Keys and Addresses

Zichao Yang

Zhongnan University of Economics & Law

Date: October 10, 2024

Anonymous Transactions

Alice wants to pay Bob via Bitcoin, and she wants the Bitcoin system to ensure that:

- Bitcoin full nodes can verify her transaction.
- This transaction should not be tied to either her or Bob's real-world identity.
- Bob can further spend the bitcoins he receives.

Anonymous Transactions

The original Bitcoin paper describes a very simple scheme for achieving those goals:

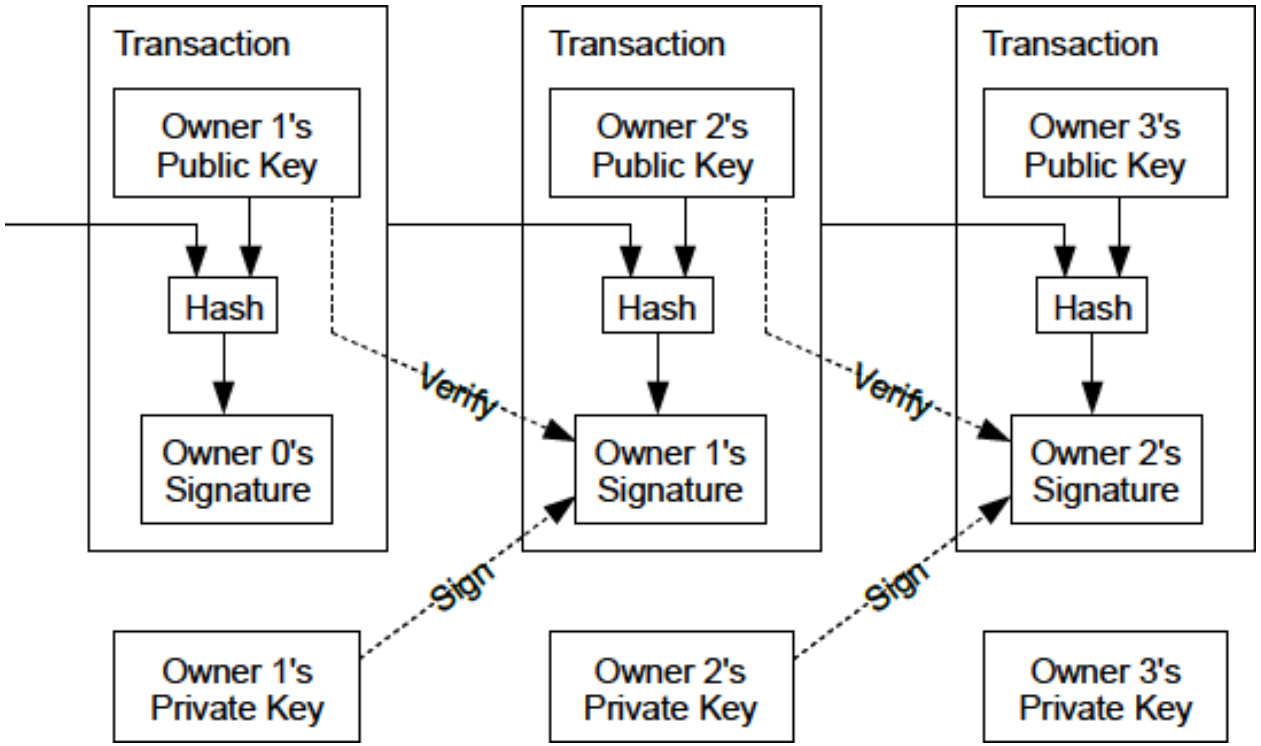


Figure 1: Transaction chain from original Bitcoin paper
Nakamoto, Satoshi, and A. Bitcoin. "A peer-to-peer electronic cash system." Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf 4.2 (2008): 15.

Anonymous Transactions

Bob accepts bitcoins to a public key (Owner 3's Public Key) in a transaction that is signed by Alice (Owner 2's Signature).

The bitcoins that Alice is spending had been previously received to one of her public keys (Owner 2's Public Key) and she uses the corresponding private key (Owner 2's Private Key) to generate her signature.

We'll examine public keys, private keys, signatures, and hash functions in this section, and then use all of them together to describe the addresses used by modern Bitcoin software.

Public Key Cryptography

Public key cryptography was invented in the 1970s and is a mathematical foundation for modern computer and information security.

Bitcoin use public key cryptography to create **a key pair** that consists of a private key and a public key.

The **public key** is used to **receive funds**, and the **private key** is used to **sign transactions** to spend the funds.

There is a mathematical relationship between the public and the private key that allows the private key to be used to generate **signatures** on messages. These signatures can be validated against the public key **without revealing the private key**.

Illustration: PGP Tool

Private Keys

A private key is simply **a number**, picked at **random**.

The public key can be calculated from the private key, so some Bitcoin wallets only store the private key.

We pick a **256 bits random number** for private key, so the private key can be any number in decimal between 0 and $n - 1$, where $n = 1.1578 \times 10^{77}$ (slightly less than 2^{256}).

The size of Bitcoin's private key space ($2^{256} \approx 10^{77}$) is an unfathomably large number. For comparison, the visible universe is estimated to contain 10^{80} atoms.

Public Keys

The public key is calculated from the private key using **elliptic curve** multiplication, which is irreversible: $K = k \times G$, where k is the private key, G is a constant point called the *generator point*, and K is the resulting public key.

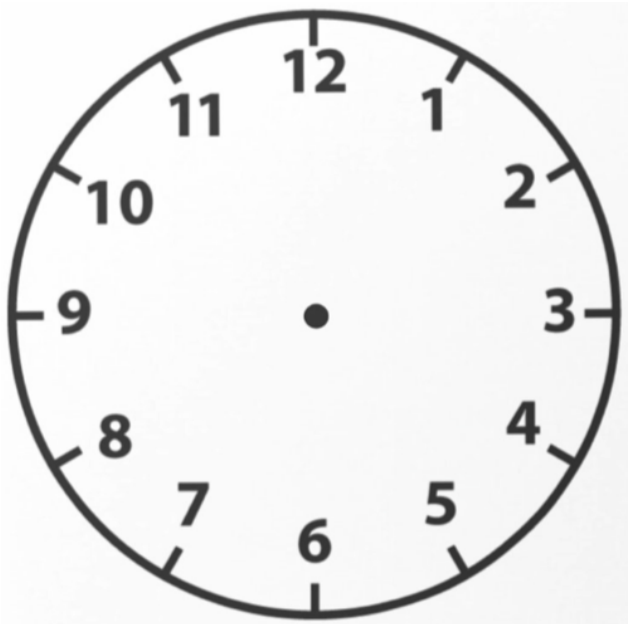
The advantage of public key cryptography is that calculating k if you know K is as difficult as trying all possible values of k (i.e., a brute-force search).

To illustrate how public key cryptography works, we need to first understand what is **elliptic curve cryptography**. Before that, we need to talk about the concept of **finite field**.

I. Finite Field

Clock Math

Q: Does it make sense to do arithmetic using just the 12 numbers on the clock?



