Lesson 4

Week 1

Lesson 1 - Introduction to Blockchain and Layer 1

Lesson 2 - Why Scalability

Lesson 3 - Introduction to Layer 2

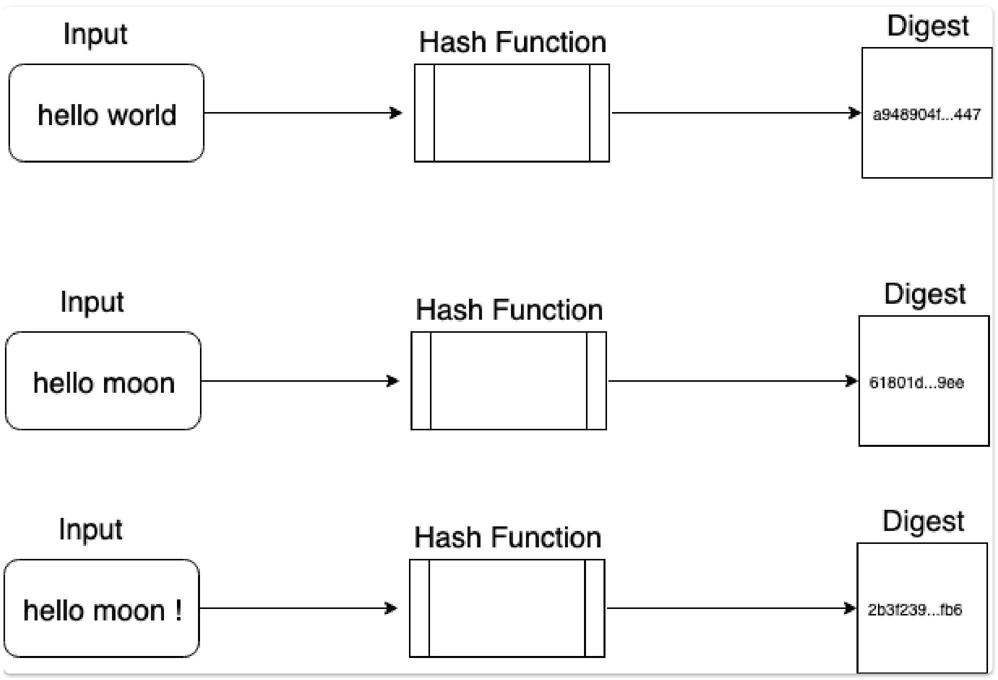
Lesson 4 - Maths and Cryptography

Today's Topics

- Cryptography review
- Mathematical terminology
- Modular arithmetic
- Group and Field theory
- Introduction to polynomials
- Complexity theory
- Introduction to Zero Knowledge Proofs

