Lesson 4

Maths / Cryptography Part 2

Quote from Remco Bloemen remco@0x.org

> *Disclaimer: contains maths*
>
> *If you don't understand something*
>
> - *Not your fault, this stuff is hard*
> - *Nobody understands it fully*
>
> *If you don't understand anything*
>
> - *My fault, anything can be explained at some level*
>
> *If you do understand everything*
> ** Collect your Turing Award & Fields Medal*

Maths background

Terminology

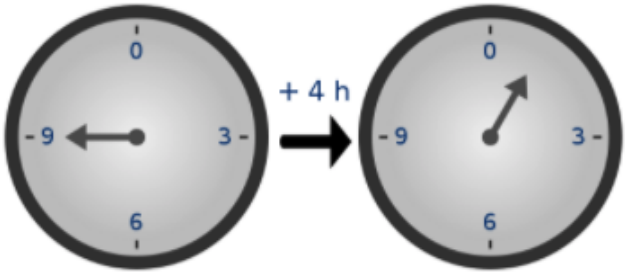
Numbers

The set of Integers is denoted by $\mathbb{Z}$ e.g. $\{\cdots,-4,-3,-2,-1,0,1,2,3,4,\cdots\}$
The set of Rational Numbers is denoted by $\mathbb{Q}$ e.g. $\{\ldots 1, \frac{3}{2}, 2, \frac{22}{7} \ldots\}$
The set of Real Numbers is denoted by $\mathbb{R}$ e.g. $\{2, -4, 613, \pi, \sqrt{2}, \ldots\}$

Fields are denoted by $\mathbb{F}$, if they are a finite field or $\mathbb{K}$ for a field of real or complex numbers we also use $\mathbb{Z}_p^*$ to represent a finite field of integers mod prime p with multiplicative inverses.

We use finite fields for cryptography, because elements have "short", exact representations.

Modular Arithmetic



Because of how the numbers "wrap around", modular arithmetic is sometimes called "clock math"

When we write n mod k we mean simply the remainder when n is divided by k. Thus
25 mod 3 = 1,
15 mod 4 = 3,
–13 mod 5 = -3 , = 2 mod 5.
It is an important fact that modular arithmetic respects sums and products.
That is,
a+b mod n = a mod n+ b mod n
and
a·b mod n=(a mod n)·(b mod n)

Fermat's little theorem

Let a be an integer and p be a prime, then:

$a^p \equiv a(mod\ p).$

which can be stated as
$a^p - 1 \equiv 1(mod\ p)$
$a \cdot a^p - 2 \equiv 1(mod\ p)$

Thus, we immediately have a way for inverting an integer a modulo a prime:

$a^-1 \equiv a^p - 2(mod\ p)$

See this introduction

QUADRATIC RESIDUES

https://en.wikipedia.org/wiki/Quadratic_residue

An integer q is a quadratic residue modulo n if there is an integer x such that
$x^2 \equiv q$ (mod n)

Group Theory

Simply put a group is a set of elements {a,b,c,...} plus a binary operation, here we represent this as •

To be considered a group this combination needs to have certain properties

1. Closure
   For all a, b in G, the result of the operation, a • b, is also in G
2. Associativity
   For all a, b and c in G, (a • b) • c = a • (b • c)
3. Identity element
   There exists an element e in G such that, for every element a in G, the equation e • a = a • e = a holds. Such an element is unique and thus one speaks of the identity element.
4. Inverse element
   For each a in G, there exists an element b in G, commonly denoted a^{-1} (or –a, if the operation is denoted "+"), such that a • b = b • a = e, where e is the identity element.

An example of a group is the set of nonzero integers (between 1 and p – 1) modulo some prime number p, which we write $\mathbb{Z}_p^*$

SUB GROUPS

If a subset of the elements in a group also satisfies the group properties, then that is a subgroup of the original group.

CYCLIC GROUPS AND GENERATORS

A finite group can be cyclic. That means it has a generator element. If you start at any point and then apply the group operation with the generator as argument a certain number of times, you go around the whole group and end in the same place,

Fields

A field is a set of say Integers together with two operations called addition and multiplication.
One example of a field is the Real Numbers under addition and multiplication, another is a set of Integers mod a prime number with addition and multiplication.
The field operations are required to satisfy the following field axioms. In these axioms, a, b and c are arbitrary elements of the field $\mathbb{F}$.

1. Associativity of addition and multiplication: a + (b + c) = (a + b) + c and a · (b · c) = (a · b) · c.
2. Commutativity of addition and multiplication: a + b = b + a and a · b = b · a.
3. Additive and multiplicative identity: there exist two different elements 0 and 1 in $\mathbb{F}$ such that a + 0 = a and a · 1 = a.
4. Additive inverses: for every a in F, there exists an element in F, denoted –a, called the additive inverse of a, such that a + (–a) = 0.
5. Multiplicative inverses: for every a ≠ 0 in F, there exists an element in F, denoted by a^{-1} , called the multiplicative inverse of a, such that a ·a^{-1} = 1.
6. Distributivity of multiplication over addition: a · (b + c) = (a · b) + (a · c).

