by taking the first $[1, 2]$ and second round vector $[0]$ values along with the verifier generated random value r_2 . Here, for verifier it is first round, however the computation is happening like in the second round of prover.

$$f^{(2)}[0] \stackrel{?}{=} f^{(1)}[0] + r_2 \cdot f^{(1)}[1] = 1 + 3 \times 2 = 1 + 6 = 7 \equiv 0 \pmod{7} \Rightarrow \text{It matches}$$

Here, the verifier know already that $f^{(2)}[0] = 0$, since it is given by the prover. Now, the computed $f^{(1)}[0] + r_2 \cdot f^{(1)}[1]$ is matching so verification is successful.

- Check the consistency in second round, by taking the initial $[2, 3, 0, 1]$ and first round vector $[1, 2]$ values along with the verifier generated random value r_1 . Here, for verifier it is second round, however the computation is happening like in the first round of prover.

$$f^{(1)}[0] \stackrel{?}{=} f_{\text{even}}[0] + r_1 \cdot f_{\text{odd}}[0] = 2 + 2 \times 3 = 8 \equiv 1 \pmod{7} \Rightarrow \text{It matches}$$

$$f^{(1)}[1] \stackrel{?}{=} f_{\text{even}}[1] + r_1 \cdot f_{\text{odd}}[1] = 0 + 2 \times 1 = 2 \pmod{7} \Rightarrow \text{It matches}$$

- The verifier is convinced that f is close to a polynomial of degree ≤ 1 .
- The verifier is convinced that the f is right polynomial.

Important Note 1: The verifier may go for all the rounds or only some rounds. In the above example, there are two rounds, may the verifier will go for first round not for the second round.

Important Note 2: The FRI is interactive; however, we need the non-interactive when we go for the Blockchain applications.

10.12.2 Merkle Tree

In cryptography, a Merkle tree or hash tree is a tree, where leaf nodes are a hash of the data block and non-leaf nodes are a hash of its child nodes. The concept of a hash tree is named after Ralph Merkle who patented it in 1979 (Merkle Tree). The sample Merkle tree is shown in figure 18. The leaf nodes (L1 and R1) of Merkle tree are hash (h) value of the data blocks (MB1, MB2, MB3. . .) and non-leaf nodes (L2 and R2, L3 and R3and Root) are the cryptographic hash (SHA256 or SHA512 or others) values of its child nodes. The root hash of the tree will be shared to manage the integrity of the entire tree data without keeping the complete tree.

The hash function used in the Merkle tree is a cryptographic hash function. The Merkle tree can be used to verify any kind of data stored, handled, and transferred in and between peer network nodes are undamaged and unaltered. Merkle tree is used in blockchain to store transactions, account balances, etc.

10.12.3 Membership Verification of Merkle Tree

This is to ensure whether the given data is member of the merkle tree or not. In any merkle tree, it can be identified through root and path hash. It has an advantage, verifier need not to have complete merkle tree to ensure the membership of data, only the root hash is enough. For example, a person \mathcal{A} contains the root node or root hash of the Merkle tree for set \mathcal{S} , and person \mathcal{B} wants to convince person \mathcal{A} that element e is in \mathcal{S} . To do this, person \mathcal{B} only have to provide person \mathcal{A} with the siblings of all of the elements in the tree in the path from e to the root node. This is only $\log(n)$ nodes. So \mathcal{B} only has to provide only $\log(n)$ pieces of data, and person \mathcal{A} only has to store the root node. With this provided information, person \mathcal{A} can recompute the root node of the tree, and check to make sure that it matches the one, it is correct and member of the tree. If the hash function is collision resistant (chance of getting same hash output for two different messages will be very complex), then he can be sure that e is in \mathcal{S} . For better understanding, we can discuss with a sample tree shown in figure 19.

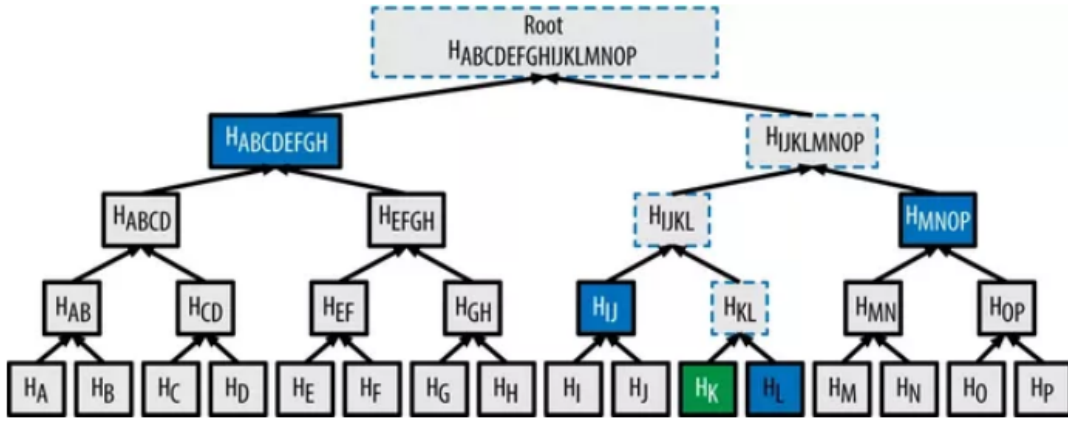


Figure 19: Merkle Tree [Copied]

If person \mathcal{A} holds $RH = H(abcdefghijklmnop)$ and person \mathcal{B} want to prove that the value k is part of the merkle tree. The person \mathcal{A} can ensure the membership without knowing k that is with $H(k)$ it can be proved. To verify the membership of $H(k)$, now along with $H(k)$, $H(l)$, $H(ij)$, $H(hmnop)$ and $H(abcdefgh)$ are required. With available $H(k)$ and $H(l)$, $H(kl)$ can be computed. Likewise, with $H(kl)$ and $H(ij)$, $H(klij)$ can be computed and using $H(klij)$ and $H(hmnop)$, $H(klijhmnop)$ can be computed. Finally, with $H(abcdefgh)$ and $H(klijhmnop)$, $H(abcdefghijklmnop)$ can be computed. The person \mathcal{A} can verify the membership of k in the merkle tree through equation 60 using the available root hash ($rh = H(abcdefghijklmnop)$) and the computed root hash ($ch = H(abcdefghijklmnop)$).

$$Decision = \begin{cases} Member & \text{If } rh == ch \\ Not a Member & \text{otherwise} \end{cases} \quad (60)$$

In certain cases, k will be given to the verifier and from that $h(k)$ will be computed by the verifier and then check the membership as per the above discussion.

How the Merkle Tree is used in the zk-STARK? We will discuss the use of merkle tree in zk-STARK in the following steps:

- We have taken f in the example of FRI. Let us consider those as the data (vector as the data) and compute merkle tree (MK_1) leaf using those vector values. So that, we will get the Root R_1 of merkle tree MK_1 .

