Intro to ZKPs

👁️ 🏃‍♀️ 🐟

Elena Nadolinski
@leanthebean

IRON FISH

Where to get these slides? 🔮

Twitter: @leanthebean

(I'll post these slides after this presentation)

Goal of this presentation

1. To give you the ammunition needed to learn about zero-knowledge proofs
2. To give an overview of the proof systems we have today (and which ones you can use today fairly easily)
3. And to give you a sense in which direction research is headed

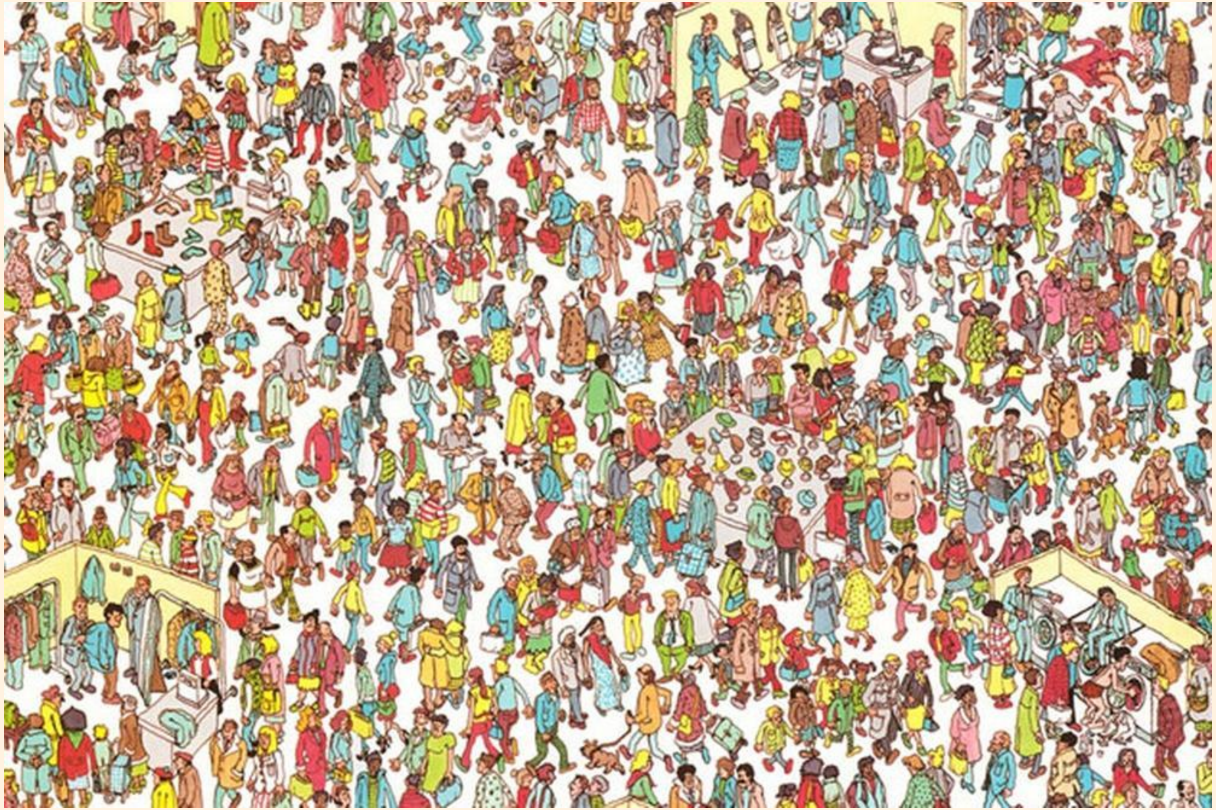
What we’ll go over

1. Brief history / background of relevant cryptography
2. Quick overview of existing relevant ZKP proof systems
3. Highlight ones you should care about more :)

What is a ZKP?

What is a ZKP?

The ability to prove honest computation without revealing inputs



ZKPs == honest computation

ZKPs == honest computation

Used for:

Scalability

Privacy

🧰🛠️⚙️ Cryptography Tools

🧰⚒️⚙️ Cryptography Tools

(This might be a lot, and maybe overwhelming, but it'll give you a good foundation to research & study ZKPs on your own)

Modular Math

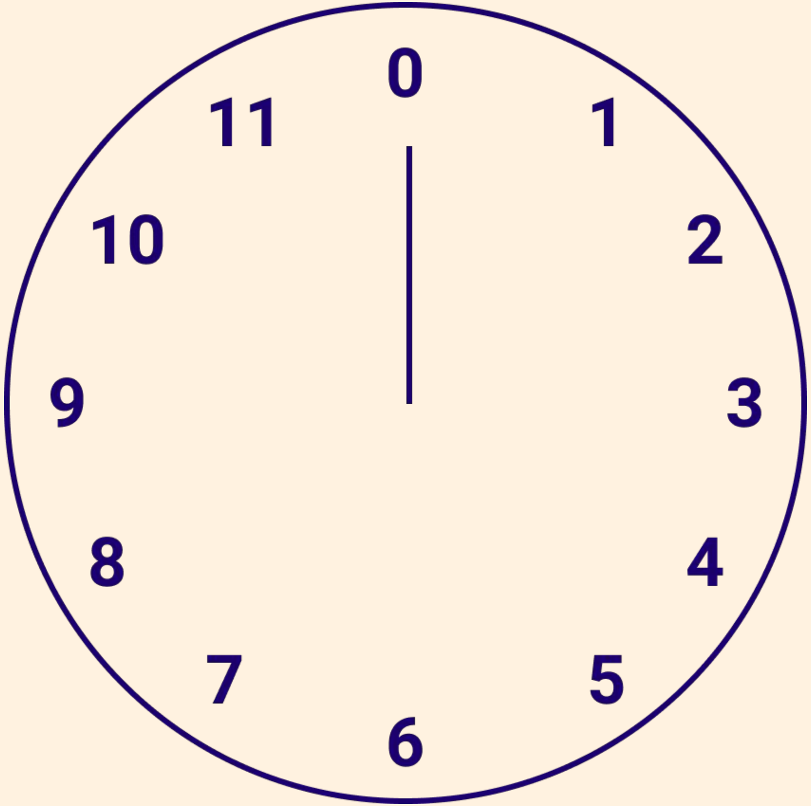
The mod operator creates ***cycles***, an overused feature for public-private key cryptography:

22 (mod 12) = 10

(Because 12 divides 22 one time evenly with 10 as the remainder)

3 + 16 (mod 12) = 7

And so on



Symmetric Encryption

- Simply the concept that given a *message* and a *key*, that *message* can be **encrypted and decrypted using the same *key***
- Substitution ciphers are one of the oldest examples (as early as 400 BC between lovers in India 🥰)
- The most common modern family of symmetric encryption is AES
- Currently the evolving standard for TLS encryption is ChaCha20-Poly1305 AEAD ('AE' refers to authenticated encryption)
- ***Both the sender and the receiver of the encrypted message must have the key in order to encrypt and decrypt the message – how do they share that key over an unsecure communications channel?***

Asymmetric Encryption

- Whitfield Diffie obsessed over that question endlessly in the 70s, travelling cross-country to find others to work on this question, until he made his way over to Stanford and met Martin Hellman.
- Together, they were the first to publicly publish (but not the first to discover!) a form of *asymmetric encryption* that allows a message to be ***encrypted using one key, but decrypted using a different one!***
- Essentially secret sharing of one key used to encrypt/decrypt a message
- ***<u>Let’s see how it works!</u>***

Diffie-Hellman (1976)

First, there is some setup. Everyone publicly agrees on a modulus (**p**) and some base (**g**). Let's have the base be 5, and the modulus be 23.

p = 23 (modulus) **g = 5** (base)

Diffie-Hellman (1976)

p = 23 (modulus) **g = 5** (base)

Alice chooses a private key $a = 3$, and sends Bob her public key

$$A = g^a \bmod p$$

$$A = 5^3 \bmod 23 = 10$$

Diffie-Hellman (1976)

p = 23 (modulus) **g = 5** (base)

Alice chooses a private key $a = 3$, and sends Bob her public key

$A = g^a \bmod p$

$A = 5^3 \bmod 23 = 10$

Bob chooses a private key $b = 5$, and sends Alice his public key

$B = g^b \bmod p$

$B = 5^5 \bmod 23 = 20$

Diffie-Hellman (1976)

Alice sends Bob her public key (**A**) and Bob sends her *his* public key (**B**)

Alice computes **s** = $\mathbf{B}^{\mathbf{a}}$ mod p

$$\mathbf{s = g^{b\,a} \bmod p = 20^3 \bmod 23 = 19}$$

Diffie-Hellman (1976)

Alice sends Bob her public key (**A**) and Bob sends her *his* public key (**B**)

Alice computes **s** = $\mathbf{B}^{\mathbf{a}}$ mod p

s = $g^{b\,a}$ mod p = 20^{3} mod 23 = 19

Bob computes **s** = $\mathbf{A}^{\mathbf{b}}$ mod p

s = $g^{a\,b}$ mod p = 10^{5} mod 23 = 19

Diffie-Hellman (1976)

Alice sends Bob her public key (**A**) and Bob sends her *his* public key (**B**)

Alice computes **s** = $\mathbf{B}^{\mathbf{a}}$ mod p

s = $g^{b\,a}$ mod p = 20^3 mod 23 = 19

Bob computes **s** = $\mathbf{A}^{\mathbf{b}}$ mod p

s = $g^{a\,b}$ mod p = 10^5 mod 23 = 19

*Now Alice and Bob can use **19** as their key to encrypt/decrypt messages*

RSA (1977)

- One of the first public-key cryptosystems
    - ***Allows for a message to be encrypted via the public key*** and ***decrypted using the private key***
- Builds on top of Diffie-Hellman
- (Check out Dusty's RSA tutorial if you're interested in implementing it!)
- ***Let's see how it works!*** 😃

RSA

Setup
Choose 2 primes: **p = 5 q = 11** and multiply them together to compute **N**:

N = p * q = 55

After this step, **p and q should be kept utterly secret or thrown away altogether** to preserve the security of the protocol.

RSA

Setup
Choose 2 primes: **p = 5 q = 11** and multiply them together to compute **N**:

N = p * q = 55

After this step, **p and q should be kept utterly secret or thrown away altogether** to preserve the security of the protocol.