FINITE FIELDS AND GENERATORS

A finite field is a field with a finitie set of elements, such as the set of integers mod p where p is a prime.

To try out operations on finite fields, see https://asecuritysite.com/encryption/finite

The **order** of the field is the number of elements in the field's set.
For a finite field the order must be either

- prime (a prime field)
  or
- the power of a prime (an extension field)

An element can be represented as an integer greater or equal than 0 and less than the field's order: {0, 1, ..., p-1} in a simple field.

Every finite field has a generator. A generator is capable of generating all of the elements in the set by exponentiating the generator .
So for generator g we can take g^0, g^1,g^2 and eventually this will give us all elements in the group

For example. taking the set of integers and prime p = 5, we get the group $\mathbb{Z}_5^*$ = {1,2,3,4}.
In the group Z$\mathbb{Z}_5^*$, operations are carried out modulo 5; hence, we don't have 3 × 4 = 12 but instead have 3 × 4 = 2, because 12 mod 5 = 2.

$\mathbb{Z}_5^*$ is cyclic and has two generators, 2 and 3, because 21 = 2, 22 = 4, 23 = 3, 24 = 1, and 31 = 3, 32 = 4, 33 = 2, 34 = 1.

In a finite field of order q, the polynomial $X^q - X$ has all q elements of the finite field as roots.

Extension Fields

A common example is the field of complex numbers, where the field of real numbers is "extended" with the additional element $-1 = i$.
Basically, extension fields work by taking an existing field, then "inventing" a new element and defining the relationship between that element and existing elements
(in this case, $i^2 + 1 = 0$),
making sure that this equation does not hold true for any number that is in the original field, and looking at the set of all linear combinations of elements of the original field and the new element that you have just created.

GROUP HOMOMORPHISMS

A homomorphism is a map between two algebraic structures of the same type, that preserves the operations of the structures.

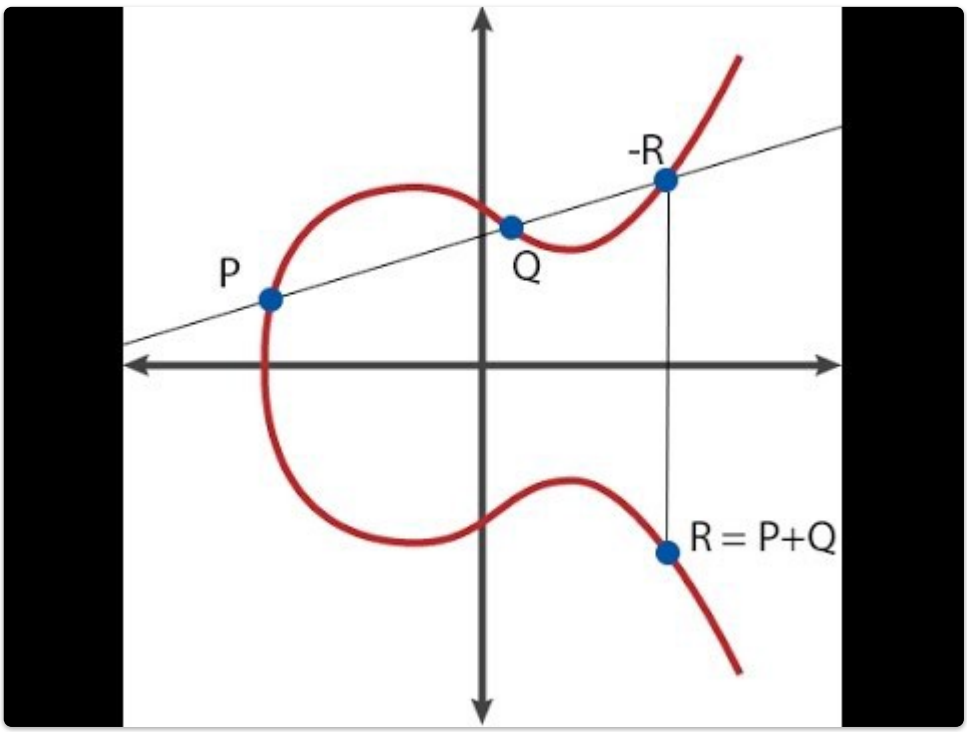
This means a map $f : A \to B$ between two groups A, B equipped with the same structure such that,

if $\cdot$ is an operation of the structure (here a binary operation), then
$f(x \cdot y) = f(x) \cdot f(y)$

Elliptic Curves

The defining equation for an elliptic curve is of the form. $y^2 = x^3 + ax + b$
For certain equations they will satisfy the group axioms

- every two points can be added to give a third point (closure);
- it does not matter in what order the two points are added (commutatitivity);
- if you have more than two points to add, it does not matter which ones you add first either (associativity);
- there is an identity element.



Pairings

Bilinear pairings are functions that take two arguments and return one output, usually denoted by

```
    e(G1, G2) --> GT.
```

with the following properties:

- Order:All three groups must have order equal to a prime r.
- Efficiency: the pairing function must be efficiently computable.