- We know $f = [0, 2, 3, 1]$ and compute hash (h) of each element individually. $l_1 = h(0), r_1 = h(2), l_2 = h(3), r_1 = h(1)$. The l and r respectively refers the left and right child of Node 1 and 2. Here, l_1, l_2, r_1, r_2 are leafs of the merkle tree.
- Now, using these leafs let us bring the next upper level node as follows:
 - * $l_{11} = h(l_1||r_1)$
 - * $r_{22} = h(l_2||r_2)$
- Now, using these values, let us bring the next upper level node that will be the root (R_1) of merkle tree 1.
 - * $Root(R_1) = h(l_{11}||r_{22})$
- Likewise it will be computed for the first round f values which is $[1, 2]$. Here 1 and 2 are the data and from that we can derive the leaf of the second merkle tree MK_2 .
 - * $l_1 = h(1)$
 - * $r_1 = h(2)$
- From the computed l_1 and r_1 , we will compute the next upper level node that will be the root (R_2) of MK_2 .
 - * $Root(R_2) = h(l_1||r_1)$
- Likewise it will be computed for the second round f values which is $[0]$. Here 0 is the data and from that we can derive the leaf of the third merkle tree MK_3 .
 - * $l_1 = h(1)$
 - * We have only one value so there is not right child and the leaf will be the Root hash or Root (R_3)
 - * $R_3 = l_1 = h(1)$
- Prover sends R_1, R_2 and R_3 to the verifier.

- Now, the verifier want to verify the claim of the prover.

- The verifier picks a random index i from the folded vector $f^{(1)} = [1, 2]$ to check consistency. Suppose the verifier chooses $i = 0$. The prover reveals:
- The value $f^{(1)}[0] = 1$ and the Merkle proof for this leaf in the Merkle tree of $f^{(1)}$ that is MK_2 :

Proof: Sibling hash r_1 such that $R_2 = h(h(1)||r_1)$

Here, the verifier knows 1 and from that the $l_1 = h(1)$ can be derived. Now, the prover has given r_1 . Now computing the hash of l_1 and r_1 by concatenating these, we get the $R_2 = h(l_1||r_1)$. It will match that means computed R_2 and prover sent R_2 will match if no modification on the vector and merkle tree otherwise it will not match. Let us consider, it matches here.

- The original values from $f^{(0)}$ that map to this folded index $f^{(1)}[0] = 0$ is $f^{(0)}[0] = 2$ and $f^{(0)}[1] = 2$. Now, verify its proof in the merkle tree MK_1 . Now, the verifier asks for the MK_2 path variables to verify the root hash. On the request of verifier, the prover gives r_1 and r_2 to the verifier.
- Now, by knowing r_1, r_2 and values of $f^{(0)}[0] = 2$ and $f^{(0)}[1] = 2$, the verifier compute the Root hash (R_1) of merkle tree MK_1 . It is computed as follows:

$$R_1 = h(h(h(2)||r_1)||h(h(3)||r_2))$$

- Now, the computed R_1 from the above and the sent R_1 by prover will be matched. If matches then the merkle proof is valid otherwise invalid.

- Once the verifier trusts the values through the Merkle proofs, he/she check the folding equation:

$$f^{(1)}[0] \stackrel{?}{=} f^{(0)}[0] + r_1 \cdot f^{(0)}[1] \pmod{7}$$

- Substituting the values that are shared by the prover that is $f^{(1)} = [1, 2]$ and $f^{(0)} = [2, 3, 0, 1]$. Using these values only the merkle proofs are validated. Now, the verifier validate the fold. Here, $f^{(1)}[0] = 1$, $f^{(0)}[0] = 2$ and $f^{(0)}[1] = 3$.

$$1 \stackrel{?}{=} 2 + 2 \cdot 3 = 2 + 6 = 8 \equiv 1 \pmod{7}$$

- The equation holds, so this index is consistent with the low-degree claim.
- Repeating this process for other sampled indices (for example, checking the $f^{(1)}[1]$) and folding rounds (for example, $f^{(2)}[0]$) allows the verifier to be confident that f corresponds to a polynomial of degree $\leq d$.

Important Note: The usage of Merkle is to achieve the succinct that is instead of sending all the proofs we need to send only required on demand.

10.12.4 Verkle Tree and Vector Commitment

Like Merkle Tree, Verkle Tree is a cryptographic data structure to ensure the data integrity. In case of Merkle Tree, we hash the data and take that as child and concatenate the two child in case of binary tree and compute the next high level node and repeat the process till the root. The root hash will be used for verifying the integrity means prover will share it with the verifier. In case of verifying the integrity of any message block (hash of the message block may be part of the merkle tree or not), no need to send all the nodes hash values, only the path nodes' hash values to be sent to the verifier. Multiple hash values (each hash value size is 256 bits or more based on the hash algorithm) need to be shared with the verifier to prove the integrity. To overcome sharing more bits or storing more bits, the Verkle tree is explored in the cryptocurrency implementation.

For example, the prover computes the merkle tree and keeps all the nodes hash (each is of 256 bits for example) in its storage and share the path hash values (normally $\log_2 N$ hash values, where N is the total message block or data and base 2 in case binary merkle tree otherwise according the maximum number of child allowed. If we take three out degree or child then it is $\log_3 N$ hash values) for a data to verifier to prove the integrity of that data. The prover need to store all nodes hash and path hash values need to be shared every time to prove the integrity of the data. The size of the hash value increase the complexity of storage and sharing hence the Verkle tree is explored.

The verkle tree uses the polynomial commitments instead of the hash values. The hash value computation is faster than the polynomial commitment however, that computation is one time for any circuit (in case of zk-STARK or zk-SNARK) hence it is explored. The implementation process of the verkle tree is as follows:

- First, we need to have the message block ($MB_1, MB_2 \dots MB_N$) that has to take part in the verkle tree.
- Now, for every message block compute the polynomial commitment and the opening proof. We know that the message block is a value, so let us consider that as the constant polynomial.

$$P(x) = 5; \text{ where } 5 \text{ is } MB_1$$

The polynomial commitment and the opening proof π as per the KZG polynomial commitment we discussed in section 10.5.1 is as follows.

$$CP = g^{P(x)}$$

$$\pi = g^{Q(x)}$$

Now, for the next level parent node, in case of figure 20 there are three child, so three child commitment will be used for computing the parent polynomial commitment. For example, consider following are the polynomial commitments and the opening proof of all three child nodes:

$$CP_1 = g^{P_1(x)}, CP_2 = g^{P_2(x)}, CP_3 = g^{P_3(x)}$$

$$\pi_1 = g^{Q_1(x)}, \pi_2 = g^{Q_2(x)}, \pi_3 = g^{Q_3(x)}$$

- Use the child commitment and create the vector

$$v = [CP_1, CP_2, CP_3]$$

- Now the values in the vector will be used to create a polynomial using the lagrange's interpolation (We already discussed the construction of polynomial from points using the lagrange's interpolation). The points are

$$[(0, CP_1), (1, CP_2), (2, CP_3)]$$

The values 0,1,2 are the x values. We can also choose other values, for example, 3,4,5 but in all the computation same should be used. Using these points, we will bring the polynomial $P_{11}(x)$.