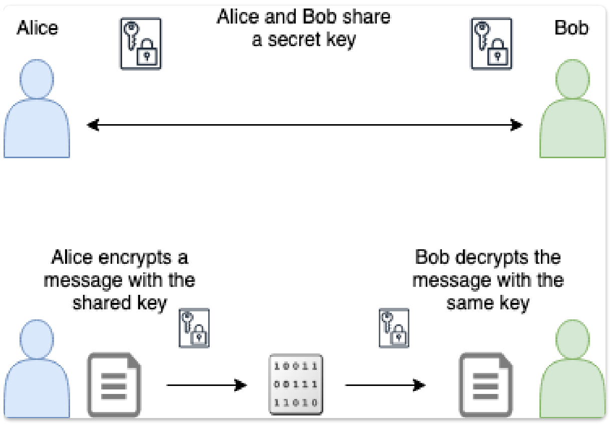
Cryptographic Background

Hash Functions



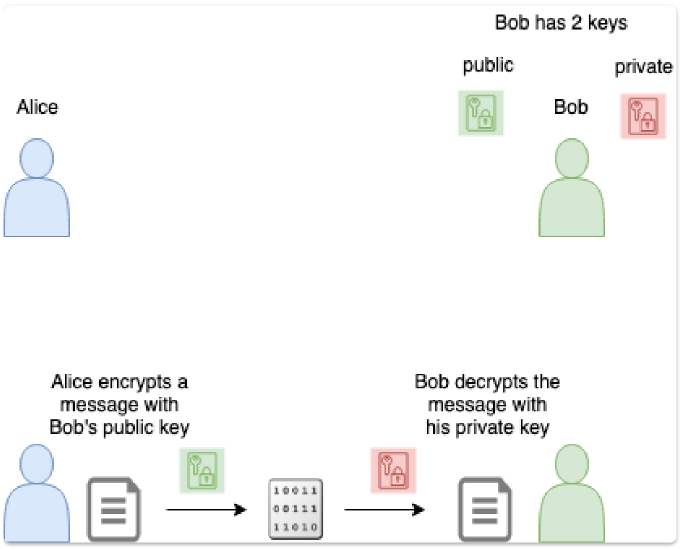
Encryption

Symmetric Encryption

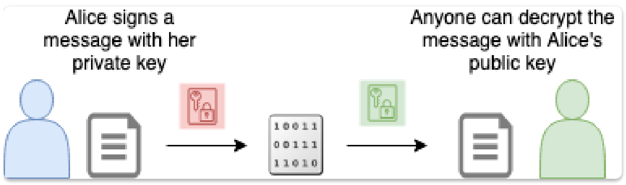

Alice
Alice and Bob share a secret key
Bob
Alice encrypts a message with the shared key
Bob decrypts the message with the same key
10011
00111
11010

Asymmetric Encryption

Sending a secret message


Bob has 2 keys
public
private
Alice
Bob
Alice encrypts a message with Bob's public key
Bob decrypts the message with his private key
10011
00111
11010

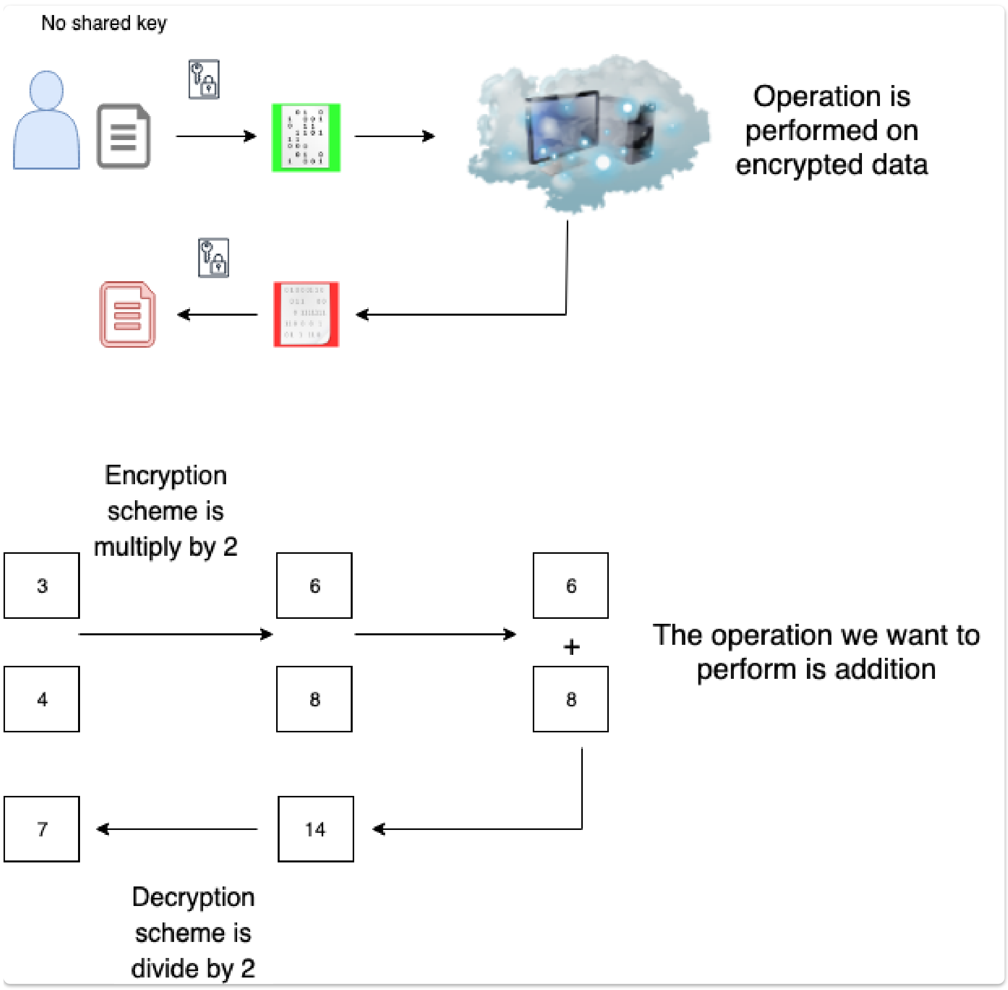
Proving ownership (knowledge of) of a private key


Alice signs a message with her private key
10011 00111 11010
Anyone can decrypt the message with Alice's public key

(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.