Compute the ***totient*** of N by computing the least common multiplier of p-1 and q-1

$\lambda(N) = \mathrm{lcm}(p-1, q-1) = \mathbf{40}$ (totient)

RSA

Key Generation

When choosing a **public key**, choose a random number, **e**, such that it is:

1. Greater than 1, but less than the totient λ(N)
   1 < e < λ(N)

2. Coprime to the totient (Coprime simply means that the two numbers don't have any common divisors except for 1).

RSA

Key Generation

When choosing a **public key**, choose a random number, **e**, such that it is:

1. Greater than 1, but less than the totient λ(N)

   1 < e < λ(N)

2. Coprime to the totient (Coprime simply means that the two numbers don't have any common divisors except for 1).

Let's pick **e = 7** as our public key. It is 1 < **e** < 40 and coprime to 40 (40 factors into 2, 2, 2, 5, 1, and 7 just factors into 1 since it's a prime number, so the two are coprime to each other as none of their divisors overlap).

RSA

Key Generation

Compute the **private key**, **d**, such that it is:

1. Greater than 1, but less than the totient $\lambda(N)$

   $\mathbf{1 < d < \lambda(N)}$

2. $\mathbf{d * e \pmod{\lambda(N)} = 1}$

For this example we'll go with **d = 23** as our private key

RSA

Encryption / Decryption

When encrypting a message, m, we can use the recipient's public key to encrypt it such that their private key can decrypt it.

Encrypting: $c = m^e \pmod N$

To decrypt a message, the recipient uses their private key, d:

Decrypting: $c^d = m^{ed} = m \pmod N$

RSA

In our example of having a private key **d = 23** and public key **e = 7** let's say we want to encrypt a message where our message, **m**, is **m = 8**

Encrypting:

$c = \mathbf{m}^{\mathbf{e}} \pmod{N}$

$c = 87 \pmod{55} = 2$

Our encrypted message (**m = 8**) is the ciphertext **c = 2**

RSA

Knowing the corresponding ***private key*** (**d = 23**) to the ***public key*** (**e = 7**) used to encrypt the message, we can decrypt the message back to its plaintext value of 8:

Decrypting:

Decrypting $c^d = m^{ed} \pmod{N} = 8$

Discrete Logarithm Problem

Both (modern) Diffie-Hellman and RSA are highly secure protocols due to the ***discrete logarithm problem***

Given the ***public key*** it is *hard* to find out the ***private key.***

Remember in Diffie-Hellman the public key is: $\mathbf{A = g^{a} \bmod p}$

To find the public key, we use "modular exponentiation", for which the reverse operation would require ***discrete logarithm***.

discrete because it involve finite sets (cycles due to the modulus).
logarithm because to reverse the exponentiation we would need to use a log

Fiat-Shamir (1986)

- Interactive proof of knowledge
- “Grandfather” of zero-knowledge proofs
- ***Allows one to prove information about a number, without revealing the number***

Fiat-Shamir

- ***g*** *is a number (called generator) everyone knows*

Fiat-Shamir

- ***g*** *is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$

Fiat-Shamir

- ***g*** *is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends ***t*** such that $t = g^v$

Fiat-Shamir

- ***g*** *is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends t such that $t = g^v$
- Bob pics a random **c**, and sends that to Alice

Fiat-Shamir

- ***g*** *is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends ***t*** such that $t = g^v$
- Bob pics a random **c**, and sends that to Alice
- Alice sends back $r = v - cx$

Fiat-Shamir

- *g is a number (called generator) everyone knows*
- Alice wants to prove that she knows x, such that $y = g^x$
- She picks a random v, and sends t such that $t = g^v$
- Bob pics a random c, and sends that to Alice
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$

Fiat-Shamir

- ***g** is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends t such that $t = g^v$
- Bob pics a random **c**, and sends that to Alice
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$
- Since $g^r y^c = g^{v-cx} g^{xc} = g^v = t$
- **Bob knows that Alice knows the "preimage" x**

Fiat-Shamir Heuristic

- Changes the *interactive* protocol to be ***non-interactive*** by introducing a ***random oracle***
  - A random oracle here can be a hash function

Fiat-Shamir (Interactive)

- ***g** is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends t such that $t = g^v$
- Bob pics a random **c**, and sends that to Alice
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$
- Since $g^r y^c = g^{v-cx} g^{xc} = g^v = t$

Fiat-Shamir (Interactive)

- ***g*** *is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends ***t*** such that $t = g^v$
- **Bob pics a random c, and sends that to Alice**
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$
- Since $g^r y^c = g^{v-cx}\, g^{xc} = g^v = t$

Fiat-Shamir (Interactive)

- ***g** is a number (called generator) everyone knows*
- Alice wants to prove that she knows **x**, such that $y = g^x$
- She picks a random **v**, and sends t such that $t = g^v$
- ~~**Bob pics a random e, and sends that to Alice**~~
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$
- Since $g^r y^c = g^{v-cx}\, g^{xc} = g^v = t$

Fiat-Shamir (Non-Interactive)

- ***g** is a number (called generator) everyone knows*
- Alice wants to prove that she knows $\mathbf{x}$, such that $y = g^x$
- She picks a random $\mathbf{v}$, and sends t such that $t = g^v$
- **Alice computes $\mathbf{c}$, where $\mathbf{c}$ = Hash(g, y, t)**
- Alice sends back $r = v - cx$
- Bob checks $t = g^r y^c$
- Since $g^r y^c = g^{v-cx} g^{xc} = g^v = t$

Shamir Secret Sharing

The core idea behind Shamir Secret Sharing is that given a polynomial of degree *k* (if k =1 it's a line, if k = 2 it's a parabola, and so on), you need k+1 points on the polynomial to rebuild the polynomial and all it's coefficients using interpolation.

Given some function f(x) for some degree k

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots a_kx^k$$

We just need *any* k+1 points from that function to rebuild it.

The ***secret*** that we're hiding is in the a_0 value.

Why is this significant / cool?

Shamir Secret Sharing

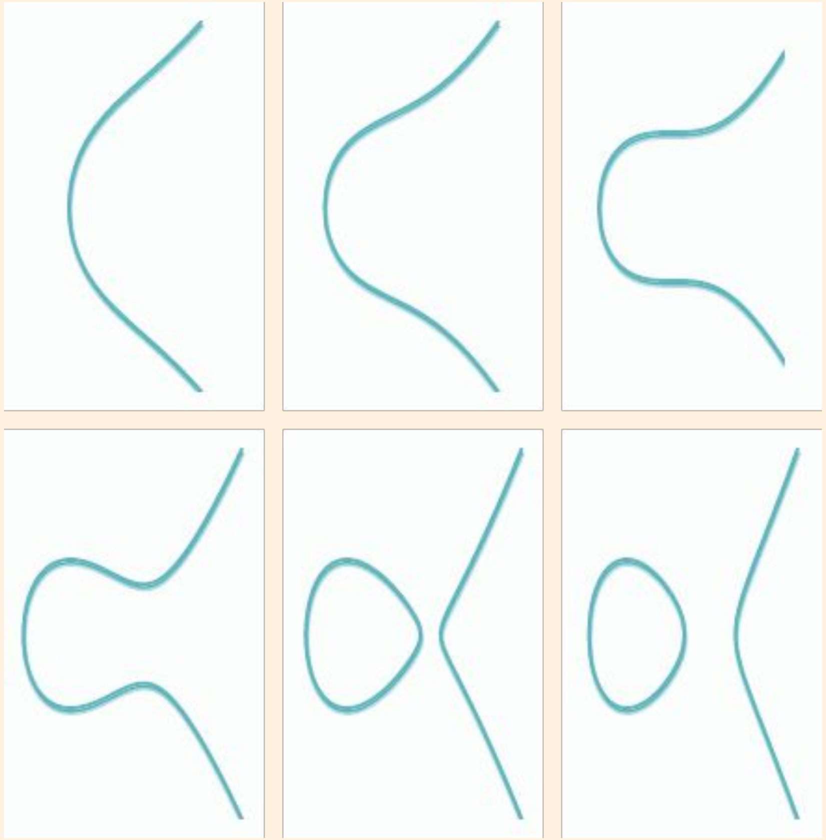
This means that we can put something like a private key in the a_0 value, and construct *m* shards such that you need *n* of *m* shards to retrieve the private key back.

This could potentially be used for key recovery!

Elliptic Curve Cryptography

- Much more ***powerful*** and ***efficient*** tool than exponentiation & modular math

- ***Great tutorial*** by Andrea Corbellini
    - (much of which we'll go over here)



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

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

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

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

(This is the ***Weierstrass*** curve form – the standard is shifting to using elliptic curves using the Montgomery & Edwards curve form)

Elliptic Curve Cryptography

~ ¾ of 100k top websites use ECDHE (https://)
(Elliptic Curve Diffie-Hellman Exchange)

96.1% of those use P256 curve (a NIST standard):

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

Parameters generated from hashing a seed

From a video by Dan Boneh

Elliptic Curve Cryptography

~ ¾ of 100k top websites use ECDHE (https://)
(Elliptic Curve Diffie-Hellman Exchange)

96.1% of those use P256 curve (a NIST standard)

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

Parameters generated from hashing a seed

From a [video by Dan Boneh]

NSA
Conspiracy
Theory

Elliptic Curve Cryptography

Abelian Group Properties for a set $\mathbb{G}$

1) **Closure**: if a and b are members of $\mathbb{G}$ then a + b is also
2) **Associativity**: (a + b) + c = a + (b + c)
3) **Identity**: a + 0 = 0 + a = a
4) **Inverse**: for every a, there exists b such that a + b = 0
5) **Commutativity**: a + b = b + a

ECC - Addition

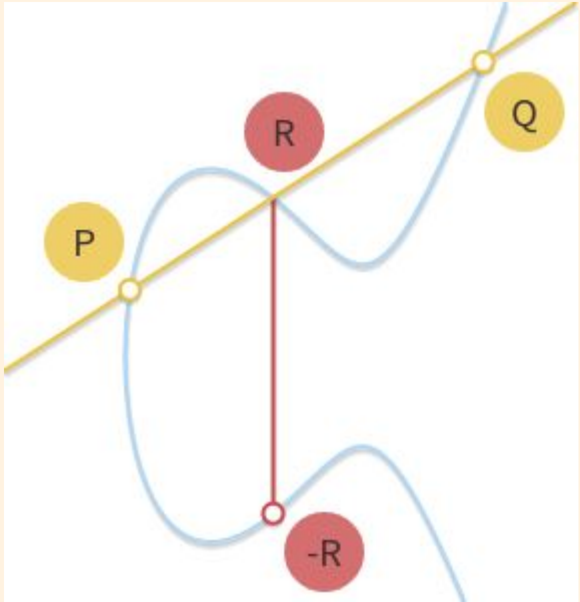
Given 3 aligned non-zero points on an elliptic curve, their sum is: P + Q + R = 0