- Now, use the polynomial $P_{11}(x)$ to compute the commitment which is also called as the vector commitment. The commitment and the opening proof computation is as follows:

$$CP_{11} = g^{P_{11}(x)}$$

$$\pi_{11} = g^{Q_{11}(x)}$$

- This process will be repeated until the root. Figure 20 can be referred for the computation. The root commitment is the digest of the verkle tree. This root commitment will be for the integrity check along with that path commitments.

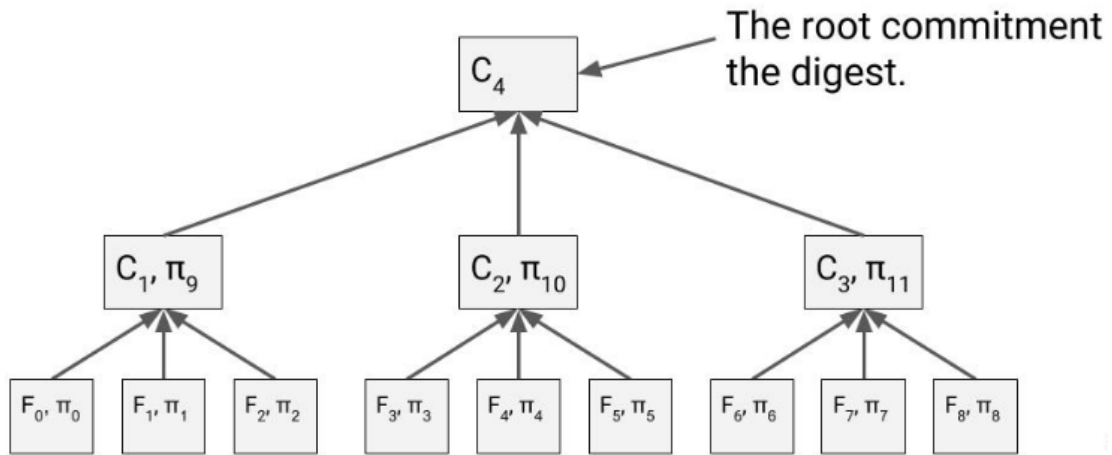


Figure 20: Verkle Tree [Copied]

Verification: Now, the verifier want to verify the integrity of a message block. The process of verification is as follows:

- Let us consider, the verifier want to verify the message block MB_3 or F_2 according to figure 20.
- The prover will provide the message block along with that commitments and the opening proof of the siblings. Those are CF_0, CF_1 and π_0, π_1 .
- In addition, according the figure 20, the commitment C_2, C_3 and π_{10}, π_{11} will be given.
- Also, get the root commitment/digest C_4 from the prover.
- Now, using the F_2 , the verifier can compute the CF_2 and the opening proof π_2 .
- Using the computed CF_2 and received CF_0 and CF_1 , the vector commitment C_1 can be computed.
- Using the computed C_1 and received C_2 and C_3 , the C_4 , which is the root digest can be computed. Now compare the received root digest and computed root digest, if both match that means the received message block F_2 is correct otherwise modified.

- **Important Note:** If we see here, the opening proof is not required in the verification. However, based on requirement these can be shared and verified. For example, if required to verify at each level of the tree then the opening proof can be used.

Pairing based Verification In the above, we have shown that the root commitments will be compared to verify the integrity of the message block F_2 . However, to ensure the integrity, pairing is used in case of KZG commitment. As we have seen in the verification part, pairing is not required. **Then what is the need of pairing?** Let us consider the case, each leaf of a verkle tree is a vector with multiple values instead of a single value that we considered in our earlier discussion. For example,

$$v_f = [v_0, v_1, v_2 \dots v_n]$$

Likewise, every leaf is a vector with same or different length. The leaf commitment is the **vector commitment** and the opening proof is the vector opening proof. Now, the verifier want to verify the value for example, v_0 which is part of v_f . To verify the v_0 , receiver needs all other values of v_f along with leaf node siblings commitment also the path nodes commitment. However, receiver will not get all values of v_f . In this case, the verifier can verify the integrity of the v_f as discussed in the verification part but not sure whether the prover given v_0 is part of v_f . Hence, the pairing based verification is used to verify the v_0 and also we need the opening proof π . This verification is only to verify the value v_0 not any other internal nodes. The verification equation is as follows:

$$e\left(\frac{C}{g^{v_i}}, g\right) \stackrel{?}{=} e(\pi_i, g^{s-i})$$

We know the polynomial commitment $C = g^{P(x)}$

$$e\left(\frac{g^{P(x)}}{g^{v_i}}, g\right) \stackrel{?}{=} e(\pi_i, g^{s-i})$$

Now, we can bring the denominator to the numerator in LHS since the base of both the numerator and denominator are same that is g .

$$e(g^{P(x)-v_i}, g) \stackrel{?}{=} e(\pi_i, g^{s-i})$$

We know the opening proof $\pi = g^{Q(x)}$, so we expand the RHS as follows.

$$e(g^{P(x)-v_i}, g) \stackrel{?}{=} e(g^{Q(x)}, g^{s-i})$$

Using the bilinear property, let us bring out the powers or exponent from inside to outside in RHS.

$$e(g^{P(x)-v_i}, g) \stackrel{?}{=} e(g, g)^{Q(x) \cdot s - i}$$

As, we have seen earlier for example in KZG discussion, we can write $P(x) - v_i = Q(x) \cdot s - i$. Also, bring the exponent outer like in previous step. So,

$$e(g, g)^{Q(x) \cdot s - i} \stackrel{?}{=} e(g, g)^{Q(x) \cdot s - i}$$

If the verifier able to see both the values are same then he/she accept the message otherwise reject it. Here,

- C is the commitment of the leaf, which is the vector v_f .
- v_i is the value for which integrity to be checked. In our case, it is v_0 .
- π_i is the opening proof of the leaf vector v_f polynomial.
- i is the vector index for the value v_0 .
- g is the generator of the group.

- s is the secret. This secret will not be given to the prover or verifier. It is only for the trusted setup. However, this will come to both the prover and verifier as the common reference string or Structured Reference String as we discussed earlier in the zk-SNARK and KZG commitments.
- Receiver will be having the values to verify the equality.

After successful verification of this pairing to ensure v_0 membership in the vector, the root polynomial commitment has to be verified as discussed in the verification part to ensure that the prover is not giving different value and vector. This means, the prover may produce different verkle tree and different vector. To ensure that the value is part of the vector and vector is member of the verkle tree, the verifier need to perform the pairing based verification for the leaf (value in the vector) that is to verify value is a member of the vector and verification of the root polynomial commitment using the path nodes polynomial commitment to verify the vector is a member of the verkle tree. If both verification are succeeds then the verifier will accept the value from the prover otherwise will reject it. This discussion is based on the KZG based vector commitment.