No shared key
Operation is performed on encrypted data
Encryption scheme is multiply by 2
3
4
6
8
6
+
8
The operation we want to perform is addition
7
14
Decryption scheme is divide by 2

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 :
>
> Private computation : fhEVM from Zama.ai
>
> Medical Data using FHE
>
> **Verifiable Random Functions**
>
> From Algorand VRFs :
>
> A Verifiable Random Function (VRF) is a cryptographic primitive that maps inputs to verifiable pseudorandom outputs. VRFs were Introduced by Micali, Rabin, and Vadhan in '99.
>
> Such functions are ideal to find block producers in a blockchain in a trustless verifiable way.
>
> **Verifiable Delay Functions**
>
> A way to show that some time, or some amount of computation has happened, similar to the technique used for Solana's Proof of History

Numbers and terminology

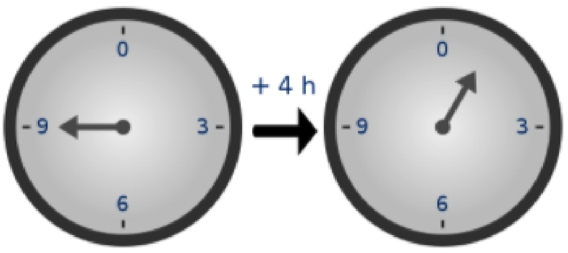
The set of Integers is denoted by $\mathbb{Z}$ e.g. {⋯,−4,−3,−2,−1,0,1,2,3,4,⋯}
The set of Rational Numbers is denoted by $\mathbb{Q}$ e.g. $\{\ldots 1, \frac{3}{2}, 2, \frac{22}{7} \ldots\}$
The set of Real Numbers is denoted by $\mathbb{R}$ e.g. $\{2, -4, 613, \pi, \sqrt{2}, \ldots\}$

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 and useful properties.

Modular Arithmetic

See this introduction



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

The remainder should be positive.

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.

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 finite 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 $\mathbb{Z}_5^*$ = {0,1,2,3,4}.
In the group $\mathbb{Z}_5^*$, 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
$2^1 = 2, 2^2 = 4, 2^3 = 3, 2^4 = 1$, and $3^1 = 3, 3^2 = 4, 3^3 = 2, 3^4 = 1$

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

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

Equivalence classes

Since

6 mod 7 = 6

13 mod 7 = 6

20 mod 7 = 6

...

we can say that 6, 13, 20 ... form an equivalence class

more formally

modular arithmetic partitions the integers into N equivalence classes, each of the form $i + kN \mid k \in \mathbb{Z}$ for some i between 0 and $N - 1$.

Thus if we are trying to solve the equation

x mod 7 = 6

x could be 6, 13, 20 ...

This gives us the basis for a one way function.

Fermat's little theorem

This is useful for finding a multiplicative inverse

$a^{-1} \equiv a^{p-2} (mod p)$
Let p = 7 and a = 2. We can compute the inverse of a as:
$a^{p-2} = 2^5 = 32 \equiv 4$ mod 7.

This is easy to verify: 2 x 4 ≡ 1 mod 7.