- Bilinearity: For any elements P1, P2 of G1 and any elements Q1, Q2 of G2, the following holds true:

  ```
  e(P1 + P2, Q1) = e(P1, Q1) e(P2, Q1)
  e(P1, Q1 + Q2) = e(P1, Q1) e(P1, Q2)
  ```

  This implies the following form, which is more often used (along with some other variants):

  ```
  e(aP, bQ) = e(P,Q)ab = e(bP, aQ)
  ```

- Non-degeneracy: the pairing of the generators of the first two groups is not the identity of the third group. If this were the case, every pairing would result in the same (the identity) element of GT:

  $e(E1, E2) \neq 1T$

The three groups must be cyclic and have the same prime order

Types of pairings

- Type 1: `G1` and `G2` are the same group, or can easily be converted into each other;
- Type 2: `G1` and `G2` are different groups; it is possible to efficiently convert an element of `G2` into an element of `G1` but not the other way around;
- Type 3: `G1` and `G2` are different groups; there is no known efficient way to compute a general element of any of these groups into an element of the other.

ELLIPTIC-CURVE PAIRINGS

At the heart of zkSNARKS we have pairings between elliptic curves

- We start with a field of elements.
- From this we can construct an elliptic curve
- An operation (such as addition) is defined over this curve, thhis gives us a group.
- Two other curves and groups are derived from the first one
- A pairing is a particular bilinear function that takes an element from each of the first two groups and produces an element of the third.

Polynomials

A polynomial is an expression that can be built from constants and variables by means of addition, multiplication and exponentiation to a non-negative integer power.

e.g. $3x^2 + 4x + 3$

Quote from Vitalik Buterin
"There are many things that are fascinating about polynomials. But here we are going to

zoom in on a particular one: **polynomials are a single mathematical object that can contain an unbounded amount of information** (think of them as a list of integers and this is obvious)."
Furthermore, **a single equation between polynomials can represent an unbounded number of equations between numbers**.
For example, consider the equation A(x)+B(x)=C(x). If this equation is true, then it's also true that:

- A(0)+B(0)=C(0)
- A(1)+B(1)=C(1)
- A(2)+B(2)=C(2)
- A(3)+B(3)=C(3)

Adding, multiplying and dividing polynomials

We can add, multiply and divide polynomials, for examples see https://en.wikipedia.org/wiki/Polynomial_arithmetic

ROOTS

For a polynomial P of a single variable x in a field K and with coefficients in that field, the root r of P is an element of K such that $P(r) = 0$

B is said to divide another polynomial A when the latter can be written as

$A = BC$

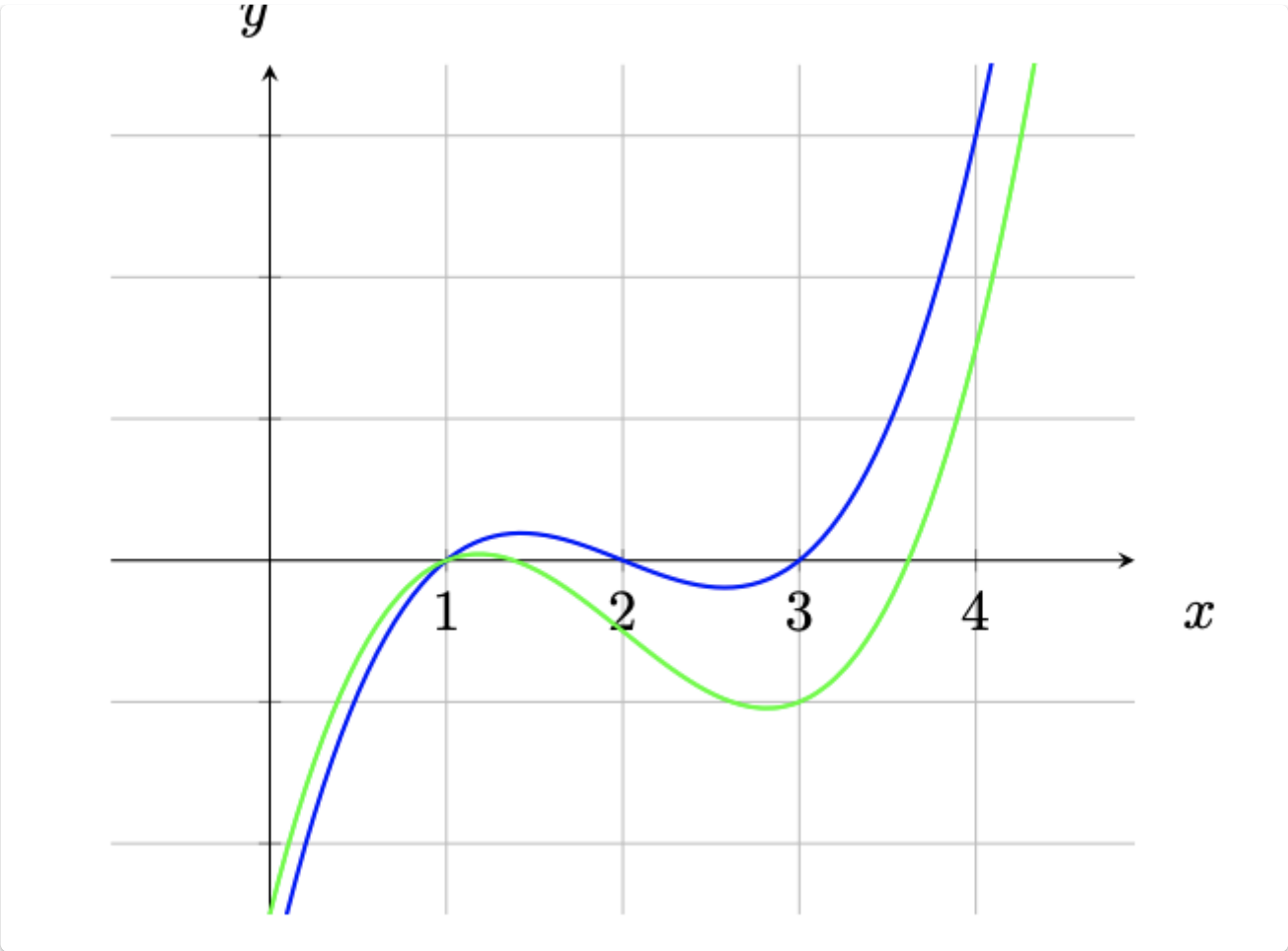
with C also a polynomial,the fact that B divides A is denoted $B|A$