Important Note: Here, we use $g^{P(x)}$ but we already discussed that if the elliptic curve is used then it will be the $g \cdot P(x)$, where g is the base point and $P(x)$ are the scalar coefficient of that polynomial. Refer KZG for the computation.

The computation cost of the Verkle tree is higher than the Merkle tree however size is lesser than the Merkle tree. We can use either Merkle or Verkle tree based on the requirement.

10.12.5 Fiat-Shamir's Transformation

The Fiat-Shamir (Amos Fiat and Adi Shamir (1986)) transformation is a method to convert an interactive proof protocol into a non-interactive one by replacing the Verifier's random challenge with a cryptographic hash. For example, we have seen in the previous section that each time, the prover needs a random from the verifier thus needs the interaction. In case of cryptocurrency application, it is not feasible and we need the non-interactive proof. Hence, the Fiat-Shamir's transformation is used. We see a generic example Fiat-Shamir's transformation from Wikipedia:

Consider that Peggy wants to prove to some one that she knows v such that $y \equiv g^v$ without revealing v . Here the modulus should come however we are not giving it for simplicity.

1. Peggy picks a random $v \in \mathbb{Z}_q^*$ (here \mathbb{Z}_q^* is the multiplicative integer field) and computes

$$t = g^v.$$

2. Peggy computes

$$c = H(g, y, t),$$

where H is a cryptographic hash function.

3. Peggy computes

$$r = v - cx \pmod q; \text{ In this statement, the modulus is considered however it is part of all steps.}$$

The resulting proof is the pair (t, r) .

4. Anyone can use this proof to calculate c and check whether

$$t \equiv g^r y^c \pmod q$$

5. Since the verifier knows t shared by Peggy from step 3 and publicly known g and y any one can compute c . Means, the verifier can compute $H(g, y, t)$. **Important Note: For the hash function, the input should be exactly same means we have to give the input as g, y, t order not like t, g, y because hash algorithm will not support commutative property.**

In case of the zk-STARK how the Fiat-Shamir's transformation applied is as follows:

- Instead of getting the random values r_1, r_2 and so on from verifier, the prover computes it for the specific round by computing the hash.
- The first challenge or random value $r_1 = H(\text{MerkleRoot})$ or we can say $r_1 = H(R_1)$ for our FRI example.
- The second challenge or random value $r_2 = H(\text{MerkleRoot}||\text{response}_1)$ or we can say $r_2 = H(R_1||\text{response}_1)$ or $r_2 = H(r_1||\text{response}_1)$ for our FRI example. Here, the “response 1” is exactly the data the prover would have sent to the verifier in the first round of an interactive protocol. This means, the prover computes the low degree vector considering the r_1 and send to the verifier. Now that is the response_1 used in the next round. Likewise, it will be concatenated in the next round. Let us say, if it is third round then it will be like $r_3 = H(R_1||\text{response}_1||\text{response}_2)$ or $r_3 = H(r_2||\text{response}_2)$

10.12.6 Implementation Process

We will see the complete zk-STARTK process in the following steps with two parties: Prover and Verifier.

Prover

- The prover want to prove to the verifier that he/she knows the secret input that is $x = 1$ for example.
- Now, create a low degree polynomial considering the finite field F_p so that the input can be applied to prove. For example,

$$P(x) = x^3 + 2x^2 + x - 4 \Rightarrow 1^3 + 2(1^2) + 1 - 4 = 3 - 4 \text{ mod } 17 = 0$$

- Next, prover compute the trace by taking the value of $x = 1$, that is the secret input.

Step	Operation	Value
t_0	x	1
t_1	x^2	1
t_2	$2 \cdot x^2$	2
t_3	x^3	1
t_4	$x^3 + 2x^2$	3
t_5	$x^3 + 2x^2 + x$	4
t_6	$x^3 + 2x^2 + x - 4$	0

- Now, there are 7 traces $T = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6\} = \{1, 1, 2, 1, 3, 4, 0\}$. Here we can say t_0 and t_6 are the boundary checks or boundary trace because t_0 is the initial and t_6 is the final output of the polynomial.
- Let us create the coordinate points like we have done in zk-SNARK: $\{(0, 1), (1, 1), (2, 2), (3, 1), (4, 3), (5, 4), (6, 0)\}$.
- Now, apply the lagranges interpolation and create a polynomial from the points. For each point (x_k, a_k) , the **Lagrange basis polynomial** $L_k(x)$ is defined as follows:

$$L_k(x) = \prod_{\substack{m=0 \\ m \neq k}}^6 \frac{x - x_m}{x_k - x_m} \pmod{17}$$

where:

- x_m are the sequence numbers 0, 1, 2, 3, 4, 5, 6
- a_k are the trace values at those steps that are 1, 1, 2, 1, 3, 4, 0

Then, the **interpolation polynomial** $f_0(x)$ is:

$$f_0(x) = \sum_{k=0}^6 a_k L_k(x) \pmod{17}$$

This is similar to how we performed the lagranges interpolation in zk-SNARK (groth16).

- The constructed polynomial will pass through all seven points (seven traces) and the degree of the polynomial will be $7 - 1 = 6$. We already discussed about the lagranges interpolation based polynomial construction. The lagranges interpolation means constructing polynomials from the given points (Interpolation is the process of estimating or constructing a function that passes through a given set of points). The constructed polynomial is $f_0(x)$. Here we consider $p = 17$ so $F_p = F_{17}$ and the constructed polynomial as follows:

$$f_0(X) = 3X^6 + 8X^5 + 7X^4 + 13X^3 + 10X^2 + 10X + 1 \pmod{17}$$

- Now prover must make sure the trace follows the correct transitions. That is, we check each step as a polynomial constraint that should vanish (here vanish means on the given value the polynomial equation should produce output as 0) if the trace is valid. In the following, $f_0(i)$ means “the value of the trace at step i ”. If we take $f_0(1)$ then for the first step the value is computed considering x as 1. The important thing to consider here is when we say x here means the steps. So, when we say $x = 1$ that is referring to step 1. So, that is the reason capital X is used in $f_0(X)$.

$$f_0(1) = 3(1)^6 + 8(1)^5 + 7(1)^4 + 13(1)3 + 10(1)^2 + 10(1) + 1 \pmod{17}$$

$$f_0(1) = 3 + 8 + 7 + 13 + 10 + 10 + 1 \pmod{17}$$

$$f_0(1) = 52 \pmod{17} = 1$$

Hence, the algebraic intermediate representation (AIR) to be computed. The AIR is a formal way to represent a computation as algebraic constraints over a finite field. The following shows the transition constraints like take the first step and computing the next and the different should be 0.