Polynomial Introduction

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

$A = BC$

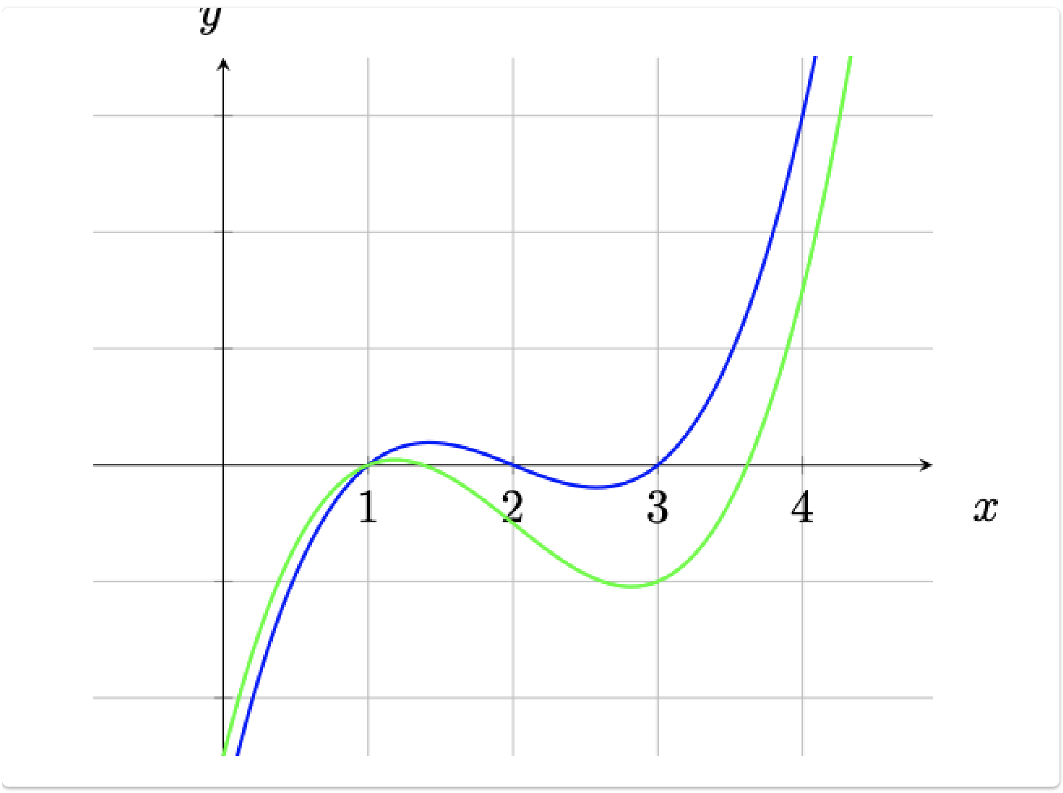
with C also a polynomial,the fact that B divides A is denoted $B|A$

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.



if f and g are polynomials and are equal, then
$f(x) = g(x)$ for all x

if f and g are polynomials and are NOT equal, then
$f(x) \neq g(x)$ for all pretty much any x

What does it mean to say 2 polynomials are equal ?

1. They evaluate to the same value or all points

2. They have the same coefficients

If we are working with real numbers, these 2 points would go together, however that is not the case when we are working with finite fields.

For example all elements of a field of size q satisfy the identity
$x^q = x$

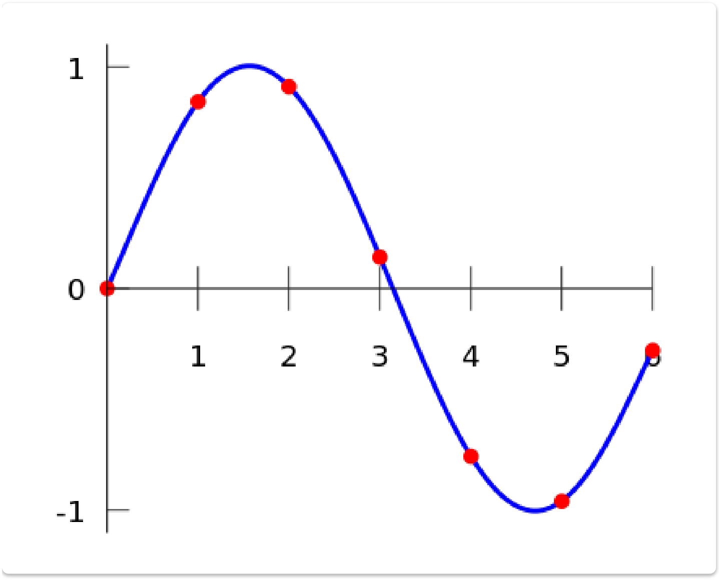
The polynomials X^q and X take the same values at all points, but do not have the same coefficients.

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)



Representations

We effectively have 2 ways to represent polynomials

1. Coefficient form
$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \ldots$

2. Point value form
   $(x_1, y_1), (x_2, y_2), \ldots$

> We can switch between the two forms by evaluation or interpolation.

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 that works.