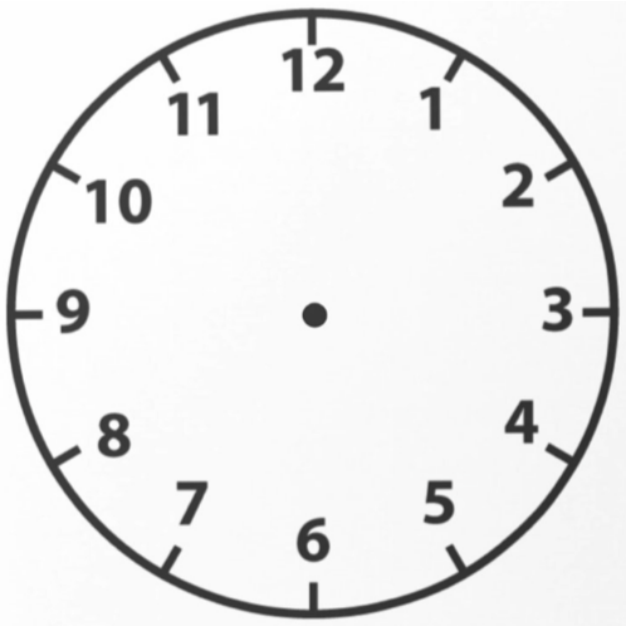
Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

Let's try some basics.

$8 + 8 = ?$
$5 + 5 + 5 + 5 + 5 = ?$

So, **addition** seems to work.

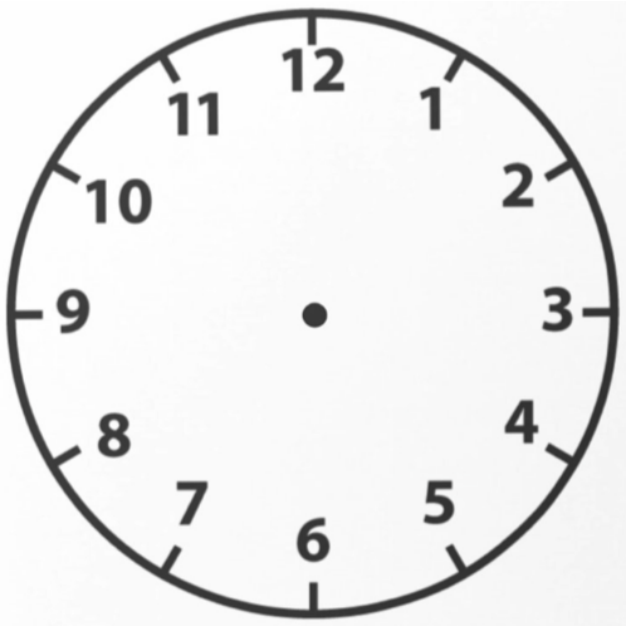


Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

And it plays nicely with **subtraction**:

$$8 + 6 = 2$$
$$8 = 2 - 6$$
$$6 = 2 - 8$$



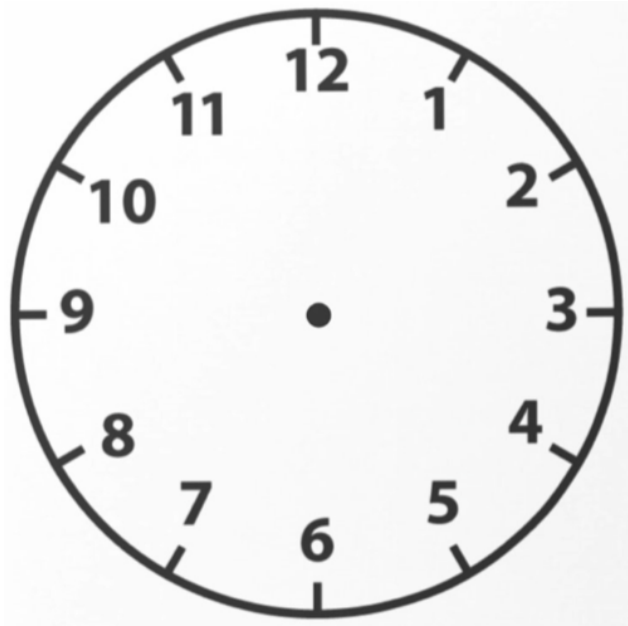
Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

In "clock arithmetic", what's 5×5?

Remember $5+5+5+5+5=1$?
Then $5 \times 4 =$?

It seems like **multiplication** also works.

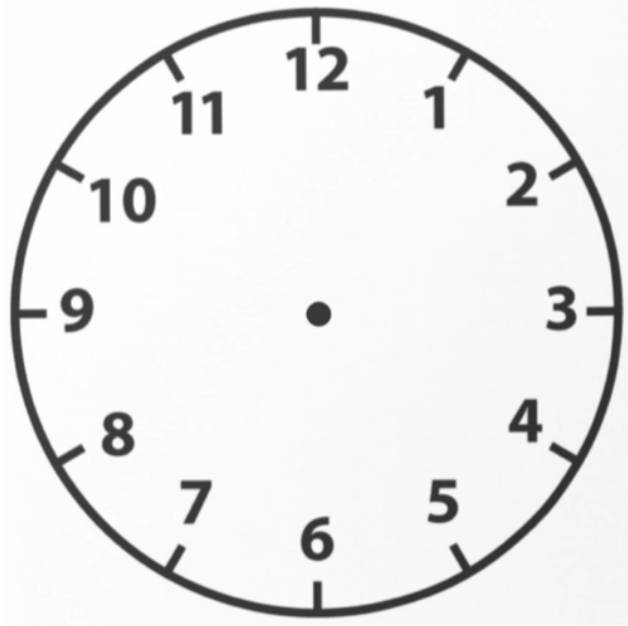


Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

Let's make a multiplication table...

x	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	2	4	6	8	10	12
3	3	6	9	12	3	6	9	12	3	6	9	12
4	4	8	12	4	8	12	4	8	12	4	8	12
5	5	10	3	8	1	6	11	4	9	2	7	12
6	6	12	6	12	6	12	6	12	6	12	6	12
7	7	2	9	4	11	6	1	8	3	10	5	12
8	8	4	12	8	4	12	8	4	12	8	4	12
9	9	6	3	12	9	6	3	12	9	6	3	12
10	10	8	6	4	2	12	10	8	6	4	2	12
11	11	10	9	8	7	6	5	4	3	2	1	12
12	12	12	12	12	12	12	12	12	12	12	12	12



Source: Risc Zero - Study Club: Introduction to Finite Fields

We call 1 the "multiplicative identity", and 12 the "additive identity" (like 0 in normal arithmetic).

Clock Math

How about clock division?

What is 8 divided by 4 in "clock arithmetic"?

Looking for x to satisfy $4x = 8$...

x	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	2	4	6	8	10	12
3	3	6	9	12	3	6	9	12	3	6	9	12
4	4	8	12	4	8	12	4	8	12	4	8	12
5	5	10	3	8	1	6	11	4	9	2	7	12
6	6	12	6	12	6	12	6	12	6	12	6	12
7	7	2	9	4	11	6	1	8	3	10	5	12
8	8	4	12	8	4	12	8	4	12	8	4	12
9	9	6	3	12	9	6	3	12	9	6	3	12
10	10	8	6	4	2	12	10	8	6	4	2	12
11	11	10	9	8	7	6	5	4	3	2	1	12
12	12	12	12	12	12	12	12	12	12	12	12	12

Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

How about clock division?

$4x = 8$ has three solutions (2, 5, and 8).

Also $4x = 1$ has no solutions.