$$f_0(1) - f_0(0^2) = 0 \Rightarrow 1 - 1^2 = 0$$

The above shows that, we take the first trace output and square it to get the second trace output. Now, if we see the difference between the first and second trace output, it is 0. Likewise, tested for all traces in the following.

$$f_0(2) - 2f_0(1) = 0 \Rightarrow 2 - 2 \cdot 1 = 0$$

$$f_0(3) - f_0(0^3) = 0 \Rightarrow 1 - 1^3 = 0$$

$$f_0(4) - (f_0(3) + f_0(2)) = 0 \Rightarrow 3 - (1 + 2) = 0$$

$$f_0(5) - (f_0(4) + f_0(0)) = 0 \Rightarrow 4 - (3 + 1) = 0$$

$$f_0(6) - (f_0(5) - 4) = 0 \Rightarrow 0 - (4 - 4) = 0$$

- All Constraint satisfied in the above.
- **Low Degree Extension:** Now we have the polynomial of degree $d = 6$, which is lower than the p value of F_p . Hence, it is low degree polynomial. Now, let us extend points to make it bigger domain. The major reason behind this extension is: If we have only the 7 points of our example then there is a possibility that prover can create another polynomial that will match with these points. Now the question is, how will it match? What the cheating prover will do, he/she takes the 7 point and extend it to prove with different polynomial. Hence, prover is made in the beginning itself to extend and send it. Now, the next question comes, is it not possible that the cheating prover will extend on the extended points?, the answer is for further extension, the degree needs to be changed and that is not allowed and identified by the verifier.

- Now, let us see how to extend the points,
 - Field: $\mathbb{F}_{17}^\times = \{1, 2, \dots, 16\}$ - Choose a subgroup H of order 8 with generator $g = 3$:

$$H = \{1, 3, 9, 10, 13, 5, 15, 11\}$$

- The group elements are computed as follows:

$$g^0 \text{ mod } 17 = 3^0 \text{ mod } 17 = 1$$

$$g^1 \text{ mod } 17 = 3^1 \text{ mod } 17 = 3$$

$$g^2 \text{ mod } 17 = 3^2 \text{ mod } 17 = 9$$

$$g^3 \text{ mod } 17 = 3^3 \text{ mod } 17 = 27 \text{ mod } 17 = 10$$

- Like wise for g^4, g^5, g^6, g^7, g^8 will be computed.
- Map trace steps $i = 0, \dots, 6$ to subgroup elements g^i as shown in the below. We have only 7 main trace so it is mapped with 7 points and the 8th one is not mapped. Now, the 8th point is the extended point. Here, prover is extending by one only.

Step i	$g^i \text{ mod } 17$	a_i
0	1	1
1	3	1
2	9	2
3	10	1
4	13	3
5	5	4
6	15	0

- Let us bring that 8th point ($g^8 \text{ mod } 17 = 11$) as follows to include in the trace.

$$f_0(X) = 3X^6 + 8X^5 + 7X^4 + 13X^3 + 10X^2 + 10X + 1 \pmod{17}$$

$$f_0(11) = 3(11)^6 + 8(11)^5 + 7(11)^4 + 13(11)^3 + 10(11)^2 + 10(11) + 1 \pmod{17}$$

$$11^1 \equiv 11 \pmod{17},$$

$$11^2 = 11 \cdot 11 = 121 \equiv 2 \pmod{17},$$

$$11^3 = 11^2 \cdot 11 \equiv 2 \cdot 11 = 22 \equiv 5 \pmod{17},$$

$$11^4 = 11^3 \cdot 11 \equiv 5 \cdot 11 = 55 \equiv 4 \pmod{17},$$

$$11^5 = 11^4 \cdot 11 \equiv 4 \cdot 11 = 44 \equiv 10 \pmod{17},$$

$$11^6 = 11^5 \cdot 11 \equiv 10 \cdot 11 = 110 \equiv 8 \pmod{17}.$$

So the powers modulo 17 of all are:

$$11^1 \text{ mod } 17 \equiv 11, \quad 11^2 \text{ mod } 17 \equiv 2, \quad 11^3 \text{ mod } 17 \equiv 5, \quad 11^4 \text{ mod } 17 \equiv 4$$

$$11^5 \text{ mod } 17 \equiv 10, \quad 11^6 \text{ mod } 17 \equiv 8.$$

$$f_0(11) \equiv 3 \cdot 8 + 8 \cdot 10 + 7 \cdot 4 + 13 \cdot 5 + 10 \cdot 2 + 10 \cdot 11 + 1 \pmod{17}$$

$$f_0(11) \equiv 24 + 80 + 28 + 65 + 20 + 110 + 1 \pmod{17}$$

$$f_0(11) \equiv 318 \pmod{17} = 5$$

- Now, the extended trace is $\{1, 1, 2, 1, 3, 4, 0, 5\}$
- Now, let us compute the **Merkle Tree** with the traces.
- We know $\{1, 1, 2, 1, 3, 4, 0, 5\}$ and compute hash (h) of each element individually. $l_1 = h(1), r_1 = h(1), l_2 = h(1), r_2 = h(1), l_3 = h(3), r_3 = h(4), l_4 = h(0), r_4 = h(5)$. The l and r respectively refers the left and right child of Node 1,2,3 and 4. Here, $l_1, l_2, l_3, l_4, r_1, r_2, r_3, r_4$ are leafs of the merkle tree. In case, we have odd number of elements then we will not be having both left and right child only one child possible. So, we will compute the hash of the single value and directly use as a right child in the next higher node. For example,

$$\begin{aligned} - l_{11} &= h(l_1 || r_1) \\ - r_{22} &= h(l_2) \end{aligned}$$

We already discussed in the merkle tree section, how to compute the merkle root. So, let us consider the $Root(R_1) = h(l_{111} || r_{111})$ is derived. **Important Note: The r in merkle tree refers to the right child and in other cases it is random number.**

- Now, let us compute the random number r_1 and this will be sent by the verifier in case of interactive model however we go for the non-interactive so prover will be deriving the r_1 by taking the hash of merkle root R_1 . Using the **Fiat-Shamir's Transform**, we compute as follows:

$$r_1 = \text{hash}(R_1) \pmod{17}$$

The hash can be the SHA256 algorithm and in some cases along with R_1 , other data can be concatenated to compute r_1 . However, here we will consider only the R_1 .

- Now, let us move for the **Fast Reed-Solomon** without interaction. That is instead of getting the random values r (r_1) from the verifier, it is already generated using the hash function in the previous step. Now, let us do the folding by dividing the trace into odd and even group based on index not based on the values. We have the extended trace $\{1, 1, 2, 1, 3, 4, 0, 5\}$ and classify as $f_{\text{even}} = \{1, 2, 3, 0\}$ and $f_{\text{odd}} = \{1, 1, 4, 5\}$. Assume $r_1 = 2$ just to understand.