Since P + Q + R = 0, then P + Q = -R

If we draw a line between 2 points P and Q, it'll intersect at a point R

ECC - Addition

If P + Q + R = 0, then P + Q = -R



If we draw a line between 2 pts P and Q, it'll intersect at a point R

Since P + Q = -R we can easily find R and use it to compute addition of two points from it

ECC - Addition

P + Q = -R

If you remember from your algebra days, given two points, we can find the slope of the line formed between them (slope == m)

If P and Q are two distinct points, the slope is:

$$\mathbf{m = (y_P - y_Q) / (x_P - x_Q)}$$

If P and Q are the same point (P == Q) then the slope is:

$$\mathbf{m = (3x_P{}^3 + a) / 2y_P}$$

ECC - Addition

$P + Q = -R$

Slope: $\mathbf{m = (y_P - y_Q) / (x_P - x_Q)}$ (or $\mathbf{m = (3x_P{}^3 + a) / 2y_P}$ if $P == Q$)

The intersection of this line with the elliptic curve is a third point $R = (x_R, y_R)$

$$\mathbf{x_R = m^2 - x_p - x_Q}$$

$$\mathbf{y_R = y_P + m(x_R - x_P) = y_Q + m(x_R - x_Q)}$$

ECC - Addition

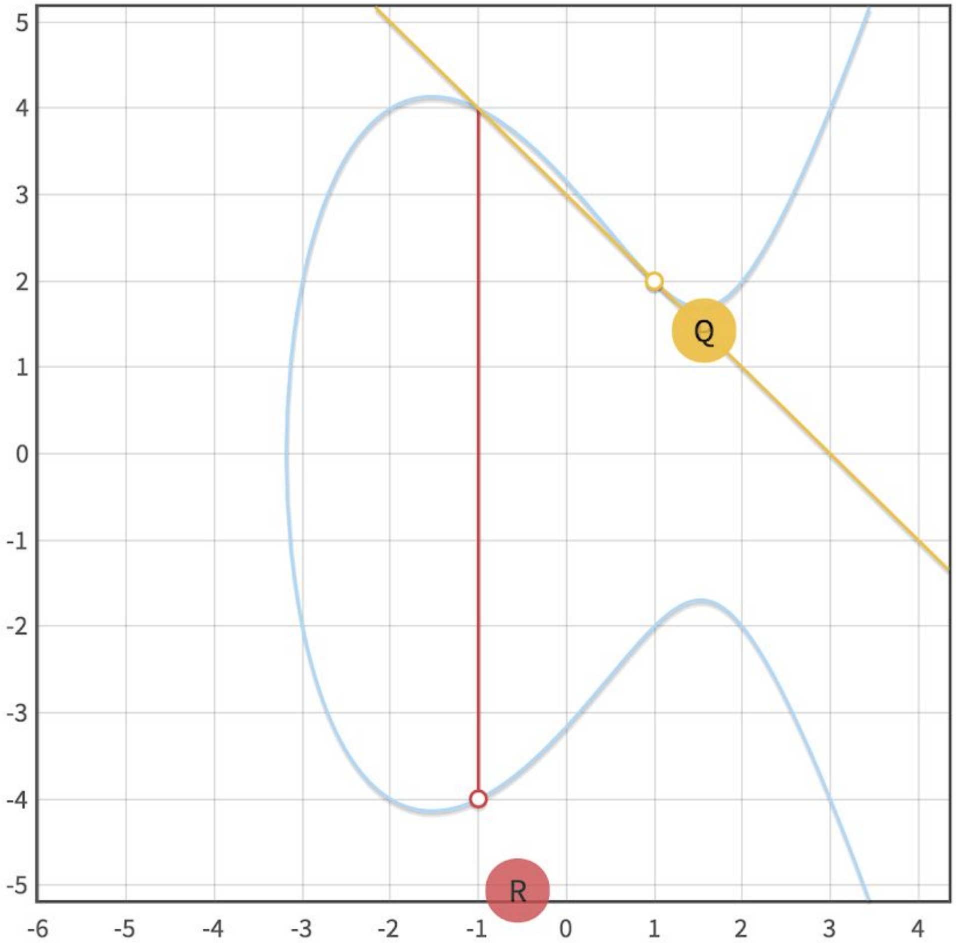
$P + P = -R$ *for* $Q = P = (1, 2)$ *on* $y^2 = x^3 - 7x + 10$

$m = (3x_P{}^3 + a) / 2y_P$ $= -1$

$x_R = m^2 - x_p - x_Q$ $= -1$

$y_R = y_P + m(x_R - x_P) = y_Q + m(x_R - x_Q)$ $= 4$

So $P + P = (-1, -4)$



Curve:	a -7	b 10
P:	x 1	y 2
Q:	x 1	y 2
$R = P + Q$:	x -1	y -4

Point addition over the elliptic curve $y^2 = x^3 - 7x + 10$ in $\mathbb{R}$.

ECC - Multiplication

Now that we know how to add P + P on an elliptic curve, we can add P to itself ***n*** times:

nP = P + P + … + P

n times

(Note that while P is a point on a curve, ***n*** here is called a scalar – it is NOT a point on a curve)

(We have clever optimization techniques to do this fast)

ECC - Multiplication

nP = P + P + … + P

n times

nP = Q : fairly easily to compute
n = Q/P : very hard to compute (no efficient method)

Logarithm Problem (it's not called the division problem for conformity reasons as the solution for modular exponentiation claimed the term first)

ECC - Multiplication

$nP = \underbrace{P + P + \ldots + P}_{\text{n times}}$

$nP = Q$: fairly easily to compute
$n = Q/P$: very hard to compute (no efficient method)

Logarithm Problem (it's not called the division problem for conformity reasons as the solution for modular exponentiation claimed the term first)

This isn't yet a ***discrete logarithm problem*** because we don't yet have cycles!

ECC - Finite Fields & Discrete Log

Re-introducing the modulus!!!

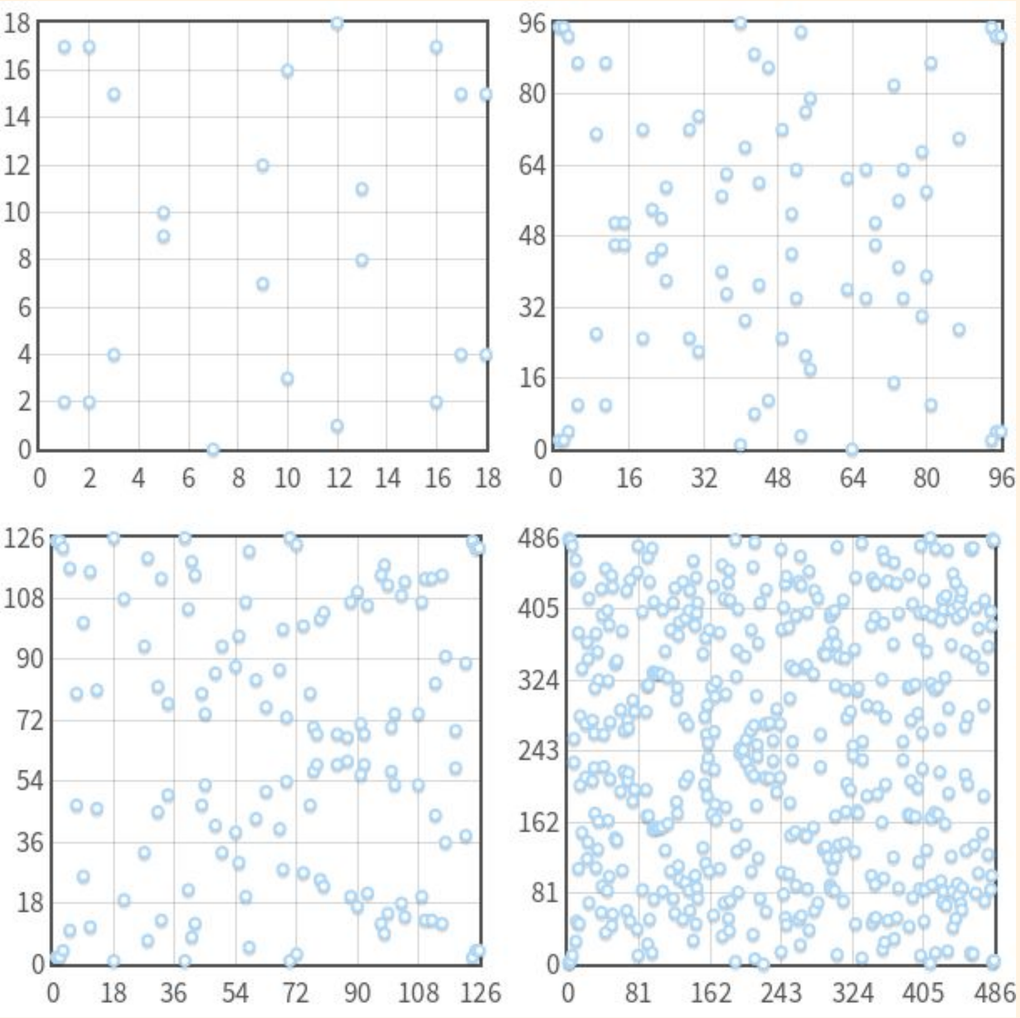
Finite field $\mathbb{F}p$: set of elements (mod p) where p is prime

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

Let's look at the curve we used previously $y^2 = x^3 - 7x + 10$ ***but now with a modulus!***

p = 19

p = 97



$$y^2 = x^3 - 7x + 10$$

p = 127

p = 487

ECC - Finite Fields (Addition)

P + P = ? in a finite field

ECC - Finite Fields (Addition)