If one root r of a polynomial $P(x)$ of degree n is known then polynomial long division can be used to factor $P(x)$ into the form
$(x - r)(Q(x))$
where
$Q(x)$ is a polynomial of degree $n - 1$.
$Q(x)$ is simply the quotient obtained from the division process; since r is known to be a root of $P(x)$, it is known that the remainder must be zero.

Schwartz-Zippel Lemma

“different polynomials are different at most points”.

Polynomials have an advantageous property, namely, if we have two non-equal polynomials of degree at most d, they can intersect at no more than d points.



Lagrange Interpolation

If you have a set of points then doing a Lagrange interpolation on those points gives you a polynomial that passes through all of those points.

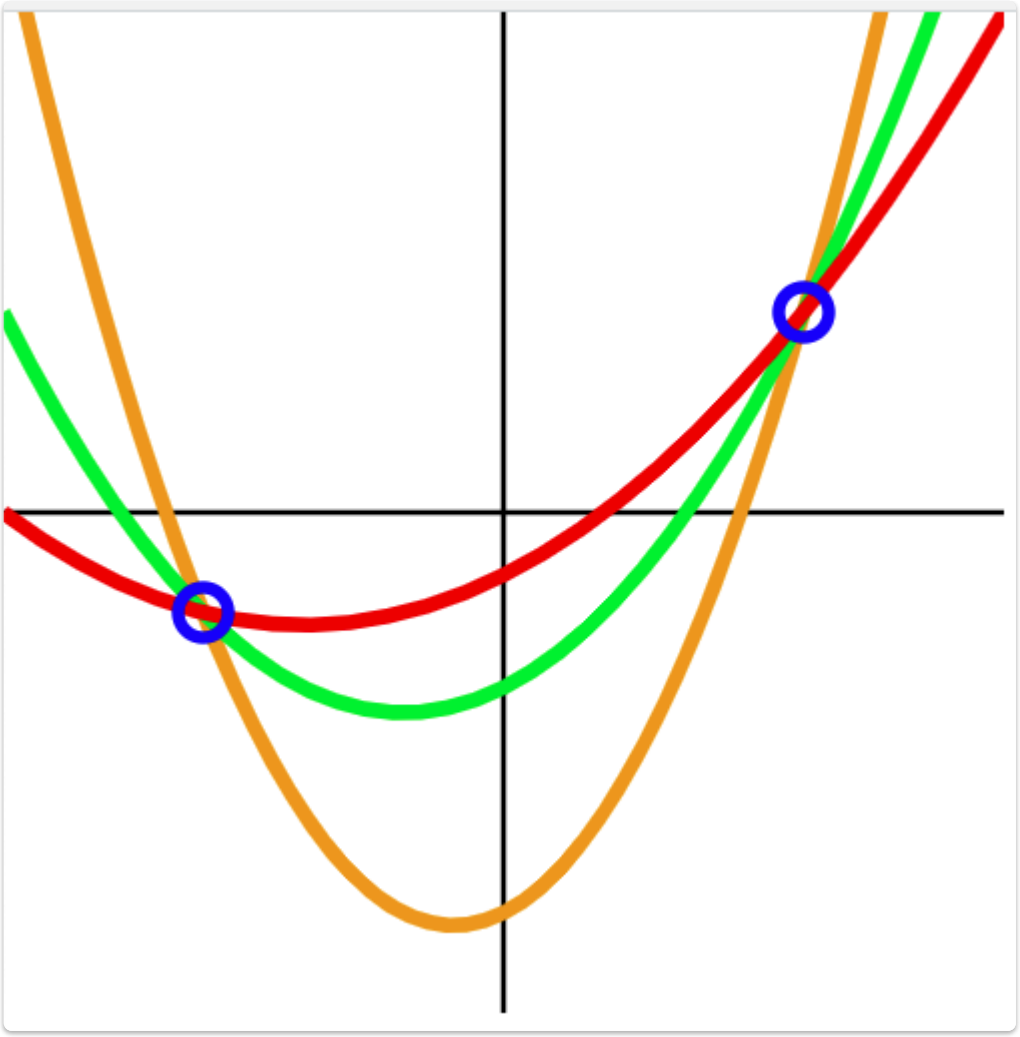
If you have two points on a plane, you can define a single straight line that passes through both, for 3 points, a single 2nd-degree curve (e.g. $5x^2 + 2x + 1$) will go through them etc. For n points, you can create a n-1 degree polynomial that will go through all of the points.