First Round $f^{(1)}$

- For each index i , here i refers the indexes of odd and even vector that is f_{odd} and f_{even} respectively.

$$f^{(1)}[i] = f_{\text{even}}[i] + r_1 \cdot f_{\text{odd}}[i] \pmod{17}.$$

- Compute entries:

$$\begin{aligned} f^{(1)}[0] &= 1 + 2 \times 1 = 1 + 2 = 3 \pmod{17}, \\ f^{(1)}[1] &= 2 + 2 \times 1 = 2 + 2 = 4 \pmod{17}. \\ f^{(1)}[2] &= 3 + 2 \times 4 = 3 + 8 = 11 \pmod{17}. \\ f^{(1)}[3] &= 0 + 2 \times 5 = 0 + 10 = 10 \pmod{17}. \end{aligned}$$

- First round vector:

$$f^{(1)} = [3, 4, 11, 10]$$

- Now, consider this vector $f^{(1)}$ and compute the merkle tree. The root hash of this merkle tree is R_2

Second Round $f^{(2)}$

- Classify the vector $[3, 4, 11, 10]$ into $f_{\text{odd}} = \{4, 10\}$ and $f_{\text{even}} = \{3, 11\}$.
- Again use the **Fiat-Shamir's Transform** and compute r_2 . So, now r_2 is computed as follows:

$$r_2 = \text{hash}(r_1 || \text{response}_1) \pmod{17}$$

We already discussed about the response in Fiat-Shamir's Transformation section. Assume r_2 as 3 just for understanding.

- For each index i , here i refers the indexes of odd and even vector that is f_{odd} and f_{even} respectively.

$$f^{(2)}[i] = f_{even}[i] + r_2 \cdot f_{odd}[i] \pmod{17}.$$

- Compute entries:

$$\begin{aligned} f^{(2)}[0] &= 3 + 3 \times 4 = 3 + 12 \pmod{17} = 15 \pmod{17}, \\ f^{(2)}[1] &= 11 + 3 \times 10 = 11 + 30 \pmod{17} = 41 \pmod{17} = 7 \pmod{17}. \end{aligned}$$

- Second round vector:

$$f^{(2)} = [15, 7]$$

- Now, consider this vector $f^{(2)}$ and compute the merkle tree. The root hash of this merkle tree is R_3

Third Round $f^{(3)}$

- Classify the vector $[15, 7]$ into $f_{odd} = \{7\}$ and $f_{even} = \{15\}$.
- Again use the **Fiat-Shamir's Transform** and compute r_3 . So, now r_3 is computed as follows:

$$r_3 = \text{hash}(r_2 || \text{response}_3) \pmod{17}$$

Assume r_3 as 4 just for understanding.

- For each index i , here i refers the indexes of odd and even vector that is f_{odd} and f_{even} respectively.

$$f^{(3)}[i] = f_{even}[i] + r_3 \cdot f_{odd}[i] \pmod{7}.$$

- Compute entries:

$$f^{(3)}[0] = 15 + 3 \times 7 = 15 + 21 \pmod{17} = 36 \pmod{17} = 2 \pmod{17}$$

- Third round vector:

$$f^{(3)} = [2]$$

- Now, consider this vector $f^{(3)}$ and compute the merkle tree. The root hash of this merkle tree is R_4

- Now folding over and prover has to send the proving parameters to the verifier.

- R_1, R_2, R_3, R_4 : Merkle roots of each Fast Reed-Solomon Non-interactive levels.
- Random challenges derived via Fiat-Shamir's transformation:

$$r_1, r_2, r_3$$

- Final FRI value that is third round vector of Fast Reed-Solomon non-interactive computation:

$$f^{(3)}(X) = 13$$

also the second and first round vector will be given

$$f^{(2)}(X) = [2, 11] \text{ and } f^{(1)}(X) = [3, 5, 10, 15]$$

- Sub-group H size or order. In our example, it is 8. We can say size of H is $|H| = 8$
- A few query positions 1 and 2 (for example, $q_1 = 2, q_2 = 5$) are derived as follows. Here, $j \in \{1, 2\}$ means at first $j = 1$ and next $j = 2$. So, now to get points, derive.

$$q_j = \text{hash}(r_3 || j) \pmod{|H|}$$

$$q_1 = \text{hash}(r_3 || 1) \pmod{|H|}$$

$$q_2 = \text{hash}(r_3 || 2) \pmod{|H|}$$

We assumed $q_1 = 2$ and $q_2 = 5$.

- For each q_j , the prover sends:

$$f_0(q_j), \quad f_0(q_j + 1)$$

We have

$$f_0(X) = 3X^6 + 8X^5 + 7X^4 + 13X^3 + 10X^2 + 10X + 1 \pmod{17}$$

We send for

$$f_0(2), \quad f_0(3)$$

The value of $f_0(2)$ is there in the trace that is 2 and $f_0(3)$ is 1. These values are computed in the base trace also. Next,

$$f_0(5), \quad f_0(6)$$

The value of $f_0(5)$ is there in the trace that is 4 and $f_0(6)$ is 0. These values are computed in the base trace also. In case, the q_j value is 7 then we have to send $f_0(7)$ and $f_0(8)$. In this case, we know $f_0(7)$ value from the extended domain and $f_0(8)$ is value does not exist in the based as well as extended domain but can be computed using the $f_0(X)$ or $g^8 \pmod{7}$. This $f_0(X)$ or $g^8 \pmod{7}$ value will not go beyond the 17 because we apply the $\pmod{17}$ and $\pmod{7}$ respectively for all computations. So, we do not have the value for $f_0(8)$ but we can generate and send.

- In addition, the path values to verify the $f_0(2), f_0(3), f_0(5)$ and $f_0(6)$ will be sent to the verifier. As we have discussed in the merkle tree membership section, the necessary intermediate nodes hash value will be sent. For an example, to validate $f_0(2)$, which is pointed by the third leaf of the merkle tree MK_1 , prover need to provide some other hash values. So, now, along with the $f_0(2)$, hash of $f_0(3)$, the hash of one layer above right/left node, then next above layer right/left node hash till before the root hash will be given.
- The verifier also knows the $p = 17$ in F_p .
- Having all the above sent values, the **Verifier** does the verification step by step.