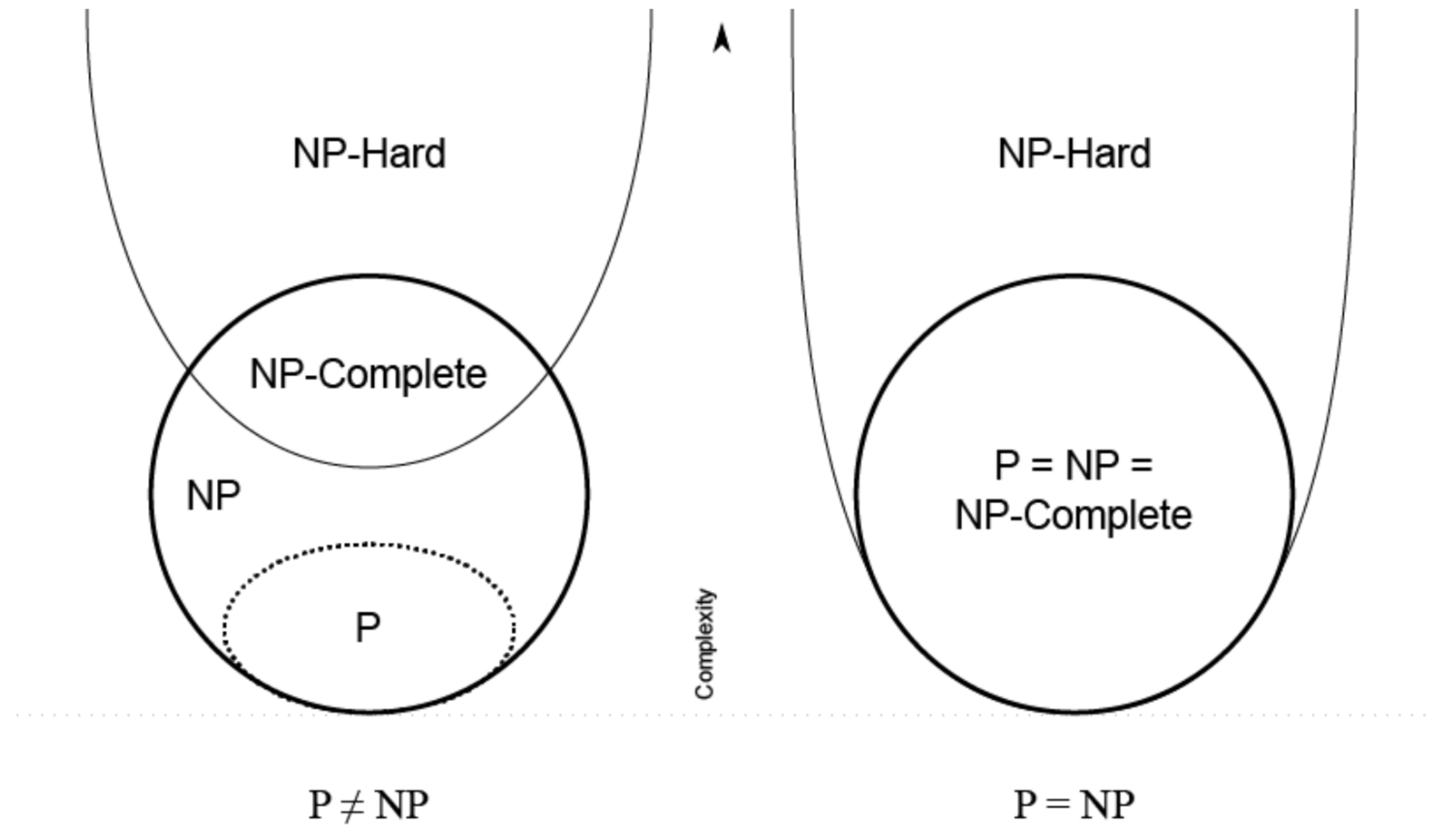
For example the travelling salesman problem tries to find the shortest route for a salesman required to travel between a number of cities, visiting every city exactly once. For a small number of cities, say 3, we can quickly try all alternatives to find the shortest route, however as the number of cities 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.

Decision Problem: A problem with a yes or no answer

Complexity Classes



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.

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."