P + P = ? in a finite field

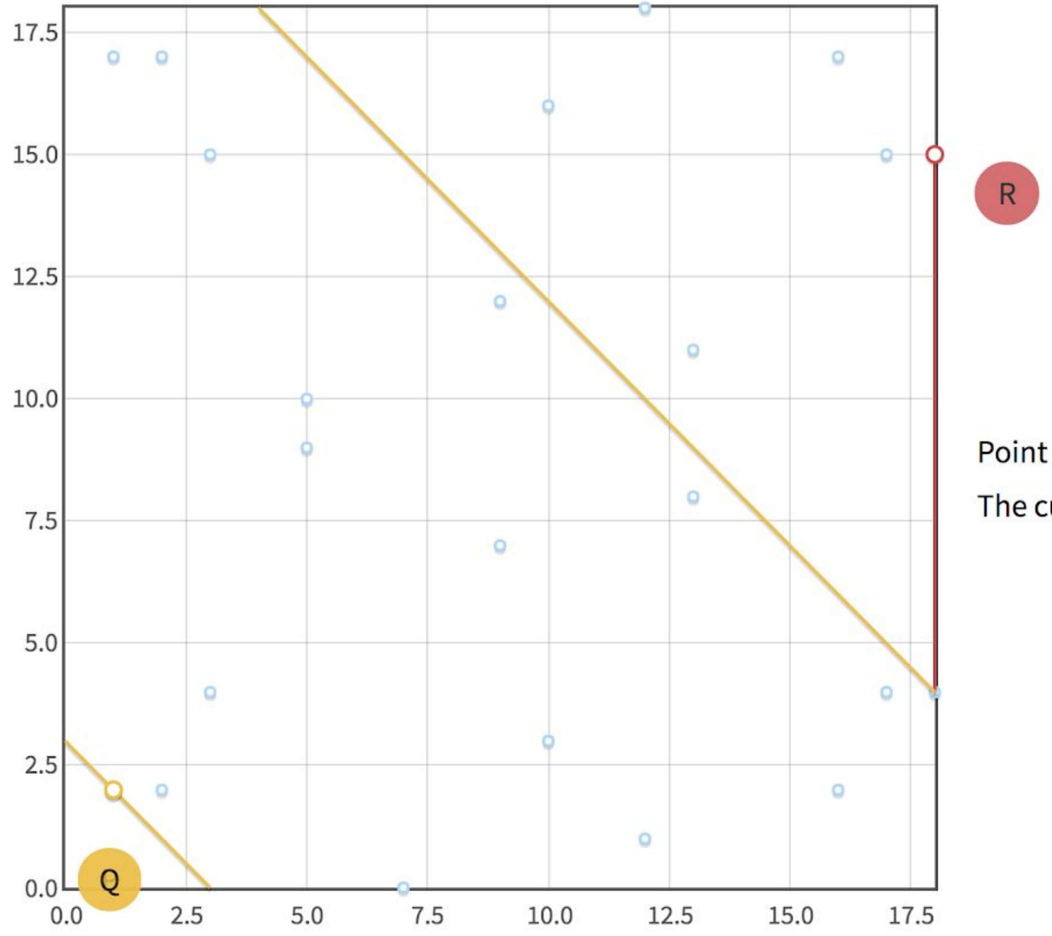
P + P = (-1, -4) on $y^2 = x^3 - 7x + 10$

For a finite field on p = 19

-1 (mod 19) = 18

-4 (mod 19) = 15

So P + P on $y^2 = x^3 - 7x + 10$ (mod 19) = (18, 15)



Curve:	a -7	b 10
Field:	p 19	
P:	x 1	y 2
Q:	x 1	y 2
$R = P + Q$:	x 18	y 15

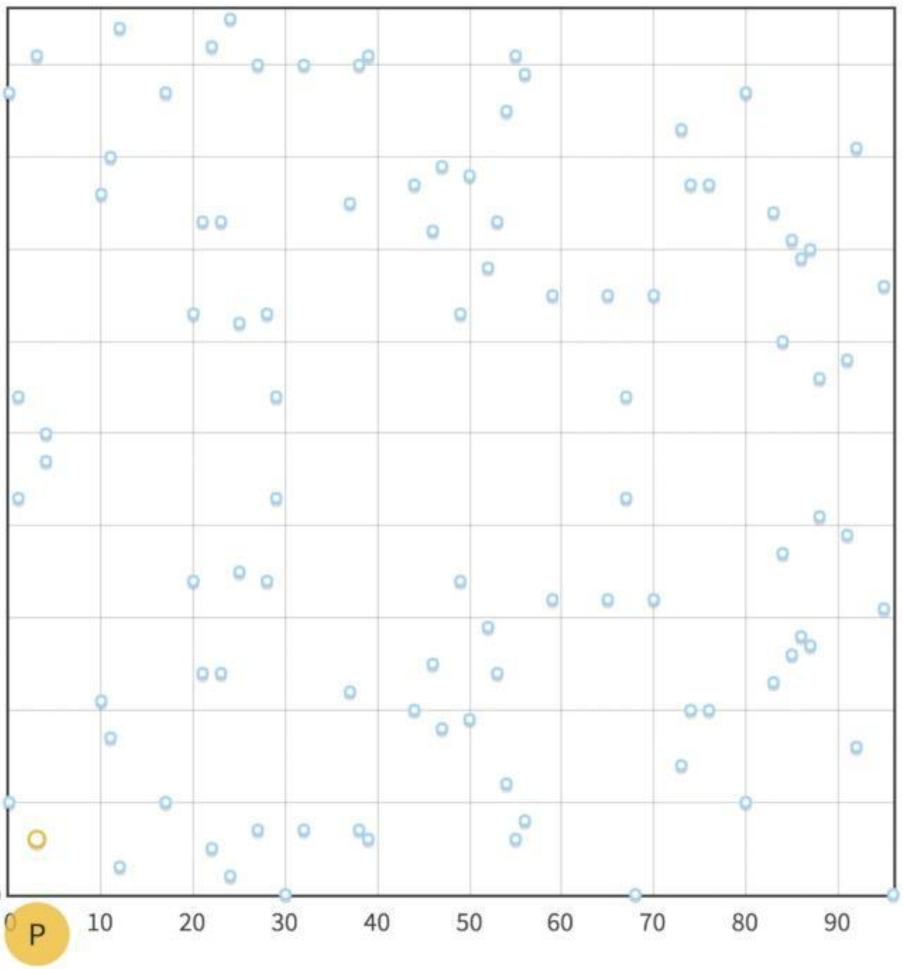
Point addition over the elliptic curve $y^2 = x^3 - 7x + 10$ in $\mathbb{F}_{19}$.

The curve has 24 points (including the point at infinity).