- First, verify the merkle path. Given the merkle path values, compute the root hash R_1 and match with the received R_1 . The prover given the path values $f_0(2)$, using that the merkle root (R'_1 , just to different the computed and received, we consider the computed as R'_1) of the merkle tree MK_1 will be computed and compared with the received merkle Root R_1 . If $R_1 = R'_1$ then accepted and proceed for further verification otherwise reject the proof and stop the process.
- Next, check the transition constraints. That is with the given $f_0(q_i + 1)$ and $f_0(q_i)$ check the following holds:

$$f_0(q_i + 1) - T(f_0(q_i)) = 0$$

This should hold because we already seen with AIR that if the values are correct then it will hold. If any modification then it will not hold. **In the equation, we see that $T(f_0(q_i))$, it has T . Here T refers that the current step or point or state is derived from the previous step that is transiting the previous step we are bringing the current step. In our earlier discussion, we have not used T . We can consider with out T also otherwise it is just a notation to represent the transition.**

- Next, verify the folding consistency using the folding formula:

$$f^{(2)}[i] = f_{\text{even}}[i] + r_2 \cdot f_{\text{odd}}[i] \pmod{p}$$

where:

- * To compute the second round vector, the verifier needs he previous round or round 1 vector values $f_{\text{even}} = [3, 11]$ and $f_{\text{odd}} = [4, 10]$ are known to the verifier because that is sent by the prover.
- * The value of $f^{(2)}[i]$ is also sent to the verifier that is $[15, 7]$

* r_2 is the random derived using the Fiat-Shamir heuristic. We know that how $r_2 = \text{hash}(r_1 || \text{response}_1)$ is computed. Receiver knows R_1 and with this he/she can compute r_1 ($r_1 = \text{hash}(R_1)$ or $r_1 = \text{hash}(R_1) \pmod{17}$, as computed at the prover). Also, receiver knows the response (response_1). Using these, the r_2 can be computed by the verifier that is $r_2 = \text{hash}(r_1 || \text{response}_1) \pmod{17}$.

For each index i , the verifier checks the folding consistency as:

$$f^{(j+1)}[i] \stackrel{?}{=} f_{\text{even}}^{(j)}[i] + r_{j+1} \cdot f_{\text{odd}}^{(j)}[i] \pmod{p}$$

For example,

$$f^{(2)}[0] \stackrel{?}{=} f_{\text{even}}^{(1)}[0] + r_2 \cdot f_{\text{odd}}^{(1)}[0] \pmod{17}$$

The verifier knows the following:

$$\begin{aligned} f^{(2)} &= [15, 7] \\ f^{(2)[0]} &= [15] \\ f^{(1)} &= [3, 4, 11, 10] \\ f_{\text{odd}}^{(2)} &= [4, 10] \\ f_{\text{even}}^{(2)} &= [3, 11] \\ f_{\text{odd}}^{(2)}[0] &= 4 \\ f_{\text{even}}^{(1)}[0] &= 3 \end{aligned}$$

$$f^{(2)}[0] \stackrel{?}{=} f_{\text{even}}^{(1)}[0] + r_2 \cdot f_{\text{odd}}^{(1)}[0] \pmod{17}$$

$$\begin{aligned} f^{(2)}[0] &\stackrel{?}{=} 3 + 3 \times 4 = 3 + 12 \pmod{17} = 15 \pmod{17} \\ &15 \stackrel{?}{=} 15 \pmod{17} \end{aligned}$$

The above matches. This ensures that the folding was performed correctly and that the vector maintains the low-degree property since the derived point is part of the equation.

- After computing $f^{(j+1)}[i]$ (that is $f^{(2)}[i]$) for all index, the verifier get the round 2 vector. The round 2 vector is $[15, 7]$. The verifier can compute the merkle tree for this vector and map with the R_3 sent by the prover. If matches then successful otherwise considered as prover is submitting the fake proof.
 - The verifier may compute another folded vector, compute the merkle tree and check if required.
 - Overall, if we see, the verifier does four verification: a) Initial Merkle Tree validation of R_1 based on the given path values, b) check the transition computation that gives value as 0, c) Compute the folded vector and check it with the received vector and d) compute the merkle tree for the folded vector and map with the sent R_2 or R_3 or others based on what is chosen.
 - If all these verification successful then the prover successfully proved it to the verifier otherwise verifier will reject the submitted proof.
- **Important Note:** a) zk-STARK does not need the trusted setup like in case of zk-SNARK (Groth16 and PLONK), b) It uses only the merkle root not all the intermediate values so it is succinct and c) The zk-STARK is computationally heavy when compared with zk-SNARK.

10.12.7 Why Polynomial Commitments?

We have seen zk-STARK and there is a need for the polynomial commitments. Why do we need that? Who do not we directly use the Integers to prove. The answers is:

- If you want to check only a few points, you would have to reveal all numbers and that leaks information.
- If we have more numbers or points or traces then we have to reveal all those points which is complex for verifier to check.
- Polynomial commitment with spot checks then verifier sees only a few points, keeping most of the traces hidden for example in zk-STARK.

11 ZK-Rollup

ZK-Rollup is the layer 2 solution in the Blockchain. The integrators create the aggregated proof and submit to the blockchain base layer for verification without disclosing the transaction details. zk-SNARK is used to achieve the ZK-Rollup. The aggregator gets the following data:

- Old State: Account balances or other assets before any transaction starts. This can be the Merkle Root.
- Transactions: The set of transactions to be rolled up will be given. The transaction contains sender, receiver, amount, signature, etc.
- Witness data: The secret key of the senders.
- Updated data: New state root that is after applying all these transactions, the state changes accordingly Merkle Root computed and that is the updated data. May also include the count of transactions included in the rollup.

Using these, aggregator or prover will generate the proof and submit to the verifier at the blockchain base layer. The verifier verifies based on the zk-SNARK accordingly accept it.

Example

Let us consider that *Alice(A)*, *Bob(B)* and *John(J)* are making the transactions. The initial balance of $A = 100$, $B = 20$ and $C = 10$. There are two transactions ($Tx_1 \Rightarrow A \rightarrow B : 5$ and $Tx_2 \Rightarrow A \rightarrow J : 10$) that happened and rollup operator will process it. The rollup defines a circuit that verifies the following:

- Sum of inputs = sum of outputs (to ensure no magical or fake token created).
- Users have sufficient balance.
- Updated balances correctly.

The zk-SNARK needs the circuit inputs that are private witness: [Alice_before = 100, Bob_before = 20, John_before = 10, Tx_1 value = 5 and Tx_2 value = 10] and Public inputs (instance): [New Merkle root / New balances, transaction hashes that is in our example, New balances: Alice=85, Bob=25, John=20 Merkle root of new balances]. The Circuit Constraints includes enforcing that each transaction is valid, enforce the sum of balances is consistent and update Merkle tree for the new balances.

Using the witness and public input (instance), the proof can be generated and submitted for the verification. The Rollup operators also submit one transaction to the blockchain base layer (Ethereum), containing the new state root (updated based on submitted transactions) and the zk-SNARK. The number of gates that are required for the zk-Rollup is in millions since various operations are included.

12 Practical Experiment Questions

Based on the concepts discussed, the following can be practiced to understand the concepts in detail.