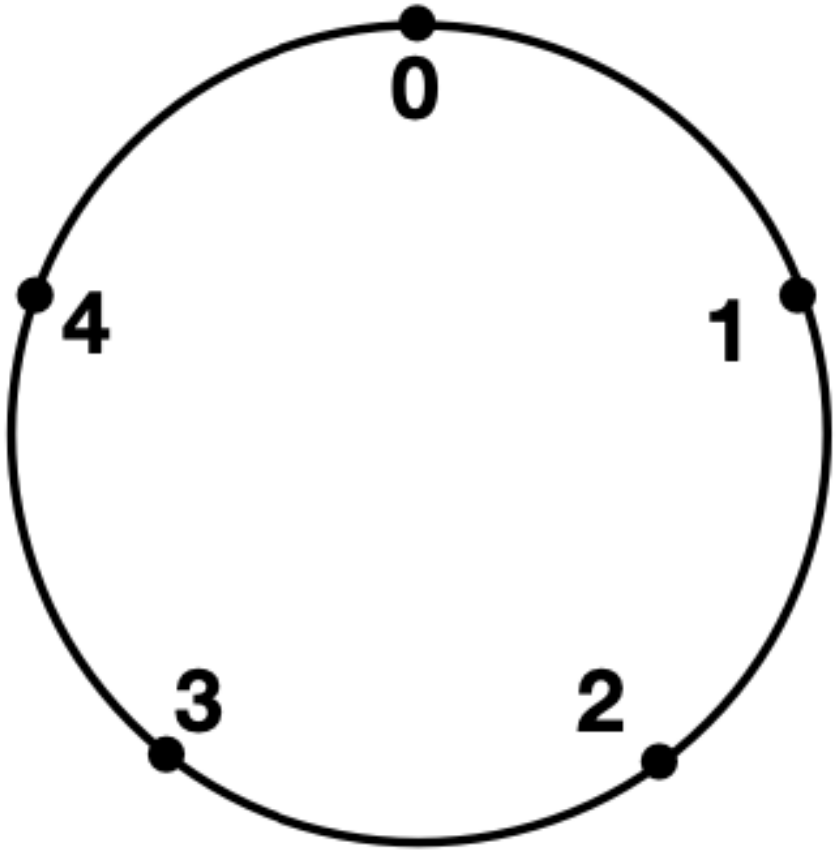
x	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	2	4	6	8	10	12
3	3	6	9	12	3	6	9	12	3	6	9	12
4	4	8	12	4	8	12	4	8	12	4	8	12
5	5	10	3	8	1	6	11	4	9	2	7	12
6	6	12	6	12	6	12	6	12	6	12	6	12
7	7	2	9	4	11	6	1	8	3	10	5	12
8	8	4	12	8	4	12	8	4	12	8	4	12
9	9	6	3	12	9	6	3	12	9	6	3	12
10	10	8	6	4	2	12	10	8	6	4	2	12
11	11	10	9	8	7	6	5	4	3	2	1	12
12	12	12	12	12	12	12	12	12	12	12	12	12

Source: Risc Zero - Study Club: Introduction to Finite Fields

Clock Math

Can we make "clock arithmetic" work with a different size of clock?

Yes! **Prime-sized clocks** are what we are looking for. We call them **fields**.



Source: Risc Zero - Study Club: Introduction to Finite Fields

Concept of Field

In mathematics, a **field** is a set equipped with two operations: addition and multiplication. And it has the following properties:

1. Closure

- **Closure under Addition**: For any two elements a and b in the field, their sum $a + b$ is also an element of the field.
- **Closure under Multiplication**: For any two elements a and b in the field, their product $a \times b$ is also an element of the field.

Concept of Field

2. Associativity

- **Associativity of Addition**: For all elements a, b, and c in the field, $(a + b) + c = a + (b + c)$.
- **Associativity of Multiplication**: For all elements a, b, and c in the field, $(a \times b) \times c = a \times (b \times c)$.

3. Commutativity

- **Commutativity of Addition**: For all elements a and b in the field, $a + b = b + a$.
- **Commutativity of Multiplication**: For all elements a and b in the field, $a \times b = b \times a$.

Concept of Field

4. Identity Elements

- **Additive Identity**: There exists an element, denoted 0, in the field such that for any element a in the field, $a + 0 = a$.
- **Multiplicative Identity**: There exists an element, denoted 1, in the field such that for any element a in the field, $a \times 1 = a$, and $1 \neq 0$.

5. Inverses

- **Additive Inverse**: For each element a in the field, there exists an element $-a$ (the additive inverse of a) such that $a + (-a) = 0$.
- **Multiplicative Inverse**: For each non-zero element a in the field, there exists an element a^{-1} (the multiplicative inverse of a) such that $a \times a^{-1} = 1$.

Concept of Field

6. Distributivity
Multiplication distributes over addition, meaning for all elements a, b, and c in the field, $a \times (b + c) = (a \times b) + (a \times c)$.

7. Non-zero Element Behavior
The set of non-zero elements of the field forms a **multiplicative group**, meaning every non-zero element has an inverse with respect to multiplication.

Examples of Fields

- **Real Numbers** ($\mathbb{R}$)
- **Complex Numbers** ($\mathbb{C}$)
- **Finite Fields**: Fields with a finite number of elements, such as $\mathbb{F}_p$ (integers modulo a prime p).

Finite Field

Q: Why do we need to define a finite field in computer cryptography? (hint: overflow and underflow)

In math notation a finite field set looks like:

$$\mathbb{F}_p = \{0, 1, 2, ..., p-1\}$$

where $\mathbb{F}_p$ is called "field of p" and $\{0, 1, 2, ..., p-1\}$ are the elements of the set.

The tool we can use to make a finite field closed under addition, subtraction, multiplication, and division is called **modulo arithmetic**.

Finite Field in Python

- Modulo Operation in Python

  $7\%3$, $7\%3.0$, $-27\%13$, $12.5\%3$

- Finite Field Addition and Subtraction

  $\mathbb{F}_{57}$: $44+33$, $9-29$, $17+42+49$, $52-30-38$

- Finite Field Multiplication

  $\mathbb{F}_{97}$: $95\times 45\times 31$, $17\times 13\times 19\times 44$, $12^7\times 77^{49}$

Finite Field Division

In normal math, division is the inverse of multiplication: $7 \times 8 = 56$ implies that $56/8 = 7$.

We can use the above definition to help us understand finite field division. In $\mathbb{F}_{19}$, we know that:

$$3 \cdot_f 7 = 21\%19 = 2 \text{ implies that } 2/_f 7 = 3$$

Now you may ask yourself that, how do I calculate $2/_f 7$ if I don't know beforehand that $3 \cdot_f 7 = 2$?

Great question! And we should introduce **Fermat's Little Theorem**.

Fermat's Little Theorem

The theorem tells us that:

$$n^{(p-1)} \% p = 1 \text{ where } p \text{ is prime.}$$
$$n^{-1} \% p = n^{(p-2)} \% p \text{ where } p \text{ is prime.}$$

Now can you figure out how to calculate:

$\mathbb{F}_{19}$: 2/7 or 7/5?

Finite Field with a Prime Number Order

Extra Q: Why the order of a finite field is always a prime number?

II. Elliptic Curves

Elliptic Curve Definition

An elliptic curve is typically defined by an equation of the following form:

$$y^2 = x^3 + ax + b$$

where a and b are constants.

Q: can we call this equation a function?

Point Addition

A point A on the elliptic curve is represented as $A = (x_A, y_A)$. The set of points on the curve includes a special point called the **infinity point**, denoted as O (or I). The infinity point O is not visible on the plot.

Point addition is that we can do an operation on two of the points on the elliptic curve and get a third point, which is also on the curve.