(We can use this in all sorts of interesting schemes as well as zkps)

HOW DO WE USE THIS IN ZKPS ?

If a prover claims to know some polynomial (no matter how large its degree is) that the verifier also knows, they can follow a simple protocol to verify the statement:
• Verifier chooses a random value for x and evaluates his polynomial locally
• Verifier gives x to the prover and asks to evaluate the polynomial in question
• Prover evaluates her polynomial at x and gives the result to the verifier
• Verifier checks if the local result is equal to the prover's result, and if so then the statement is proven with a high confidence

Why is low degree important

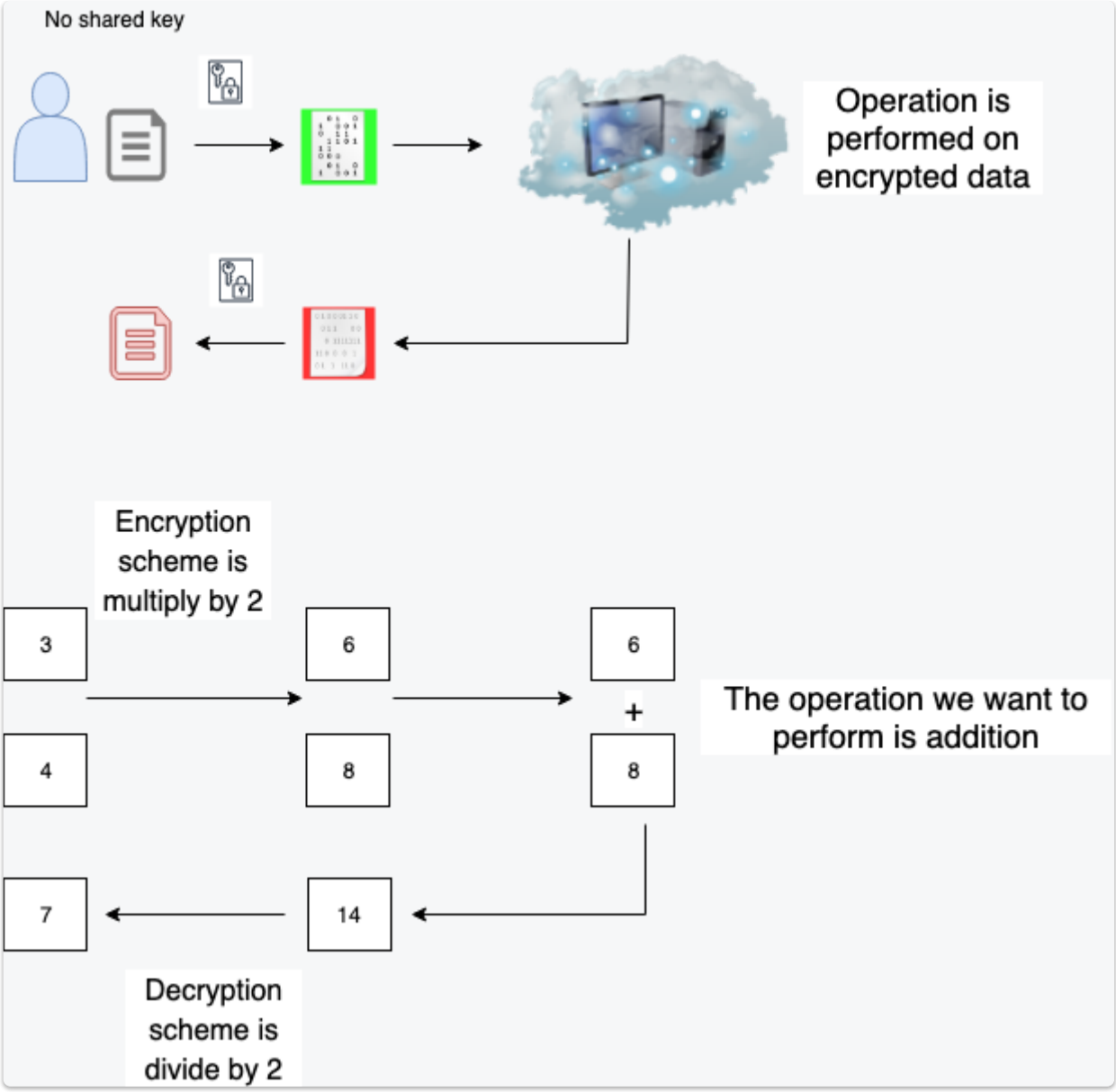


In general, there is a rule that if a polynomial P is zero across some set S={x1,x2...xn} then it can be expressed as
P(x)=Z(x)∗H(x), where
Z(x)= (x−x1)∗(x−x2)∗...∗(x−xn) and
H(x) is also a polynomial.
In other words, **any polynomial that equals zero across some set is a (polynomial) multiple of the simplest (lowest-degree) polynomial that equals zero across that same set**.

(Fully) Homomorphic Encryption

Fully Homomorphic Encryption , the 'holy grail' of cryptography, is a form of encryption that allows arbitrary computations on encrypted data.

Homomorphic encryption is a form of encryption with an additional evaluation capability for computing over encrypted data without access to the secret key. The result of such a computation remains encrypted. Homomorphic encryption can be viewed as an extension of either symmetric-key or public-key cryptography. Homomorphic refers to homomorphism in algebra: the encryption and decryption functions can be thought as homomorphisms between plaintext and ciphertext spaces.



Alice, the data owner, encrypts data with her key and sends it to an outsourced machine for storage and processing.
The outsourced machine performs arbitrary computations on the encrypted data without learning anything about it.
Alice decrypts the results of those computations using her original key, retaining full confidentiality, ownership, and control.

Example :

Medical Data using FHE

BITCOIN SPLIT-KEY VANITY MINING

Bitcoin addresses are hashes of public keys from ECDSA key pairs. A vanity address is an address generated from parameters such that the resultant hash contains a human-readable string (e.g., 1BoatSLRHtKNngkdXEeobR76b53LETtpyT).
Given that ECDSA key pairs have homomorphic properties for addition and multiplication, one can outsource the generation of a vanity address without having the generator know the full private key for this address.

For example,
Alice generates a private key (a) and public key (A) pair, and publicly posts A.