ECC - Groups

$y^2 = x^3 + 2x + 3 \pmod{97}$ at point $P(3, 6)$

Let's calculate some multiples of P



Curve: a 2 b 3

Field: p 97

n: n 0

P: x 3 y 6

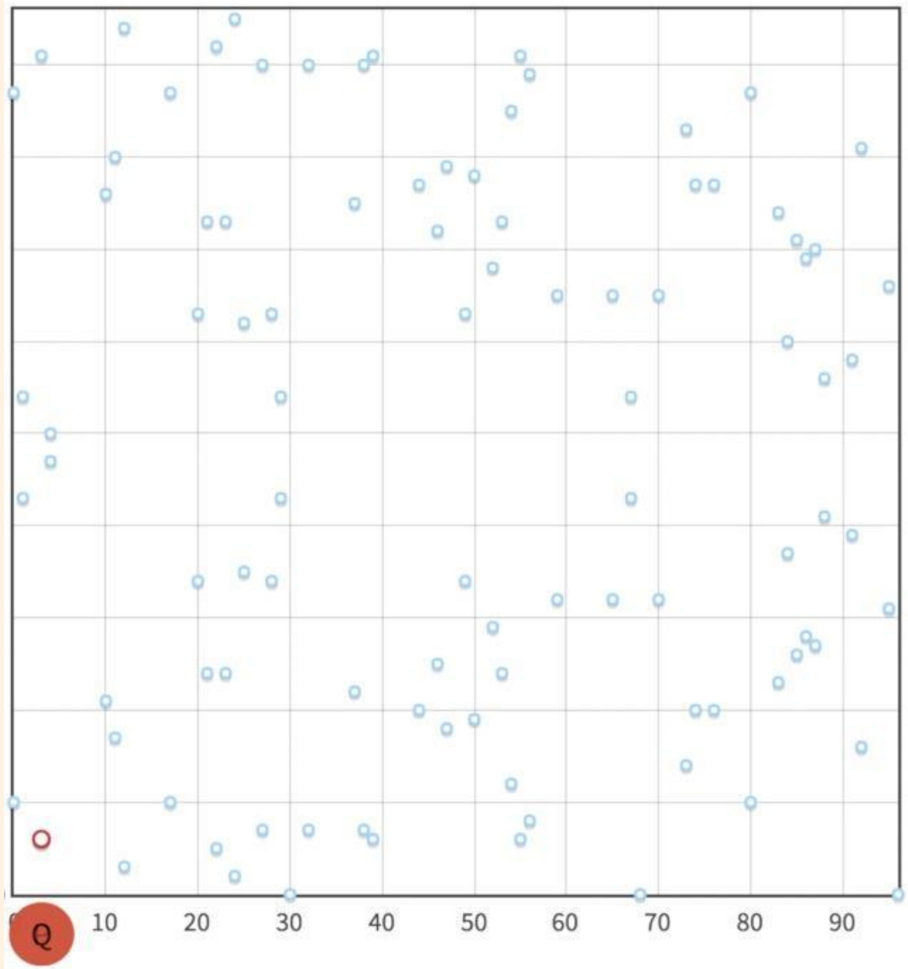
$Q = n \cdot P$: x Inf y Inf

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 0



Curve: a 2 b 3

Field: p 97

n: n 1

P: x 3 y 6

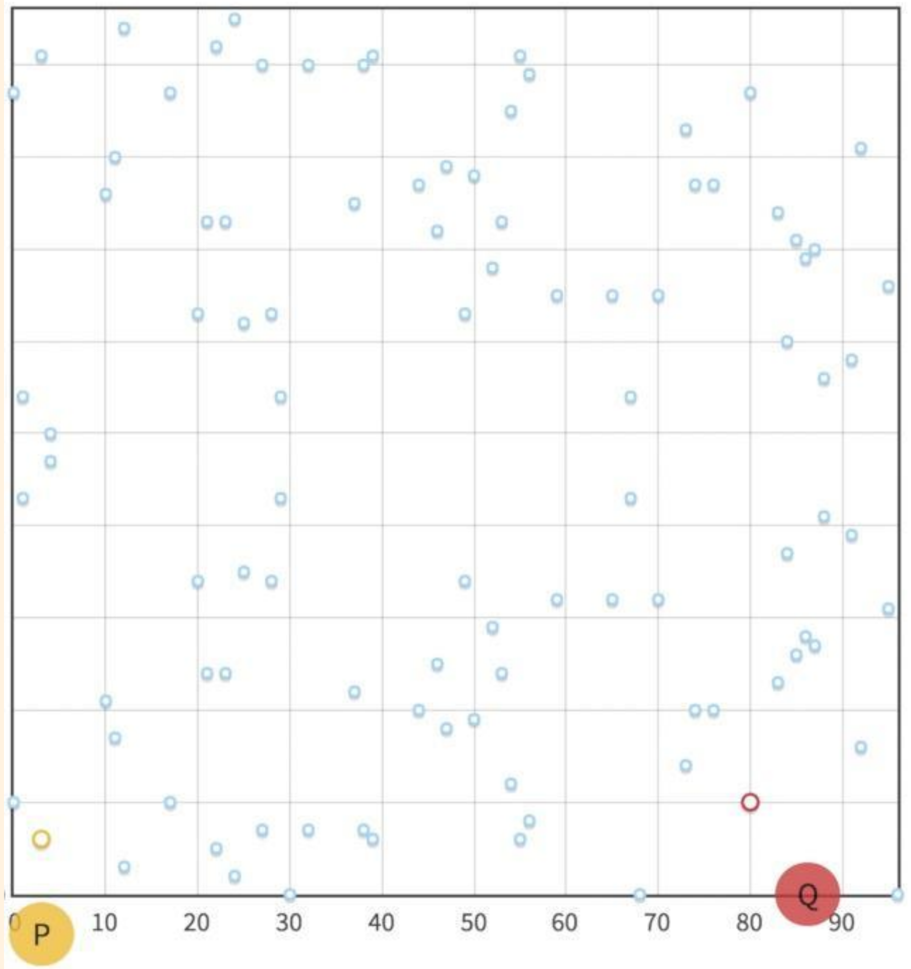
$Q = n \cdot P$: x 3 y 6

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 1



Curve: a 2 b 3

Field: p 97

n: n 2

P: x 3 y 6

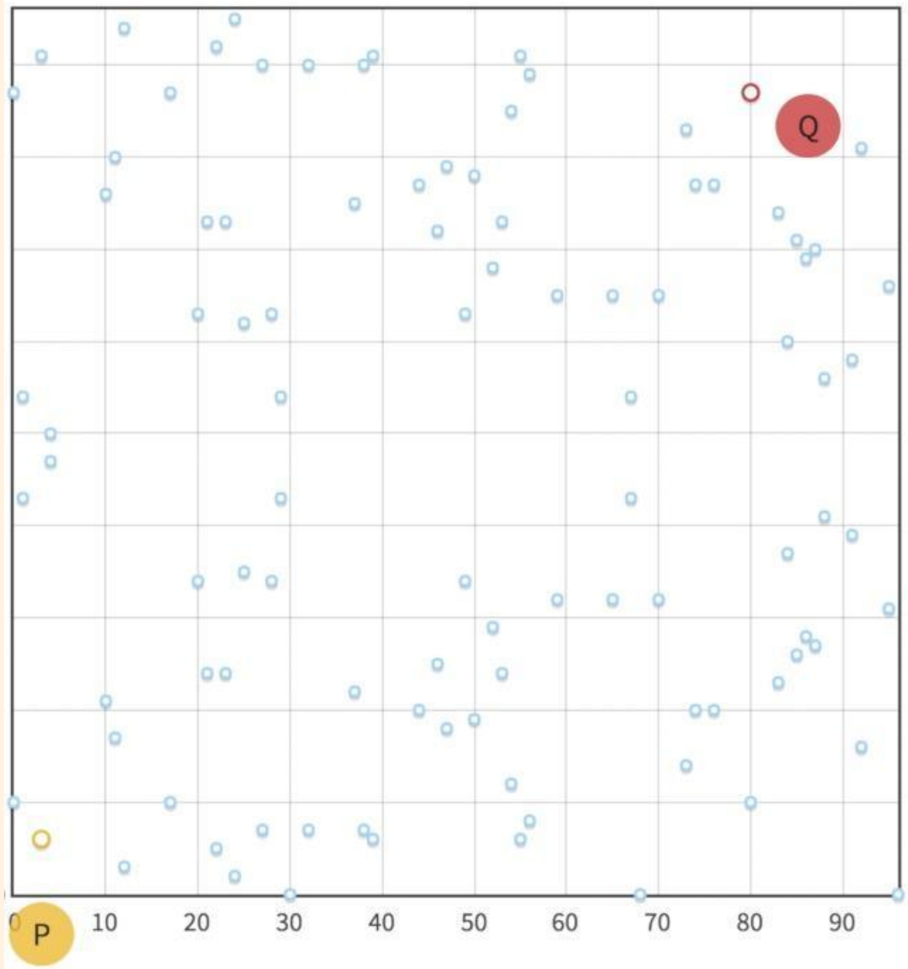
$Q = n \cdot P$: x 80 y 10

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 2



Curve: a 2 b 3

Field: p 97

n: n 3

P: x 3 y 6

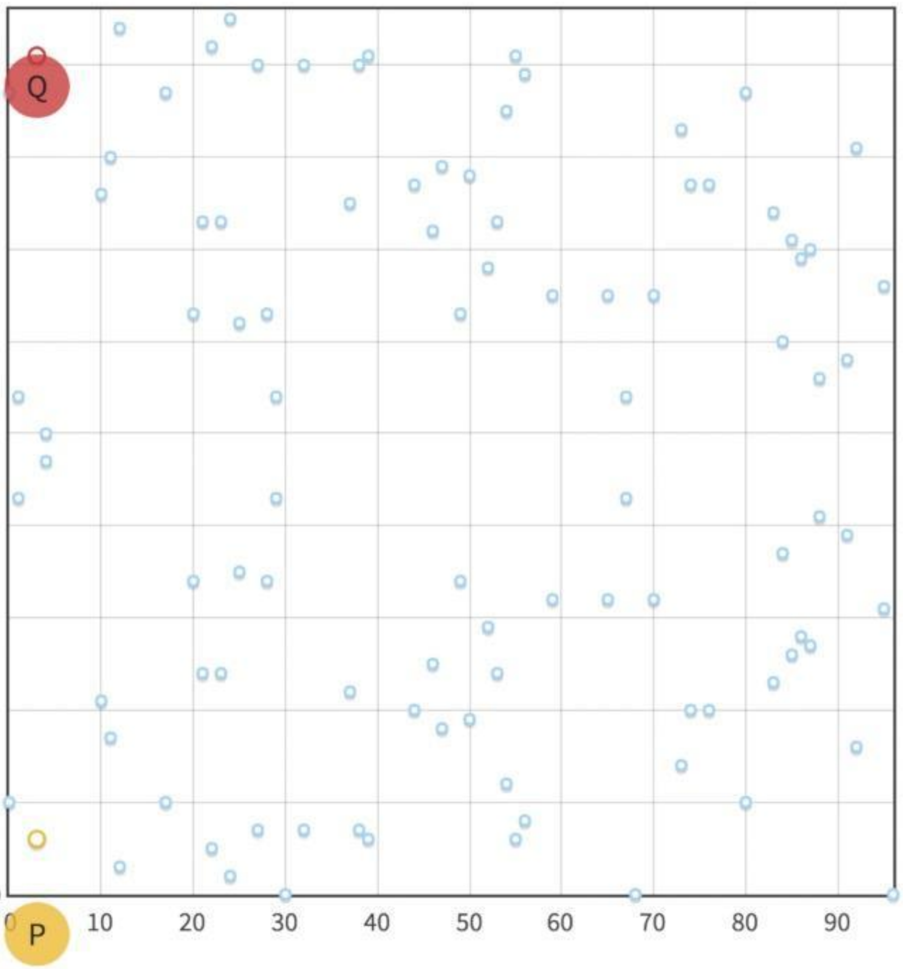
$Q = n \cdot P$: x 80 y 87

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 3



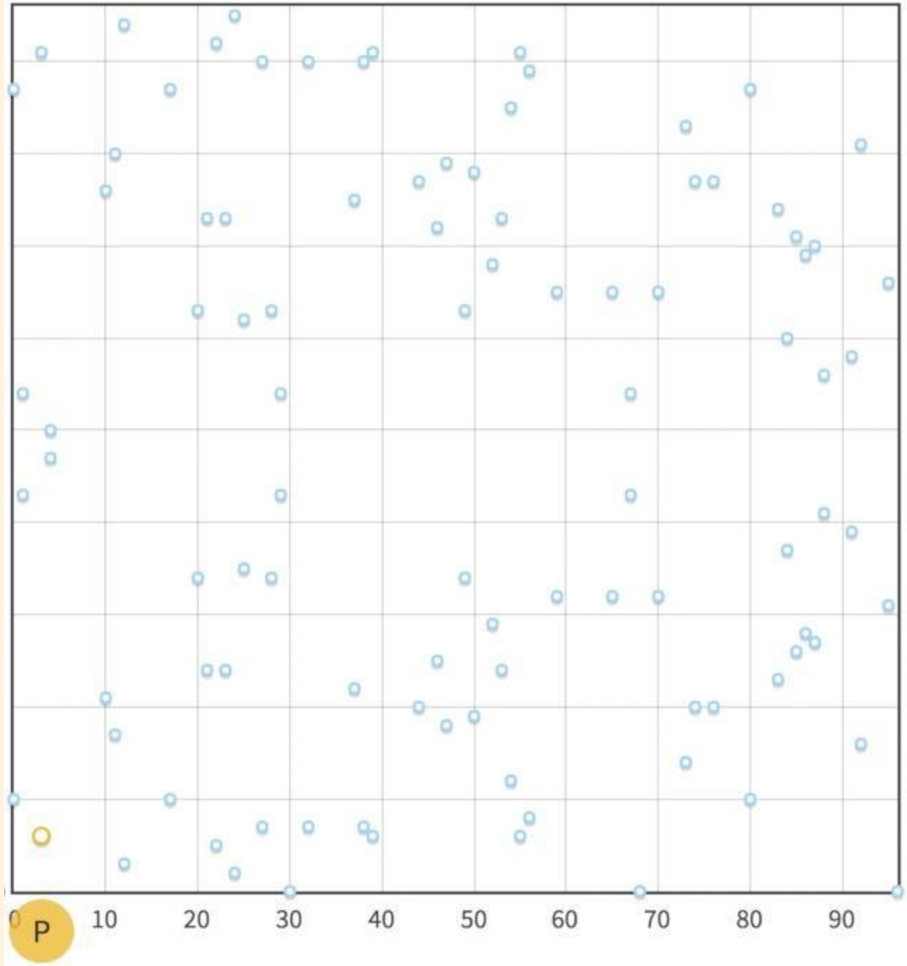
Curve:	a 2	b 3
Field:	p 97	
n:	n 4	
P:	x 3	y 6
$Q = n \cdot P$:	x 3	y 91