Point addition satisfies certain properties:

- **identity**: O + A = A
- **invertibility**: A + (-A) = O
- **commutativity**: A + B = B + A
- **associativity**: (A + B) + C = A + (B + C)

Point Addition

To add two points A and B on the elliptic curve, the following rules apply:

Case 1: A and B are distinct

- **Slope Calculation:** Calculate the slope m of the line passing through points A and B:

$$m = \frac{y_B - y_A}{x_B - x_A}, \quad (x_A \neq x_B)$$

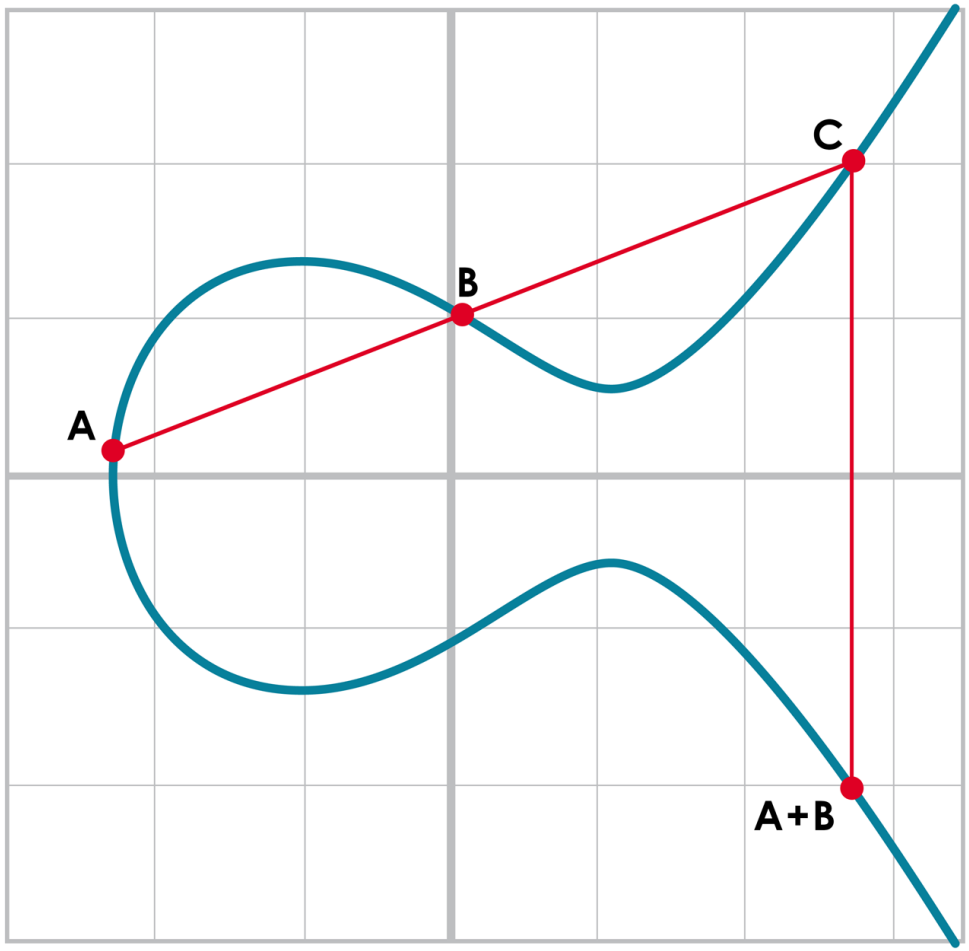
- **New Point Calculation:** The coordinates of the sum $C = A + B$ are given by:

$$x_C = m^2 - x_A - x_B$$

$$y_C = m(x_A - x_C) - y_A$$

Geometric Illustration

Case 1:



Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Point Addition

Case 2: $A = B$ (Point Doubling)

If A and B are the same point, the process is slightly different:

- **Slope Calculation:** The slope m is calculated using the derivative of the curve at that point:

$$m = \frac{3x_A^2 + a}{2y_A}, \quad (y_A \neq 0)$$

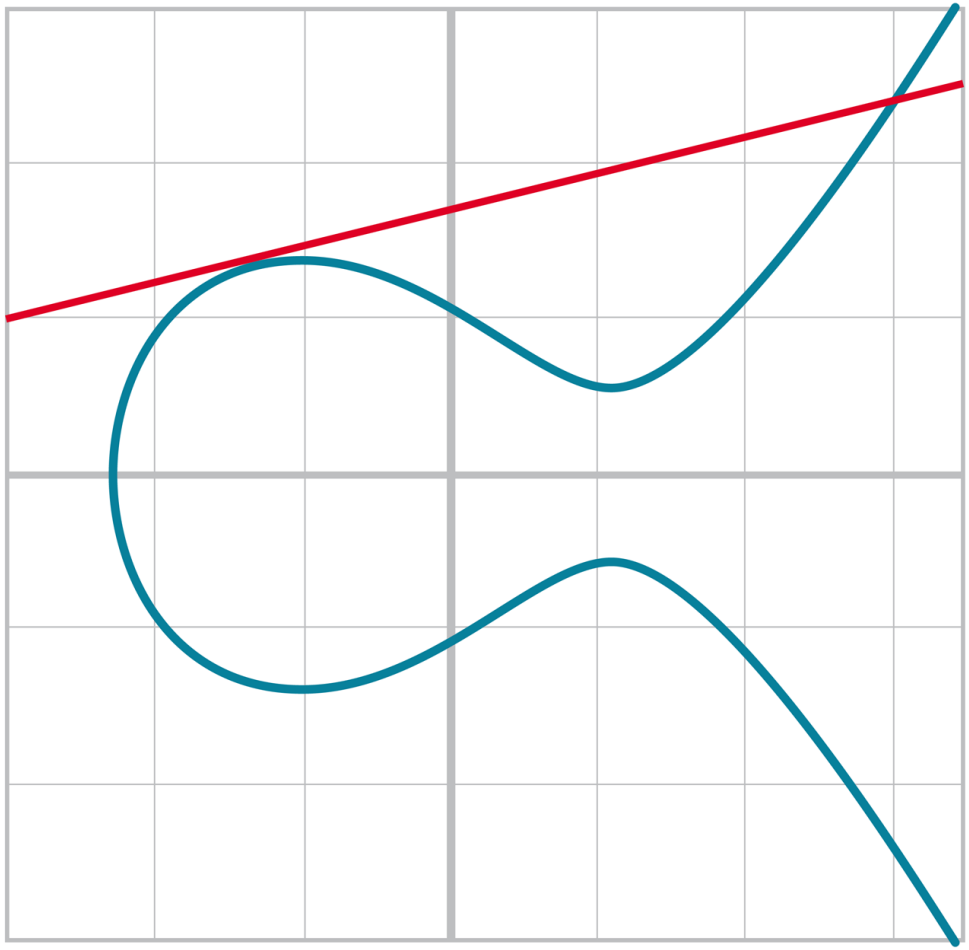
- **New Point Calculation:** The coordinates of the sum $C = A + A$ are:

$$x_C = m^2 - 2x_A$$

$$y_C = m(x_A - x_C) - y_A$$

Geometric Illustration

Case 2:



Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Point Addition

Case 3: A or B is the point at infinity O If either A or B is the point at infinity, the result of the addition is simply the other point:

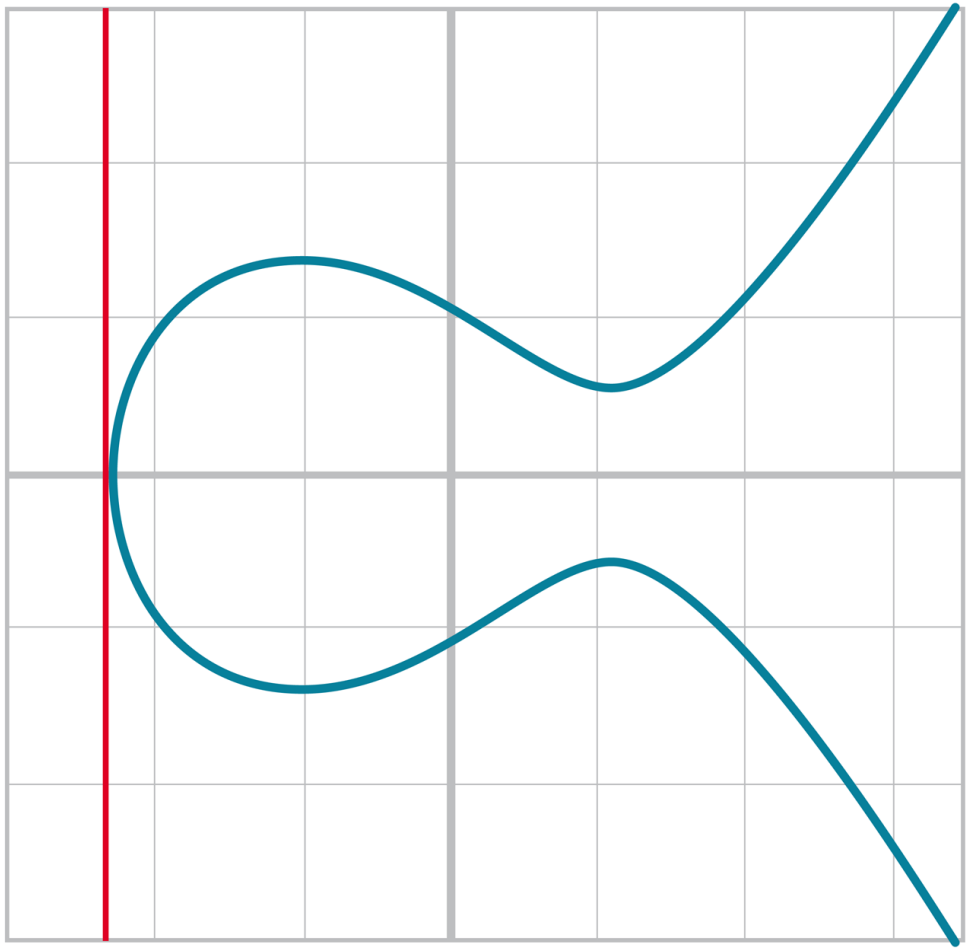
$$C = A + O = A$$

Special Cases:

- In Case 1, if $x_A = x_B$ and $y_A \neq y_B$, the slope is undefined, and we have: $C = A + B = O$
- In Case 2, if $y_A = 0$, the slope is undefined, and we have: $C = A + A = O$

Geometric Illustration

Special Case 2:



Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

III. Elliptic Curves Cryptography

Elliptic Curves over Finite Fields

In part II, we discussed how to conduct appoint addition over real numbers. Now we will show that the point addition equations can also be used over the finite field discussed in part I.

However, in order to apply point addition on finite fields, we need to use the addition/subtraction as defined in part I.

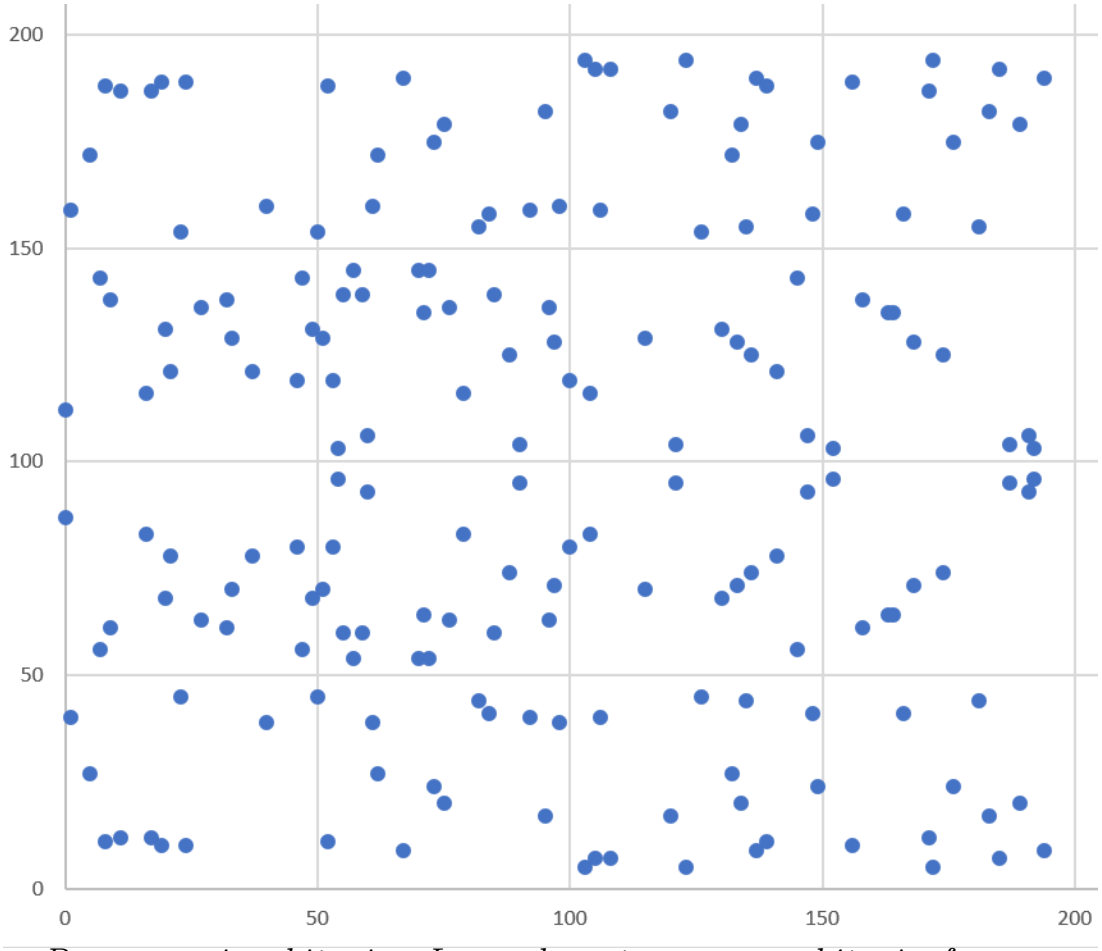
Example: define an elliptic curve over a finite field as $y^2 = x^3 + 7$ over F_{103}, check if the point $(17, 64)$ is on the curve.

$$y^2 = 64^2 \% 103 = 79$$
$$x^3 + 7 = (17^3 + 7) \% 103 = 79$$

Hence, $(17, 64)$ is on the curve.

Elliptic Curves over Finite Fields

The plot of an elliptic curve looks vastly different in a finite field.



Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Elliptic Curves over Finite Fields

What can we tell from the above graph?

It is very much a scattershot of points and there's no smooth curve here.

The only pattern is that the curve is symmetric right around the middle (because of the y^2 term).

The graph is not symmetric over the *x-axis* as in the curve over reals, but about halfway up the *y-axis* due to there not being negative numbers in a finite field.

Exercise: Evaluate whether these points are on the curve $y^2 = x^3 + 7$ over F_{223}: (192, 105), (17, 56), (200, 119), (1, 193), (42, 99)

Point Addition over Finite Fields

All of the equations we learned from part II work over finite fields.

Exercise:

For the curve $y^2 = x^3 + 7$ over F_{223}, find:

(170, 142) + (60, 139)

(47, 71) + (17, 56)

(143, 98) + (76, 66)

Scalar Multiplication for Elliptic Curves

Because we can add a point to itself, we can introduce some new notation:

$$(170, 142) + (170, 142) = 2 \cdot (170, 142)$$

This is what we call **scalar multiplication**. That is, we have a scalar number in front of the point. We can do this because we have defined point addition and point addition is associative.

One property of scalar multiplication is that **it's really hard to predict without calculating**.

Let's see a scalar multiplication graph in the next slide.