Bob generates a key pair (b, B) such that hash(A + B) results in a desired vanity address.
He sells b and B to Alice.
A, B, and b are publicly known, so one can verify that the address = hash(A + B) as desired.
Alice computes the combined private key (a + b) and uses it as the private key for the public key (A + B).
Similarly, multiplication could be used instead of addition.

Homomorphic Hiding

(Taken from the ZCash explanation)

If $E(x)$ is a function with the following properties

- Given $E(x)$ it is hard to find x
- Different inputs lead to different outputs so if $x \neq y$ $E(x) \neq E(y)$
- We can compute $E(x+y)$ given $E(x)$ and $E(y)$

The group $\mathbb{Z}_p^*$ with operations addition and multiplication allows this.

Here's a toy example of why Homomorphic Hiding is useful for Zero-Knowledge proofs:
Suppose Alice wants to prove to Bob she knows numbers x,y such that $x+y=7$

1. Alice sends $E(x)$ and $E(y)$ to Bob.
2. Bob computes $E(x+y)$ from these values (which he is able to do since E is an HH).
3. Bob also computes $E(7)$, and now checks whether $E(x+y)=E(7)$. He accepts Alice's proof only if equality holds.

As different inputs are mapped by E to different hidings, Bob indeed accepts the proof only if Alice sent hidings of x, y such that $x+y=7$. On the other hand, Bob does not learn x and y as he just has access to their hidings 2(https://electriccoin.co/blog/snark-explain/#id5).

Complexity Theory

Complexity theory looks at the time or space requirements to solve a problem, particularly in terms of the size of the input.
We can classify problems according to the time required to find a solution, for some problems there may exist an algorithm to find a solution in a reasonable time, whereas for other problems we may not know of such an algorithm, and may have to 'brute force' a solution, trying out all potential solutions until one is found.

For example the travelling salesman problem tries to find the shortest route for a salesman required to travel between a number of cities, visiting every city exactly once. For a small

number of cities, say 3, we can quickly try all alternatives to find the shortest route, however as the number of cities grows, this quickly becomes unfeasible.

Based on the size of the input n , we classify problems according to how the time required to find a solution grows with n.
If the time taken in the worst case grows as a polynomial of n, that is roughly proportional to n^k for some value k, we put these problems in class P for polynomial. These problems are seen as tractable.

We are also interested in knowing how long it takes to verify a potential solution once it has been found.