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 4



Curve: a 2 b 3

Field: p 97

n: n 5

P: x 3 y 6

$Q = n \cdot P$: x Inf y Inf

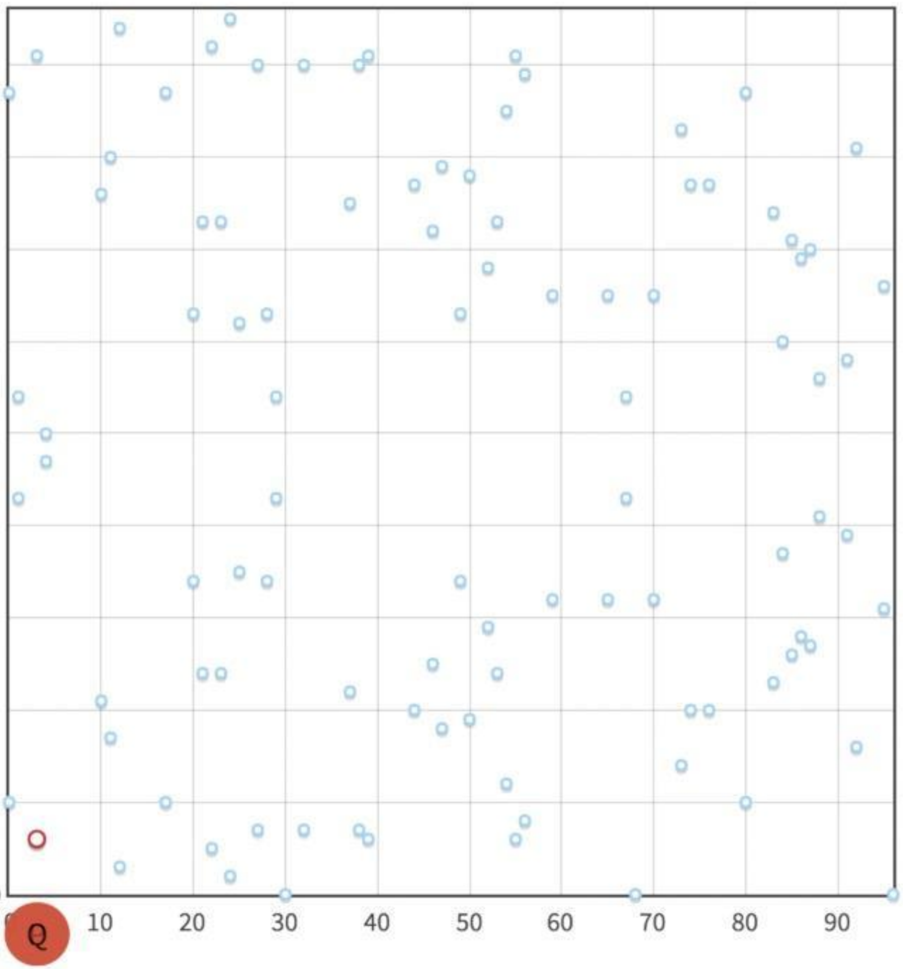
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 5

Same as n = 0



Curve:	a 2	b 3
Field:	p 97	
n:	n 6	
P:	x 3	y 6
$Q = n \cdot P$:	x 3	y 6

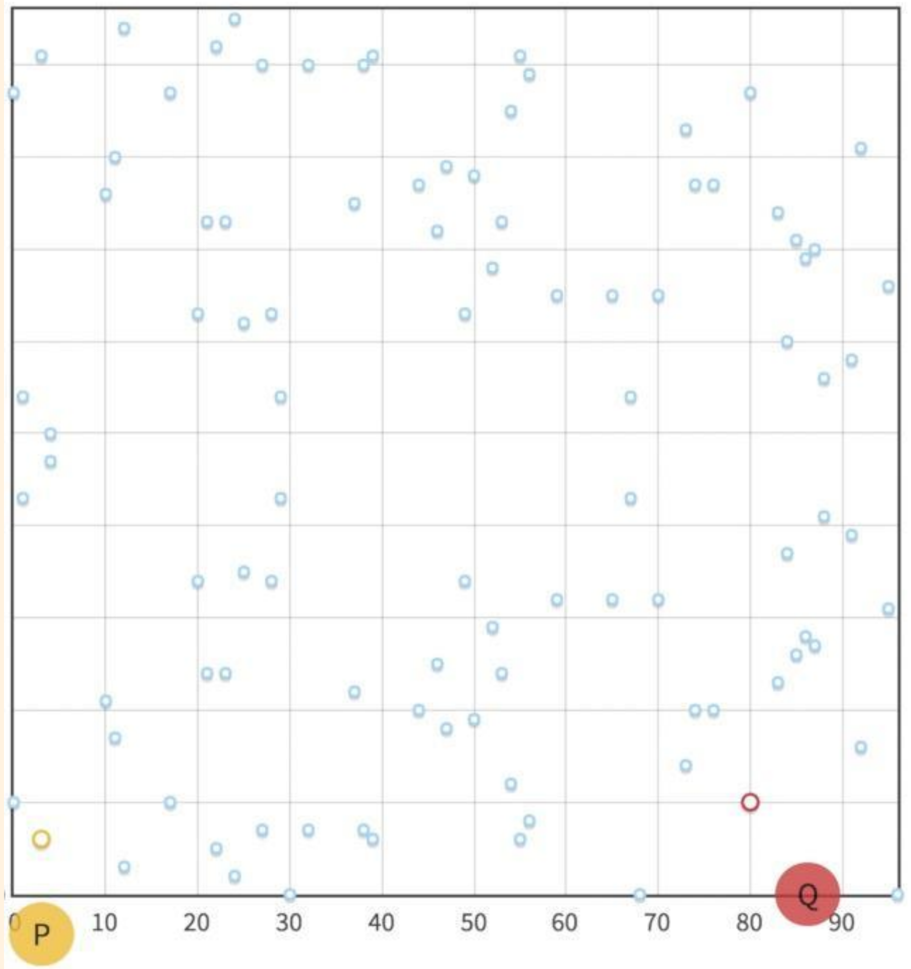
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 6

Same as n = 1



Curve: a 2 b 3

Field: p 97

n: n 7

P: x 3 y 6

$Q = n \cdot P$: x 80 y 10

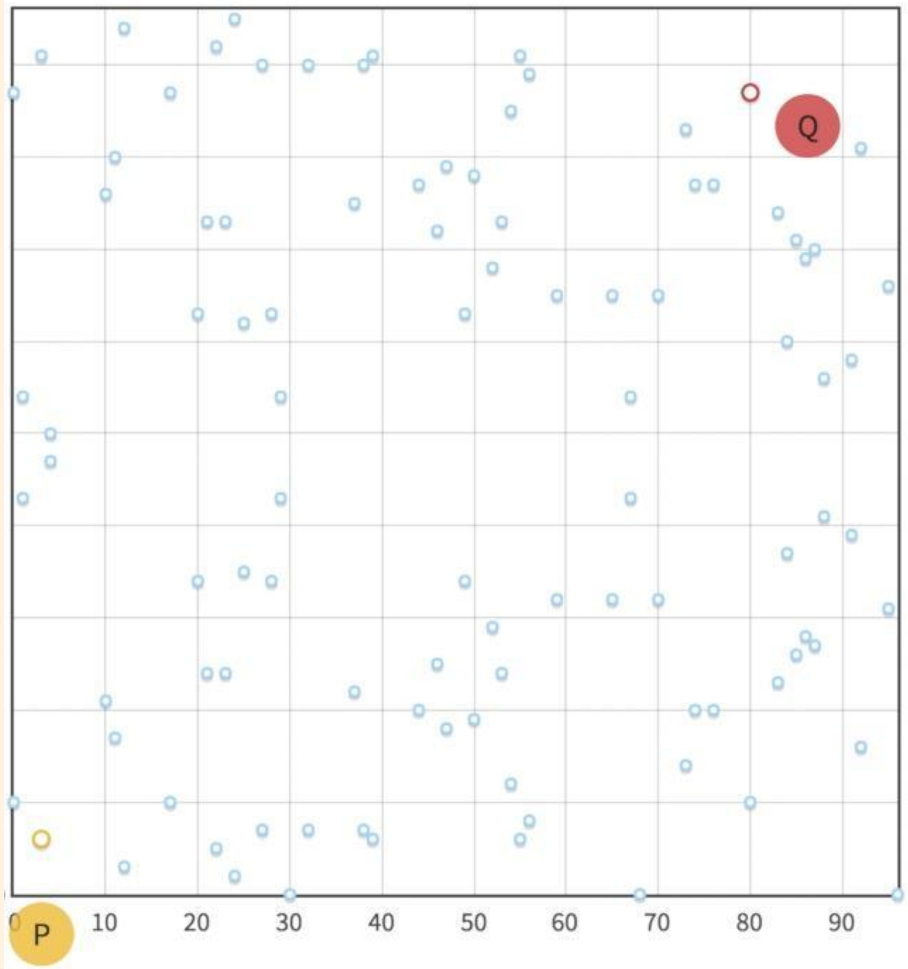
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 7

Same as n = 2



Curve:	a 2	b 3
Field:	p 97	
n:	n 8	
P:	x 3	y 6
$Q = n \cdot P$:	x 80	y 87

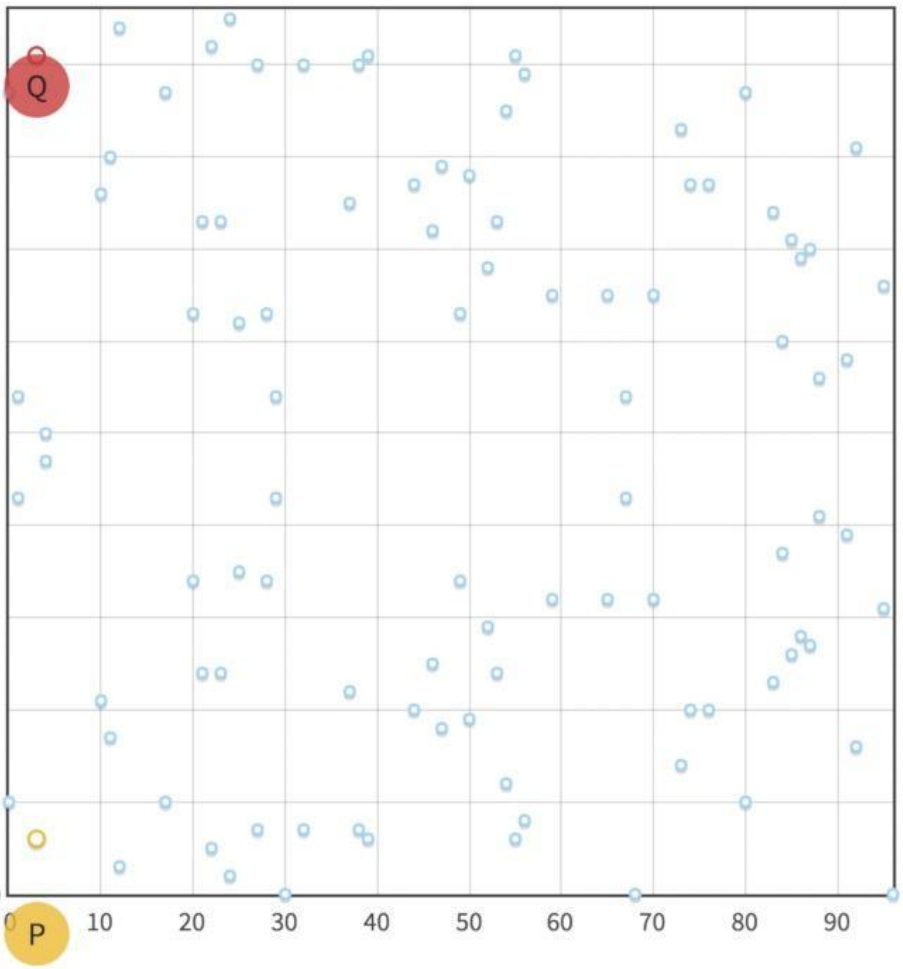
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 8