Scalar Multiplication for Elliptic Curves

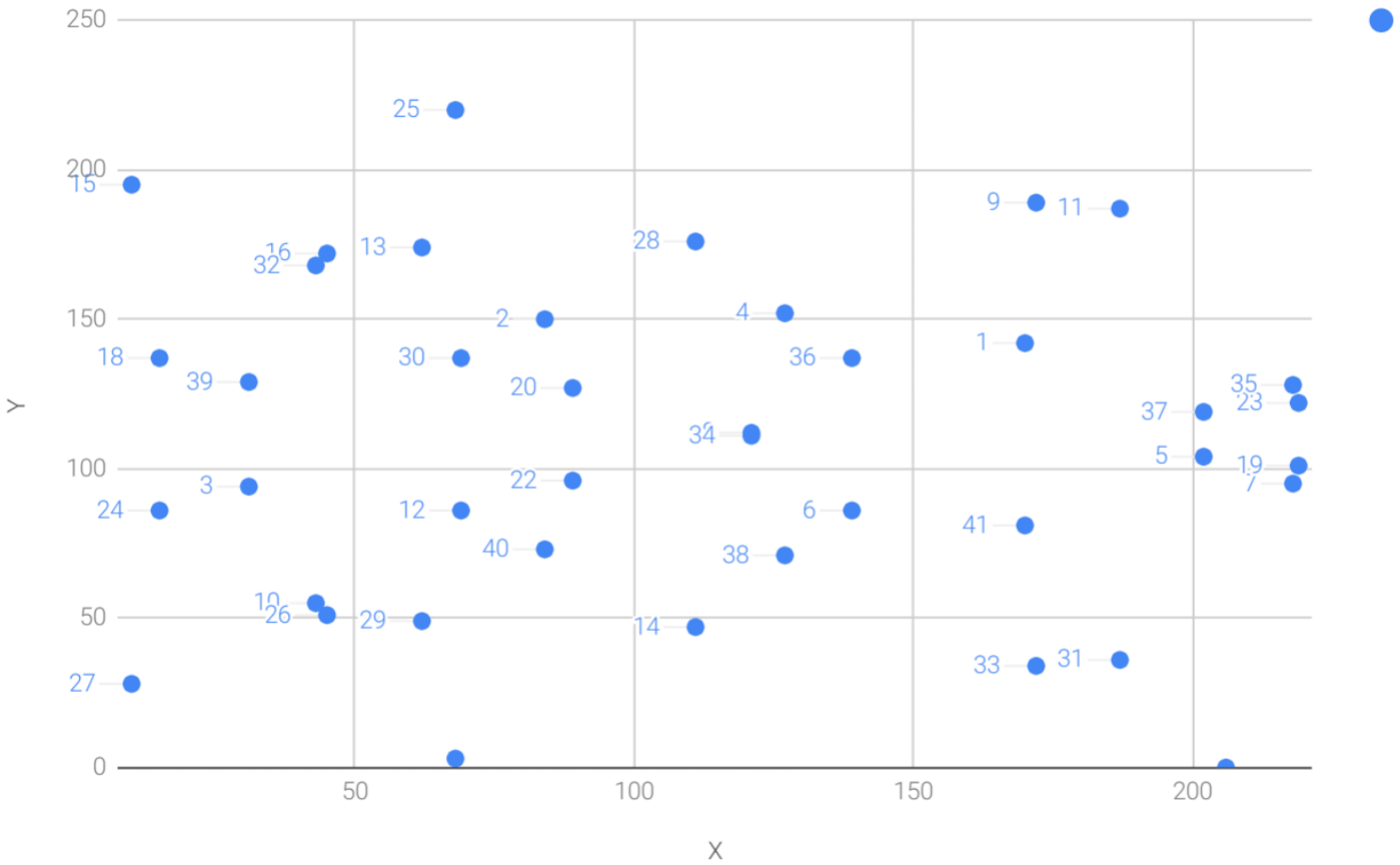


Figure 2: Scalar multiplication results for $y^2 = x^3 + 7$ over F_{223} for point (170,142)

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Scalar Multiplication for Elliptic Curves

Discrete log problem: performing scalar multiplication is straightforward, but doing the opposite, point division, is not. Discrete log problem is the basis of elliptic curve cryptography. Origin of DLP

Another property of scalar multiplication is that **at a certain multiple, we get to the point at infinity** (remember the point at infinity is 0).

If we imagine a point G (e.g. (170, 142) in the above plot) and scalar-multiply until we get the point at infinity, we end up with a set:

$$\{G, 2G, 3G, 4G, ...nG\} \text{ where } nG = 0$$

This set is called a **group**, and because n is finite, we have a *finite group* (or more specifically, a *finite cyclic group*).

Group VS. Field

A **group** is a set of elements combined with a binary operation (e.g., addition or multiplication) that satisfies four key properties: closure, associativity, identity, and inverses.

A **field** is a more structured set with two operations: addition and multiplication. A field is essentially a system where both of these operations satisfy group-like properties, with a few more rules.

Elliptic curve cryptography is built on the **group** structure of elliptic curves, where addition operation is available. (Recall...we never mentioned point multiplication on elliptic curves.)

Extra: the Origin of the Discrete Log Problem

In the original form, the DLP asks:

Given a base element g in a finite cyclic group and a result q (which is the result of repeatedly multiplying g by itself some number of times), what is the integer k such that:

$$g^k = q \pmod{p}$$

where p is a prime and g is a generator of the group.

- In this setting, g is the known **generator** (or base), and q is the known result. The problem is to find k, which is called the **discrete logarithm** of q with respect to g.
- Finding k is computationally hard when the group is large, which is why DLP is important for cryptographic security.

Extra: the Origin of the Discrete Log Problem

In the context of **Elliptic Curve Cryptography (ECC)**, the DLP takes a slightly different form, using **scalar multiplication** instead of exponentiation. Here's how it works:

- **Scalar Multiplication**: Given a generator point G on an elliptic curve and an integer k, scalar multiplication involves adding the point G to itself k times. The result is a new point $Q = kG$ on the elliptic curve. This operation is analogous to exponentiation in finite field cryptography.

- The **Elliptic Curve Discrete Logarithm Problem (ECDLP)** can be stated as follows:
  - Given two points on an elliptic curve, G and Q, where $Q = kG$, find the integer k.
  - In other words, the problem is to find k (the scalar) such that Q is the result of multiplying G by k, i.e., $Q = kG$.

Discrete Log Problem Illustration

The key to making scalar multiplication into public key cryptography is using the fact that scalar multiplication on elliptic curves is very hard to reverse. Note the previous exercise.

Let's calculated the point $Q = kG = k \cdot (47, 71)$ for different k in F_{223} in python. What can you find?

If you look closely at the numbers, there's no real discernible pattern to the scalar multiplication.

Scalar multiplication looks really random, and that's what gives this equation **asymmetry**. An asymmetric problem is one that's easy to calculate in one direction, but hard to reverse.

Different Orders: Order of the Finite Field

Let's take a look at the elliptic curve equation again:

$$y^2 = x^3 + ax + b \pmod{p}$$

The p represents the **order of the finite field** $\mathbb{F}_p$ over which the elliptic curve is defined.

p is typically a large prime number, and it defines the size of the finite field that determines the number of possible x-coordinates (and corresponding y-coordinates) for points on the curve.

You can think of p as the parameter that defines how large the canvas can be to draw the elliptic curve.

Different Orders: Order of the Elliptic Curve Group

The total number of points on an elliptic curve (including the point at infinity) over $\mathbb{F}_p$ is called the **group order**.

By **Hasse's theorem**, the number of points on an elliptic curve E defined over $\mathbb{F}_p$, denoted $\#E(\mathbb{F}_p)$, is approximately p, but it lies within the following range:

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

So, the actual number of points is typically very close to p, but not exactly equal to it. For $p = 2^{256}$, the number of points on the elliptic curve would be close to 2^{256}, but still smaller than 2^{256}.