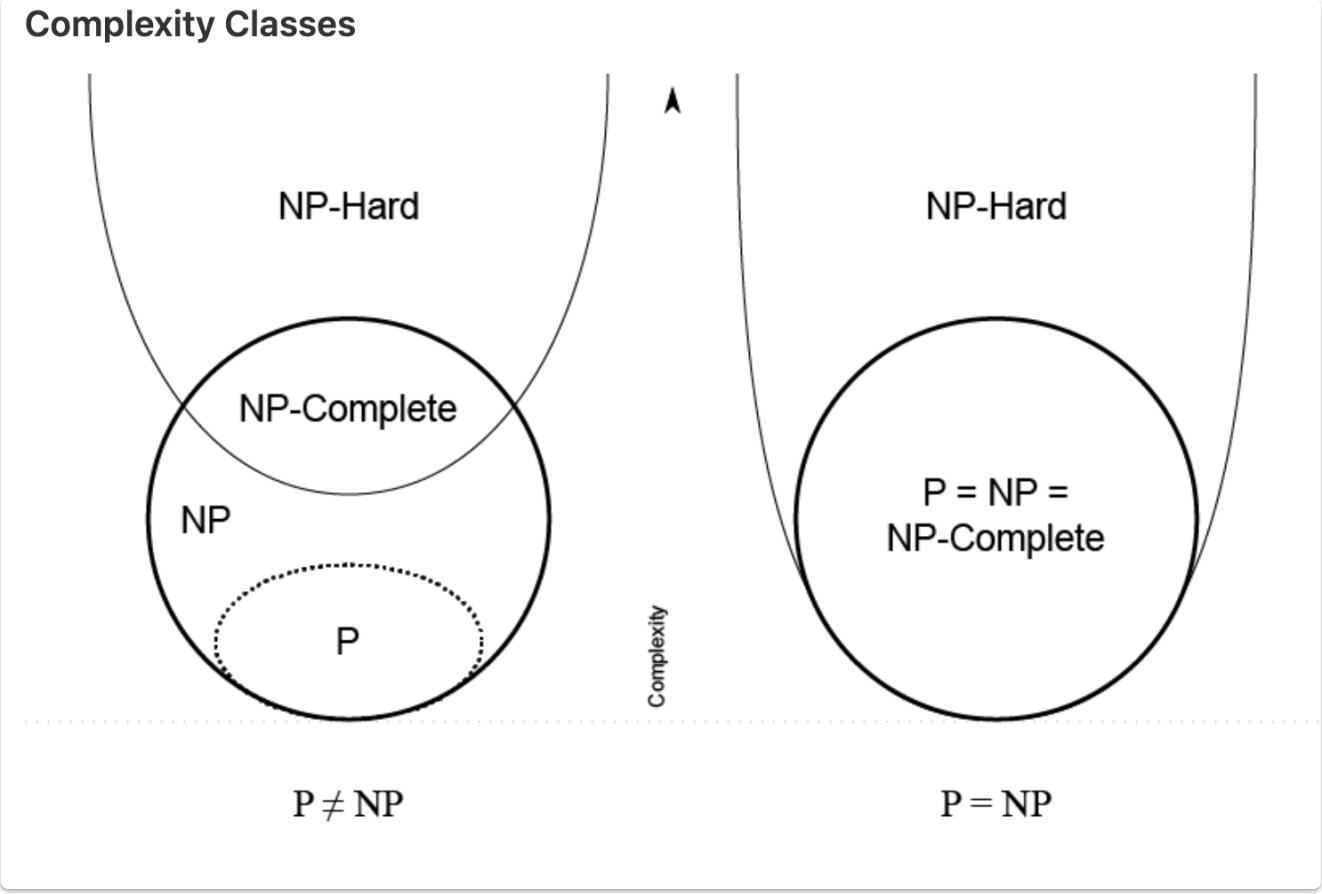
A computational problem can be viewed as an infinite collection of instances together with a solution for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself.

https://en.wikipedia.org/wiki/Computational_complexity_theory

Decision Problem: A problem with a yes or no answer
Complexity classes :
https://upload.wikimedia.org/wikipedia/commons/thumb/7/7e/Comparison_computational_complexity.svg/220px-Comparison_computational_complexity.svg.png

From "Everything provable is provable in zero knowledge"
https://dl.acm.org/doi/pdf/10.5555/88314.88333

"Assuming the existence of a secure probabilistic encryption scheme, we show that every language that admits an interactive proof admits a (computational) zero-knowledge interactive proof. This result extends the result of Goldreich, MiCali and Wigderson, that, under the same assumption, all of NP admits zero-knowledge interactive proofs."



P

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

NP

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is “yes” have proofs that can be verified in polynomial time, even though the solution may be hard to find.
This means that if someone gives us an instance of the problem and a witness to the answer being yes, we can check that it is correct in polynomial time, that is you can run some polynomial-time algorithm that will verify whether you’ve found an actual solution.
For example, the problem of recovering a secret key with a known plaintext is in NP, because
you can check that a candidate key is the correct key by verifying that encrypting the plaintext with that key and showing that it equals the supplied cypher text.
The process of finding a potential key (the solution) can’t be done in polynomial time, but checking whether the key is correct is done using a polynomial-time algorithm.

NP-COMPLETE

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.
Intuitively this means that we can solve Y quickly if we know how to solve X quickly.

Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances
x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes

NP-HARD

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.
The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.
But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

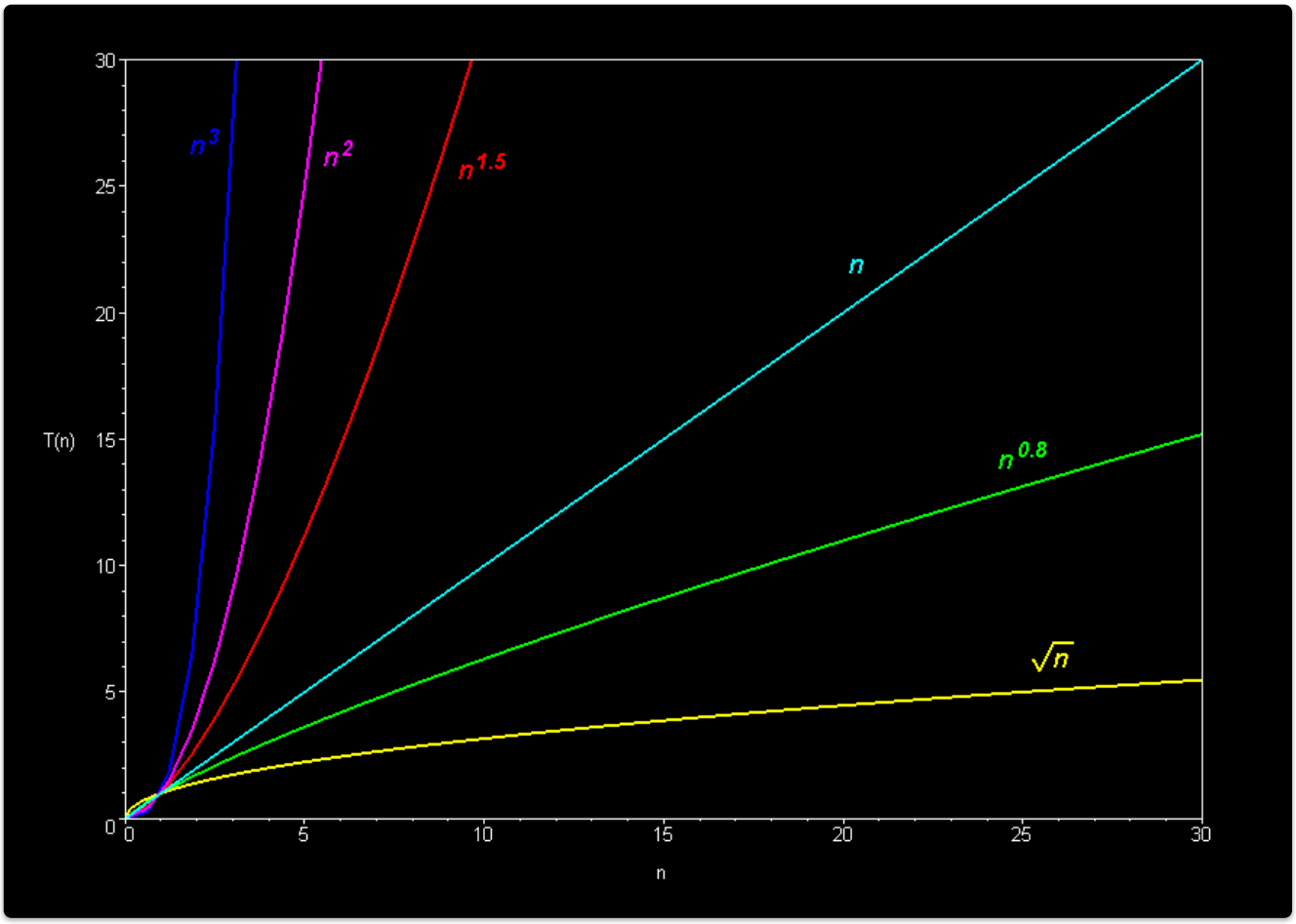
Even the task of winning in certain video games can sometimes be proven to be NP-complete (for famous games including Tetris, Super Mario Bros., Pokémon, and Candy Crush Saga). For example, the article "Classic Nintendo Games Are (Computationally) Hard" (https://arxiv.org/abs/1203.1895) considers "the decision problem of reachability" to determine the possibility of reaching the goal point from a particular starting point.
Some of these video game problems are actually even harder than NP-complete and are called NP-hard. We say that a problem is NP-hard when it's at least as hard as NP-complete problems. More formally, a problem is NP-hard if what it takes to solve it can be proven to also solve NP-complete problems.

So being NP-complete doesn't mean that all instances of a given problem are hard, but that as the problem size grows, many of them are.

Big O notation

In plain words, Big O notation describes the **complexity** of your code using algebraic terms.
It describes the time or space required to solve a problem in the worse case in terms of the size of the input.

For example if we say for input size n
$O(n^2)$
we are saying that as n increases, the time taken to solve the problem goes up to proportional to the square of n.



We use this notation when comparing ZKP systems

	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	Strong 😒	Collision resistant hashes 😄	Discrete log 😒