Same as n = 3



Curve:	a 2	b 3
Field:	p 97	
n:	n 9	
P:	x 3	y 6
$Q = n \cdot P$:	x 3	y 91

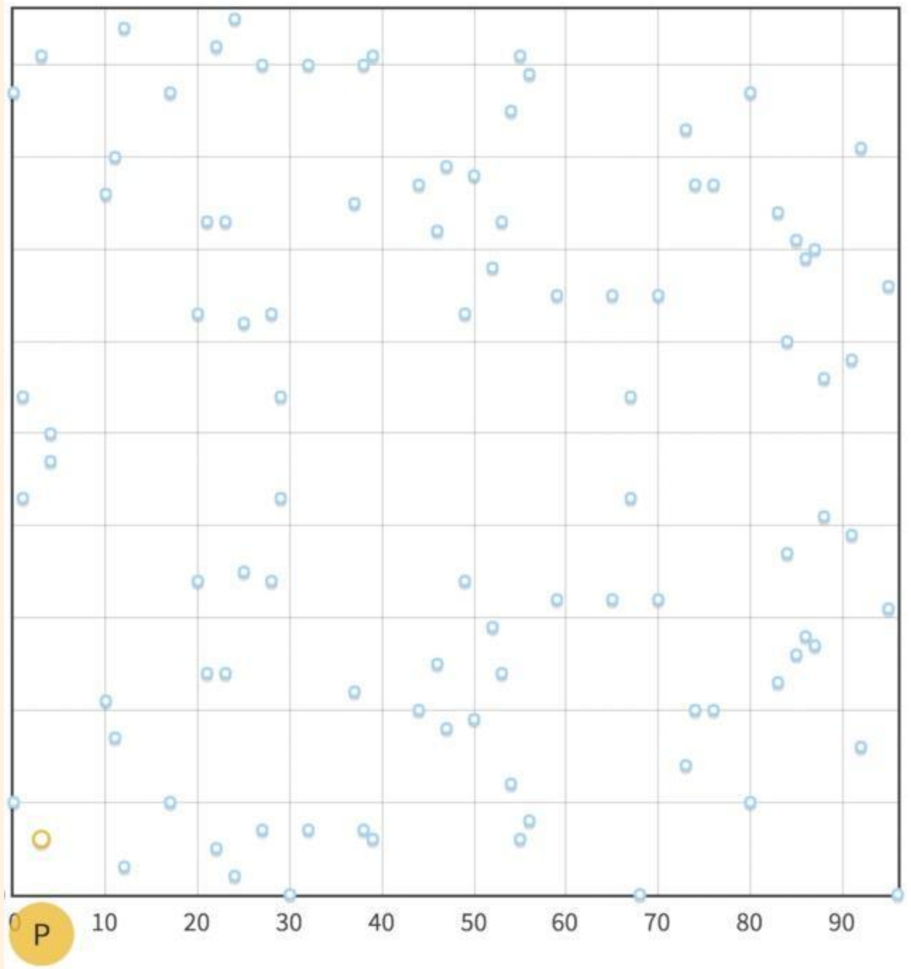
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 9

Same as n = 4



Curve:	a 2	b 3
Field:	p 97	
n:	n 10	
P:	x 3	y 6
$Q = n \cdot P$:	x Inf	y Inf

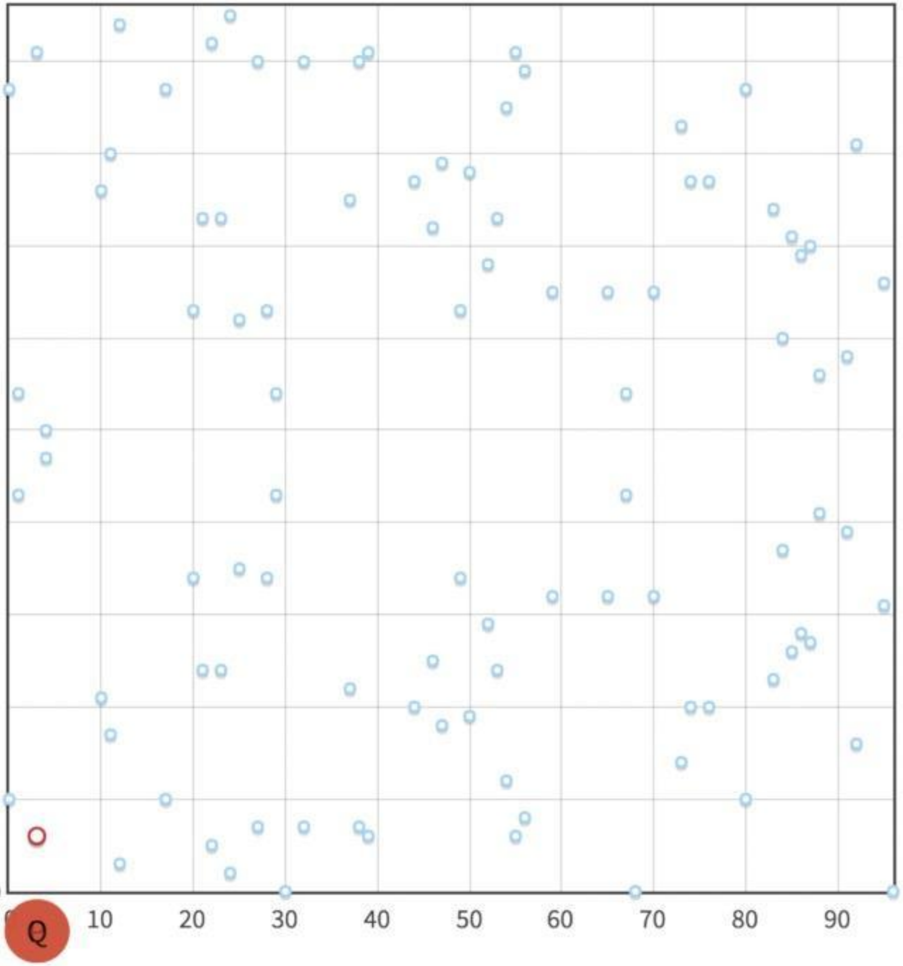
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 10

Same as n = 5, 0



Curve:	a 2	b 3
Field:	p 97	
n:	n 11	
P:	x 3	y 6
$Q = n \cdot P$:	x 3	y 6

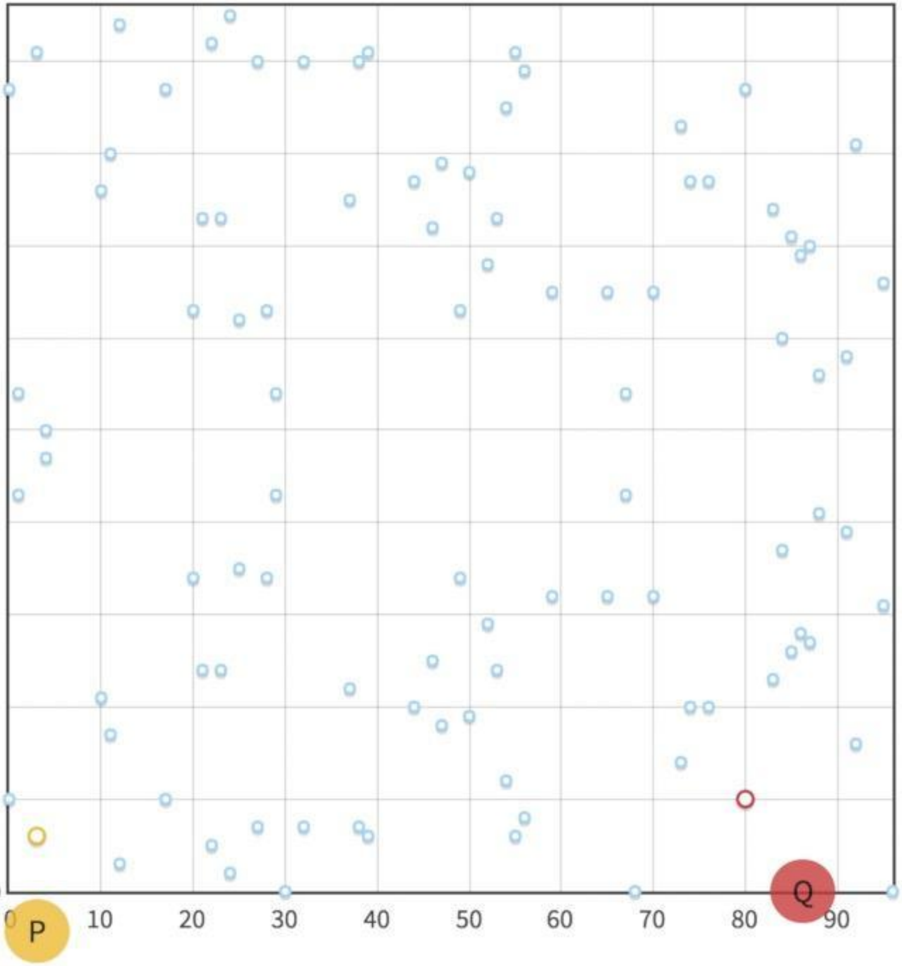
Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 11

Same as n = 1, 6



Curve: a 2 b 3

Field: p 97

n: n 12

P: x 3 y 6

$Q = n \cdot P$: x 80 y 10

Scalar multiplication over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The subgroup generated by P has 5 points.

n = 12

Same as n = 2, 7

… and the pattern continues

… and the pattern continues

(yay we have cycles!)