- i. Implement the Length Extension attack: Generate a hash value for an input message (concatenated with the secret) and give that hash value along with padded message as input again to the same hash function and take the output (O_1). Later compute the hash value for the original input message (concatenated with the secret) concatenated with the padded message and get the output (O_2). The attack will be considered as successful if $O_1 = O_2$. Consider the SHA256 hash function which has 512 bits block size.
- ii. Write a program and test whether the project coordinates give the result as affine coordinates.
- iii. Implement the point doubling and point addition in the elliptic curve. Choose your own curve and implement using Python with the help of existing packages. If possible write the code and create your own package.
- iv. Implement the ECDSA (signature generation and verification) using the curve chosen in the previous experiment.
- v. Implement bilinear pairing (either Tate or Weil based) using the SageMath or PBC (Pairing-Based Cryptography) Library in C.
- vi. Implement the BLS signature to validate the pairing based cryptography using the previous experiment learning.
- vii. Implement the KZG polynomial commitment scheme. If possible write your own code or utilizing the existing kzg libraries.
- viii. Implement the Groth16 zk-SNARK protocol considering all the mathematical process. Ensure you display the output of each process.
- ix. Implement the PLONK zk-SNARK protocol considering all the mathematical process. Ensure you display the output of each process.
- x. Implement the Merkle Tree considering the N messages.
- xi. Implement zk-Rollup considering limited user accounts and transactions.

13 References

- Ben Adida, Pairing-Based Cryptography <https://courses.csail.mit.edu/6.897/spring04/L25.pdf> [accessed on 02/10/2025]
- <https://en.wikipedia.org/wiki/Pairing> [accessed on 02/10/2025]
- Dan Boneh, The Basics of Pairings, 3rd BIU Winter School on Cryptography <https://www.youtube.com/watch?v=F4x2kQTKYFY> [accessed on 02/10/2025]
- Linear and Bilinear Functionals, <https://www.math.ttu.edu/~klong/5310-fall10/functionalnotes.pdf> [accessed on 02/10/2025]
- https://en.wikipedia.org/wiki/BLS_digital_signature [accessed on 02/10/2025]
- Bhaskar Krishnamachari, Understanding Zero-Knowledge Proofs: Part 2 — Math Basics for Cryptography <https://medium.com/@bhaskark2/understanding-zero-knowledge-proofs-part-2-math-basics-for-cryptography-4733a9dd85d2> [accessed on 10/10/2025]
- Adam Luciano, zk-SNARKs — A Realistic Zero-Knowledge Example and Deep Dive, <https://medium.com/coinmonks/zk-snarks-a-realistic-zero-knowledge-example-and-deep-dive-c5e6eaa7131c> [accessed on 10/10/2025]
- Encoding Labs, [ZK] Part 2: ZK-SNARK Generation, <https://medium.com/@encodinglabs/zk-part-2-zk-snark-generation-1b03233fce11> [accessed on 10/10/2025]

- Risen Crypto, R1CS and QAP - From Zero to Hero with Finite Fields & sagemath <https://risencrypto.github.io/R1CSQAP/> [accessed on 10/10/2025]
- <https://crypto.stackexchange.com/questions/53974/when-adding-two-points-on-an-elliptic-curve-why-flip-over-the-x-axis> [accessed on 20/10/2025]
- <https://crypto.stackexchange.com/questions/15928/in-bilinear-pairings-what-is-the-difference-between-type-2-and-type-3> [accessed on 24/10/2025]
- Jens Groth, On the Size of Pairing-based Non-interactive Arguments, <https://eprint.iacr.org/2016/260.pdf> [accessed on 25/10/2025]
- Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg, "Constant-Size Commitments to Polynomials and Their Applications", <https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf> [accessed on 02/11/2025]
- The KZG/Kate Polynomial Commitment Scheme, <https://risencrypto.github.io/Kate/> [accessed on 04/11/2025]
- Fiat-Shamir heuristic, https://en.wikipedia.org/wiki/Fiat%E2%80%93Shamir_heuristic [accessed on 04/11/2025]
- Eli Ben-Sasson. Iddo Bentov, Yinon Horesh, Michael Riabzev, Scalable, transparent, and post-quantum secure computational integrity, <https://eprint.iacr.org/2018/046.pdf> [accessed on 04/11/2025]
- Eli Ben-Sasson, zk-STARKs Uncovered: Eli Ben-Sasson's Intense Masterclass — Basecamp Cohort, <https://www.youtube.com/watch?v=jg9KSNOO2XY> [accessed on 07/11/2025]
- Dan Boneh, ZKP MOOC Lecture 5: The Plonk SNARK. <https://www.youtube.com/watch?v=A0oZVEXav24> [accessed on 14/11/2025]
- Jan Oberst Towards, Stateless Clients in Ethereum: Benchmarking Verkle Trees and Binary Merkle Trees with SNARKs, <https://arxiv.org/html/2504.14069v1> [accessed on 21/11/2025]
- John Kuszmaul, "Verkle Trees", <https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf> [accessed on 21/11/2025]
- Merkle-Damgård construction, https://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction [accessed on 22/11/2025]
- Sponge Construction, https://en.wikipedia.org/wiki/Sponge_function [accessed on 22/11/2025]
- Length extension attack, https://en.wikipedia.org/wiki/Sponge_function [accessed on 22/11/2025]
- Dario Catalano and Dario Fiore, Vector Commitments and their Applications, <https://eprint.iacr.org/2011/495.pdf> [accessed on 22/11/2025]

14 Annexure

14.1 Additional Pairing Properties and Relations

The bilinear pairing has various properties and relations. The following are properties:

Bilinearity in sums:

$$e(P_1 + P_2, Q) = e(P_1, Q) \cdot e(P_2, Q)$$

$$e(P, Q_1 + Q_2) = e(P, Q_1) \cdot e(P, Q_2)$$

Scalar multiplication pulled out on one side:

$$e(aP, Q) = e(P, Q)^a, \quad e(P, bQ) = e(P, Q)^b$$

Swapping scalars and points in symmetric pairing:

$$e(aP, bQ) = e(bP, aQ)$$

This holds if and only if $G_1 = G_2$.

Commutativity of pairing (symmetric case):

$$e(P, Q) = e(Q, P)$$

Inverse property:

$$e(-P, Q) = e(P, Q)^{-1}, \quad e(P, -Q) = e(P, Q)^{-1}$$

Identity property: Let \mathcal{O} be the identity point on the curve,

$$e(\mathcal{O}, Q) = 1, \quad e(P, \mathcal{O}) = 1$$

Distributivity of scalar multiplication across pairing:

$$e(aP, bQ + cR) = e(P, Q)^{ab} \cdot e(P, R)^{ac}$$

Powering the output by a scalar: For integer k ,

$$e(P, Q)^k = e(kP, Q) = e(P, kQ)$$

Non-degeneracy (important property):

$$e(P, Q) = 1 \quad \forall Q \in G_2 \implies P = \mathcal{O}$$

$$e(P, Q) = 1 \quad \forall P \in G_1 \implies Q = \mathcal{O}$$

This ensures the pairing is not trivial.

Bilinearity extended to linear combinations: For scalars a_i, b_j and points $P_i \in G_1, Q_j \in G_2$,

$$e\left(\sum_i a_i P_i, \sum_j b_j Q_j\right) = \prod_{i,j} e(P_i, Q_j)^{a_i b_j}$$