Different Orders: Order of a Certain Point

Given a generator point G, the number of times you can add G to itself before reaching the point at infinity, O, is called the **order of the point** G.

$$Q = kG = O, \text{ where } k \text{ is the order of the point G.}$$

Q: can you tell me the order of the point G in our previous example?

The total group order is the number of points on the elliptic curve, but an individual point (i.e., G) can lie in a smaller cyclic subgroup. In our case, G belongs to a cyclic subgroup of order 21.

Relationship of different orders: $k \leq \#E(\mathbb{F}_p) < p$

Different Orders

Q: among the three orders we discussed, which one is essential to make sure the encryption is strong?

Different Orders

Both the **order of the elliptic curve group** and the **order of the generator point** are crucial for strong encryption.

- **Order of the Elliptic Curve Group**: This is the most critical factor for ensuring security, as it dictates the overall difficulty of the discrete logarithm problem.
- **Order of the Generator Point**: This is also important, as it helps prevent certain types of attacks, like small subgroup attacks.

Well-defined elliptic curves, like *secp256k1* used in Bitcoin, pick the generator point G that generates the largest prime-order subgroup. Actually in *secp256k1* the entire group of points on the curve is the prime-order subgroup.

The Curve for Bitcoin

The definition of an elliptic curve over a finite field is
$y^2 = x^3 + ax + b \pmod{p}$

For the curve used by Bitcoin, *secp256k1*, the parameters are:
(1) $a = 0$, $b = 7$, making the equation $y^2 = x^3 + 7$

(2) $p = 2^{256} – 2^{32} – 977$

(3) the generator point (G_x, G_y):
$G_x =$
0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
$G_y =$
0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

(4) order of the generator point:
$n =$ 0xfffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

What is the $0x$?

Values that begin with $0x$ are hexadecimal (base-16) numbers.

In computer science and digital electronics, hexadecimal numbers are often used because they provide a more human-readable way to represent binary data.

Here's what 0x means:

- $0x$ is a prefix indicating that the number that follows is in hexadecimal format.
- Hexadecimal is a base-16 numeral system, meaning it uses 16 different symbols: 0-9 and A-F.
  - 0-9 represent values from 0 to 9.
  - A, B, C, D, E, F represent values from 10 to 15.

IV. Public Key Cryptography

Public Key Cryptography

Now we finally have the tools that we need to do public key cryptography operations!

The key operation that we need is $P = eG$, which is an asymmetric equation. We can easily compute P when we know e and G, but we cannot easily compute e when we know P and G. This is the discrete log problem described earlier.

Generally, we call e the **private key** and P the **public key**. Note here that the private key is a single 256-bit number and the public key is a coordinate (x,y), where x and y are each 256-bit numbers. G is the generator point predefined by the algorithm.

Here we only discuss using elliptic curve cryptography (ECC) to implement public-key cryptography.

Public-Key Encryption and Private-Key Decryption

Using public-key cryptography to **encrypt** a message with a public key and then **decrypt** it with a private key is a fundamental part of secure communication.

- **Encryption**: The sender uses the **recipient's public key** to encrypt the message. The public key is publicly available, so anyone can encrypt messages for the recipient.
- **Decryption**: The recipient uses their **private key** to decrypt the message. Since the private key is kept secret, only the recipient can decrypt the message.

Encryption Using ECC

1. The sender encrypts the message using the recipient's **public key** $P_{\text{pub}} = k \cdot G$ (where G is the generator point and k is the recipient's private key).
2. The sender chooses a random number r and computes the point $R = r \cdot G$.
3. The sender then computes the shared secret $S = r \cdot P_{\text{pub}}$.
4. The message is then encrypted using a symmetric key encryption scheme (e.g., AES), where the key is derived from the shared secret S.
5. The ciphertext and R are sent to the recipient.

Decryption Using ECC

1. The recipient receives the ciphertext and R.
2. The recipient computes the shared secret $S = k \cdot R$ using their **private key** k.
3. The derived key from S is used to decrypt the ciphertext and recover the original message.

Q: Can you tell how the encryption works?

Hint: $S = r \cdot P_{\text{pub}} = r \cdot k \cdot G = k \cdot R$

Public key cryptography is widely used for communication encryption. However, in Bitcoin, public key cryptography is not primarily used for encrypting messages but for **digital signatures**.

Signing Procedure

1. The sender chooses the private key e and generate the public key P: $P = eG$
2. The sender chooses a random number k and computes the point $R = k \cdot G$. We only need to track the x coordinate of $R : (r, y)$, let's call it r.
3. The sender then chooses a pair of (u, v) that satisfy the following relationship: $uG + vP = kG$.
4. The sender further hashes the original message (i.e. tx data) into a fixed size string, z.
5. The sender finds a s to make the following equations hold: $u = z/s$ and $v = r/s$
6. Finally, the sender sends out (r, s) as the signature.

Verification Procedure

1. The receiver receives the signature (r, s).
2. The receiver hashes the original tx data into a fixed size string, z.
3. The receiver now can calculate a pair of (u, v) based on $u = z/s$ and $v = r/s$.
4. The receiver can calculate $uG + vP = kG = R : (r', y')$
5. Finally the receiver checks if $r' = r$, if it dose then the signature is verified.

Let's dive in....

Think about the following three questions:

Q1: is there only one pair of (u, v)?

Q2: how to find the s that can satisfy $u = z/s$ and $v = r/s$?

Q3: can the sender send out (r, k) instead of (r, s)?

Q4: can I reuse the k to sign multiple messages?

Let's dive in....

Q1: is there only one pair of (u, v)?

Nope. $uG + vP = kG \Rightarrow vP = (k–u)G$

Since $v \neq 0$, we can divide by the scalar multiple v: $P = ((k–u)/v)G$

If we know e, we have: $eG = ((k–u)/v)G \Rightarrow e = (k–u)/v$

Hence, since the sender knows the private key e, it is quite easy for her to generate as many pairs of qualified (u, v) as she wish.

However, if a person does not know e, it is very difficult to find out a pair of (u, v) that happens to satisfy $uG + vP = kG$.

$uG + vP = kG$ is equivalent to the **discrete log problem**.

Let's dive in....

Q2: how to find the s that can satisfy $u = z/s$ and $v = r/s$?

$$
\begin{aligned}
uG + vP &= R = kG \\
uG + veG &= kG \\
u + ve &= k \\
z/s + re/s &= k \\
(z + re)/s &= k \\
s &= (z + re)/k
\end{aligned}
$$

Here you can see, if you want to easily find out the right s, you also need to know the private key e.

Let's dive in....

Q3: can the sender send out (r, k) instead of (r, s)?

Nope. After receiving (r, k), the receiver can calculate (u, v), then we have:

$$\begin{aligned} uG + vP &= R \\ uG + veG &= kG \\ kG – uG &= veG \\ (k – u)G &= veG \\ (k – u) &= ve \\ (k – u)/v &= e \end{aligned}$$

So the receiver can back out the sender's private key e.

Let's dive in....

Q4: can I reuse the k to sign multiple messages?

Nope, you shouldn't! Suppose you reuse k to sing two messages, whose hash values are z_1 and z_2, then:

$$\begin{aligned} s_1 = (z_1 + re)/k, s_2 = (z_2 + re)/k \\ s_1/s_2 = (z_1 + re)/(z_2 + re) \\ s_1(z_2 + re) = s_2(z_1 + re) \\ s_1re - s_2re = s_2z_1 - s_1z_2 \\ e = (s_2z_1 - s_1z_2)/(rs_1 - rs_2) \end{aligned}$$

Now the receiver can back out the sender's private key e.