ECC - Groups

$y^2 = x^3 + 2x + 3 \pmod{97}$ on $P(3, 6)$ we saw a cycle

There are just 5 distinct points:

0, P, 2P, 3P, 4P

ECC - Groups

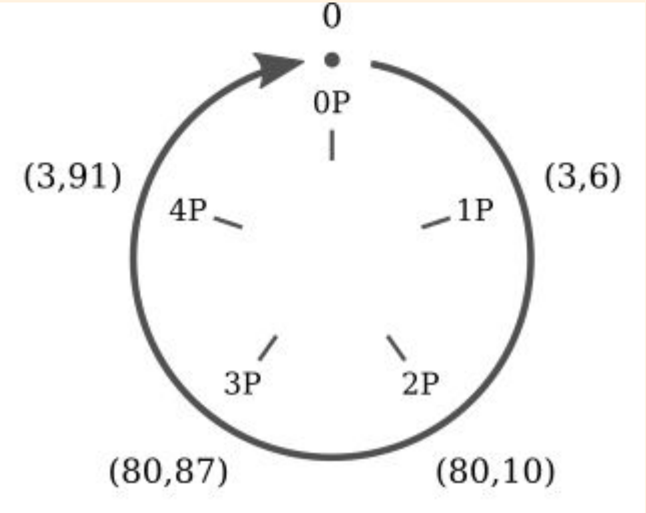
$y^2 = x^3 + 2x + 3 \pmod{97}$ on P(3, 6) we saw a cycle

There are just 5 distinct points:

0, P, 2P, 3P, 4P

These five points are closed under addition.

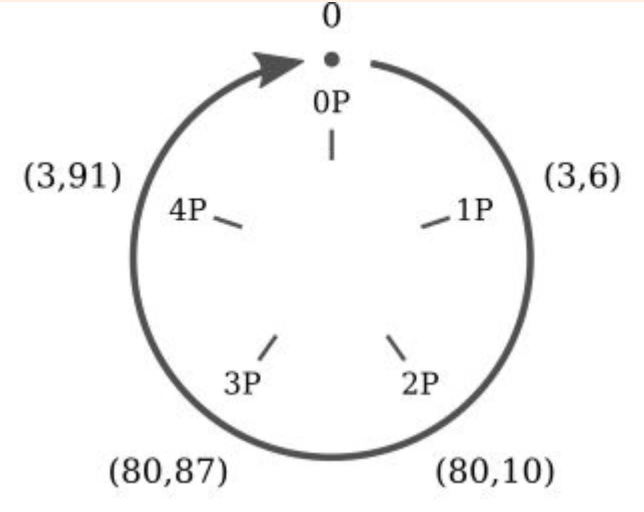
However you add 0, P, 2P, 3P or 4P, the result is always one of these five points.



ECC - Groups

The **point P(3, 6)** on $y^2 = x^3 + 2x + 3 \pmod{97}$
is therefore said to be a **generator**
or **base point** of the cyclic subgroup

and the **order** of this subgroup is
therefore 5 (the smallest n such that nP=0)



ECC - Groups

Q = nP mod p : easy to calculate
n = P/Q mod p : **discrete logarithm problem**

If you choose your curve, parameters, and generator point carefully, finding n becomes very very very hard

And this is exactly how the **ECDSA** signature scheme works

ECDSA

ECDSA signature (used in Bitcoin, Ethereum, etc):

1. **private key** random integer **d** from {1, … n-1} where n is the *order* of the subgroup

2. **public key** where **G** is the *base point* **H = dG**

ECDSA

As of today, the discrete logarithm problem for elliptic curves seems to be "harder" when compared to other similar problems used in cryptography (modular exponentiation). This means we need fewer bits for the integer in order to achieve the same level of security as with other cryptosystems.=

A Bitcoin private key is **256** bits and gives 128-bit security level

To achieve the same security with RSA (using modular exponentiation) you would need **3092** bit length key!

Zero-Knowledge Proofs!

Zero-Knowledge Proofs (1985)

1985 paper "The Knowledge Complexity of Interactive Proof-Systems"

Shafi Goldwasser, Silvio Micali, and Charles Rackoff

First coined the term **zero-knowledge proofs** for their *interactive* protocol

zk-SNARKs

[Bit+11] paper first coined the term **zk-SNARKs**

The “Pinocchio” paper [PHGR13] first made zk-SNARKs applicable for general computing

And the Groth16 (by Jens Groth, in 2016) paper made zk-SNARKs *really* efficient

Elliptic curve + pairing based

zk-SNARKs

SNARK construction flow (at least for PHGR13 / Groth16 SNARKs)

1. Computation
2. Arithmetic Circuit
3. R1CS (rank 1 constraint system)
4. QAP (quadratic arithmetic program)
5. SNARK

zk-SNARKs

SNARK construction flow (at least for PHGR13 / Groth16 SNARKs)

1. **Computation**
2. **Arithmetic Circuit**
3. **R1CS (rank 1 constraint system)**
4. **QAP (quadratic arithmetic program)**
5. SNARK

For these steps, I recommend this tutorial by Stefan Demil from Decentriq

zk-SNARKs

SNARK construction flow (at least for PHGR13 / Groth16 SNARKs)

1. Computation
2. Arithmetic Circuit
3. R1CS (rank 1 constraint system)
4. QAP (quadratic arithmetic program)
5. **SNARK**

For this step, I *highly highly* recommend this tutorial by Maksym Petkus

zk-SNARKs

TL;DR:

- For SNARKs we have a programmatic way to transform a ***statement*** into a ***language of polynomials***.
- Like with any proof system, there is a ***prover***, and a ***verifier***, and a ***challenge***.
- To make the challenge non-interactive there is a “hard coded” common reference string (CRS) or SRS (Structured Reference String) which is part of the trusted setup.
- The SRS is encrypted in order for it to be reused, which requires multiplication of encrypted values with elliptic curves, which leads to the requirement of something called ***elliptic curve pairings***

Why is Groth16 so great?

ZKPs are typically graded on:

- prover time
- proof size
- verification time

zk-SNARKs (Groth16) have:

- a (fairly) efficient **prover time**
- constant **proof size** (192 bytes)
- constant verification time

Why is Groth16 so great?

ZKPs are typically graded on:

- prover time
- proof size
- verification time

zk-SNARKs (Groth16) have:

- a (fairly) efficient **prover time**
- constant **proof size** (192 bytes)
- constant verification time

And sometimes on:

- The size of the SRS/CRS
- Cryptographic assumptions
- + other criteria

Why is Groth16 so great?

ZKPs are typically graded on:

- prover time
- proof size
- verification time

zk-SNARKs (Groth16) have:

- a (fairly) efficient **prover time**
- constant **proof size** (192 bytes)
- constant verification time

However, it has one big downside – a 🦠 trusted setup 🦠 (for which I recommend this [tutorial] by Daniel Benarroch from Qedit)

ZKPs without a trusted setup

2017 -> onwards there's been an ***explosion*** in ZKP research to try and mitigate the trusted setup requirement while still competing with Groth16 on performance

Using interactive oracle proofs (**IOP**), algebraic holographic proving systems (**AHP**), polynomial commitment schemes (**PC**), and more

ZKPs without a trusted setup

2017

- Bulletproofs
- Hyrax
- vSQL
- FRI
- Ligero
- + others

ZKPs without a trusted setup

2017

- Bulletproofs
- Hyrax
- vSQL
- FRI
- Ligero
- + others

2018

- STARKs
- Aurora
- vRAM
- GKM+18
- + others

ZKPs without a trusted setup

2017

- Bulletproofs
- Hyrax
- vSQL
- FRI
- Ligero
- + others

2018

- STARKs
- Aurora
- vRAM
- GKM+18
- + others

2019

- Libra* (not FB Libra)
- Sonic
- Supersonic
- PLONK
- RedShift
- Halo
- Marlin
- Fractal
- DARK
- Spartan
- AuroraLight
- DEEP-FRI
- FRI-based PCS
- Virgo
- Spartan
- + others

** Technically has a trusted setup, but a universal one*

ZKPs without a trusted setup

2017

- Bulletproofs
- Hyrax
- vSQL
- FRI
- Ligero
- + others

2018

- STARKs
- Aurora
- vRAM
- GKM+18
- + others

2019

- Libra* (not FB Libra)
- Sonic
- Supersonic
- PLONK
- RedShift
- Halo
- Marlin
- Fractal
- DARK
- Spartan
- AuroraLight
- DEEP-FRI
- FRI-based PCS
- Virgo
- Spartan
- + others

2020

- Halo2
- Pickles
- Quarks
- + others?

ZKPs without a trusted setup

2017

- Bulletproofs
- Hyrax
- vSQL
- FRI
- Ligero
- + others

2018

- STARKs
- Aurora
- vRAM
- GKM+18
- + others

2019

- Libra* (not FB Libra)
- Sonic
- Supersonic
- PLONK
- RedShift
- Halo
- Marlin
- Fractal
- DARK
- Spartan
- AuroraLight
- DEEP-FRI
- FRI-based PCS
- Virgo
- Spartan
- + others

2020

- Halo2
- Pickles
- Quarks
- + others?

Which ZKPs should I pay attention to today?

Groth16 🌟

- Overall (proof size, prover time, verification time) it is currently the gold standard of ZKPs
- R1CS is great – leads to high level DSLs (domain specific languages) or advanced toolsets:
  - ZoKrates, libsnark, bellman, circom, zinc (I think?), aleo (uses ZEXE which way more involved)
  - Very mature toolset that you can use today (!!) to write ZKP apps & smart contracts

Which ZKPs should I pay attention to today?

PLONK – no trusted setup; universal & updatable

- Prover time overall slower than Groth16 – *but not by much!*
  - And even outperforms for some operations, like MiMC hashes or, I believe, Pedersen commitments on bn128 curve
- Proof size bigger – *but not by much!*
- Verification time slower – *but not by much!*
- It unfortunately does not use R1CS, and therefore all the great tools for building circuits are not applicable here, however Aztec is working on Noir, Coda/Mina is working on Pickles, and Mir is working on Plonky which are *almost* production ready/usable

- **Halo2** is super exciting (though still a WIP) and uses PLONK
- *To learn how PLONK works, I recommend this tutorial by Vitalik*

.. And that's it! 😅

Questions?