Programming: Sign and Verification

Program 1: verify whether the signatures are valid.

```
P = (0x887387e452b8eacc4acfde10d9aaf7f6d9a0f975aabb10d006e4da568744d06c,
0x61de6d95231cd89026e286df3b6ae4a894a3378e393e93a0f45b666329a0ae34)

# signature 1
z = 0xec208baa0fc1c19f708a9ca96fdeff3ac3f230bb4a7ba4aede4942ad003c0f60
r = 0xac8d1c87e51d0d441be8b3dd5b05c8795b48875dffe00b7ffcfac23010d3a395
s = 0x68342ceff8935ededd102dd876ffd6ba72d6a427a3edb13d26eb0781cb423c4

# signature 2
z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d
r = 0xeff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c
s = 0xc7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6
```

Let's code it out in python.

Programming: Sign and Verification

Program 2: Sign a message.

```
# Sign the following message z with the private key e:

e = 12345
z = int.from_bytes(hash256('Programming Bitcoin!'), 'big')
```

Let's code it out in python.

V. From Keys to Addresses

Public Key, Address and Signature

In this section, we discuss the following three topics:

- Public Key Serialization & Compression
- Address Generation
- Signature Serialization

Public Key Serialization

Bitcoin uses Elliptic Curve Digital Signature Algorithm (**ECDSA**) to create secure key pairs (private and public keys) that enable users to sign transactions and authenticate messages securely.

The raw public key is just two large integers (x, y), and different systems may interpret or store them in various ways. This can lead to compatibility issues. So we introduce the concept of **serialization**. Serialization not only addresses incompatibility issues but also reduces the data payload in transactions.

There is a standard for serializing ECDSA public keys, called Standards for Efficient Cryptography (**SEC**), and there are two forms of SEC format: *uncompressed* and *compressed*.

Public Key: Uncompressed SEC Format

Here is how the uncompressed SEC format for a given point $P = (x, y)$ is generated:

- Start with the prefix byte, which is $0x04$.
- Next, append the x coordinate in 32 bytes.
- Finally, append the y coordinate in 32 bytes

Total Length: 65 bytes (1-byte prefix + 32-byte x-coordinate + 32-byte y-coordinate).

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

- 04 - Marker
- x coordinate - 32 bytes
- y coordinate - 32 bytes
```

Figure 3: Uncompressed SEC format

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Public Key: Compressed SEC Format

On the elliptic curve, there are the only two solutions for a given x, so if we know x, we know the y coordinate has to be either y or $p–y$ (recall: $–y\%p = (p–y)\%p$).

Since p is a prime number greater than 2, we know that p is odd. Thus, if y is even, $p–y$ must be odd, vice versa.

Hence, the x **coordinate** and the **evenness of the** y **coordinate** are sufficient to describe the point (x, y), and this is the idea behind the compressed SEC format.

Public Key: Compressed SEC Format

Here is the serialization of the compressed SEC format for a given point $P = (x, y)$:

- Start with the prefix byte. If y is even, it's $0x02$; otherwise, it's $0x03$.
- Next, append the x coordinate in 32 bytes.

Total Length: 33 bytes (1-byte prefix + 32-byte x-coordinate).

```
0349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a

        - 02 if y is even, 03 if odd - Marker
        - x coordinate - 32 bytes
```

Figure 4: Compressed SEC format

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Public Key: Compressed SEC Format

One more question!

Calculating y given the x coordinate requires us to calculate a square root in a finite field. How to do that?

It turns out that if the finite field prime $p\%4 = 3$, the solution is easy:

$p\%4 = 3 \Rightarrow (p+1)\%4 = 0 \Rightarrow (p+1)/4$ is an integer.

Public Key: Compressed SEC Format

Let's try to solve $w =?$ if $w^2 = v$:

1. From Fermat's little theorem: $w^{p-1}\%p = 1$, we can get that: $w^2 = w^2 \cdot 1 = w^2 \cdot w^{p-1} = w^{p+1}$
2. Since p is odd (recall p is prime), we know we can divide $p+1$ by 2 and still get an integer, implying: $w = w^{(p+1)/2}$
3. Now we can rewrite it as: $w = w^{(p+1)/2} = w^{2(p+1)/4} = (w^2)^{(p+1)/4} = v^{(p+1)/4}$
4. There are two possible $w's$: $w = v^{(p+1)/4}$ or $w = p - v^{(p+1)/4}$

No surprise, the p used in secp256k1 is such that $p\%4 == 3$, so we can use the above formula (i.e., $w^2 = v \Rightarrow w = v^{(p+1)/4}$ or $p - v^{(p+1)/4}$) in Bitcoin.

Convert Public Key into Address

1. **Obtain the Public Key:**
   - Use either the compressed or uncompressed public key.
2. **SHA-256 Hash:**
   - Compute the SHA-256 hash of the public key.
   - sha256_hash = SHA256(public_key)
3. **RIPEMD-160 Hash:**
   - Compute the RIPEMD-160 hash of the SHA-256 hash.
   - ripemd160_hash = RIPEMD160(sha256_hash)
   - **Result:** A 20-byte public key hash (PKH).
4. **Add Version Byte:**
   - Prepend the version byte to the PKH.
   - **Version Byte for Mainnet:** $0x00$ ($0x6f$ for testnet)
   - versioned_payload = $0x00$ + ripemd160_hash

Convert Public Key into Address

5. **Compute the Checksum:**
   - Compute the double SHA-256 hash of the versioned payload.
   - checksum_full = SHA256(SHA256(versioned_payload))
   - Extract the first 4 bytes as the checksum.
   - The **checksum** helps detect errors in the address.

6. **Create the Binary Address:**
   - Append the checksum to the versioned payload.
   - binary_address = versioned_payload + checksum

7. **Base58Check Encoding:**
   - Encode the binary address using Base58Check encoding.
   - **Alphabet:** Uses Bitcoin's Base58 alphabet (omitting easily confused characters).
   - bitcoin_address = Base58CheckEncode(binary_address)
   - **Base58Check encoding** ensures the address is compact and user-friendly.

Signature Serialization

The standard for serializing signatures is called Distinguished Encoding Rules (**DER**) format.

DER signature format is defined like this:

1. Start with the $0x30$ byte.
2. Encode the length of the rest of the signature (usually $0x44$ or $0x45$) and append.
3. Append the marker byte, $0x02$.
4. Encode r, prepend it with the $0x00$ byte if r's first byte $\geq 0x80$ to indicate it is a positive integer.
5. Append the marker byte, $0x02$.
6. Encode s, prepend with the $0x00$ byte if s's first byte $\geq 0x80$ to indicate it is a positive integer.

Signature Serialization

```
3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf213
20b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801
c31967743a9c8e10615bed
```

- 30 - Marker
- 45 - Length of sig
- 02 - Marker for r value
- 21 - r value lenth
- 00ed...8f - r value
- 02 - Marker for s value
- 20 - s value length
- 7a98...ed - s value

Figure 5: Signature Serialization: DER Format

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Extra: Private Key Compression

Generally, we are not going to need to serialize our private key that often, as it does not get broadcast. If you do want to compress your private key, you can use Wallet Import Format (WIF).

Here is how the WIF format is created:

1. For mainnet, start with the prefix $0x80$, for testnet $0xef$.
2. Encode the secret in 32-byte big-endian.
3. If the SEC format used for the public key address was compressed, add a suffix of $0x01$.
4. Combine the prefix from step 1, serialized secret from step 2, and suffix from step 3.
5. Do a hash256 of the result from step 4 and get the first 4 bytes.
6. Take the combination of step 4 and step 5 and encode it in Base58.