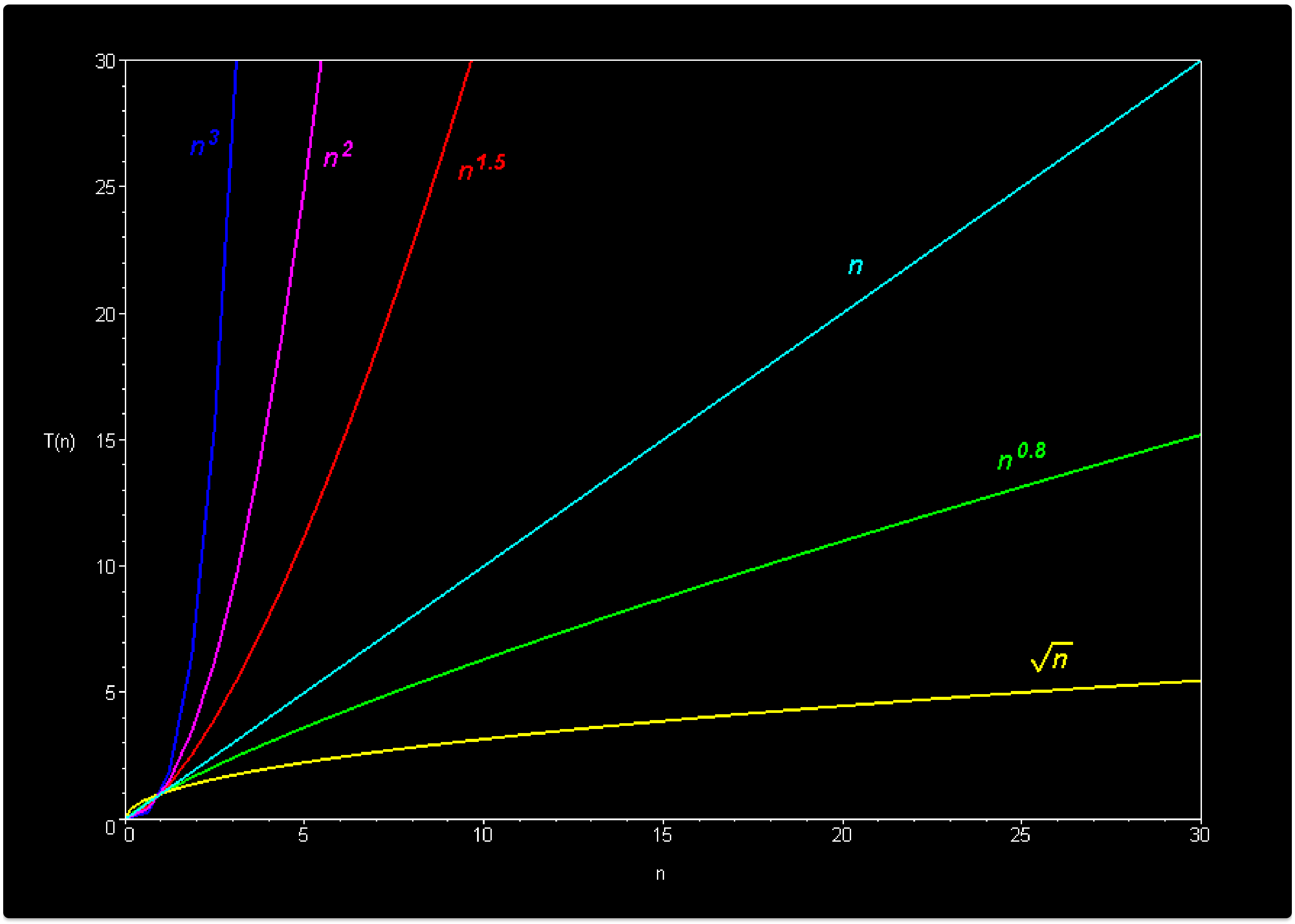
IP

A class that lies at the heart of zkps is the Interactive Proof class.
In complexity theory they are related to the other complexity classes, and we shall see how the idea of interactive proofs, as opposed to traditional static proofs is widely used in zk proving systems.
Interactive proof videos from Alessandro Chiesa

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 😏

Short Zero Knowledge Proof Introduction

What is a zero knowledge proof ?

A loose definition :
It is a proof that there exists or that we know something, plus a zero knowledge aspect, that is the person verifying the proof only gains one piece of information - that the proof is valid or invalid.

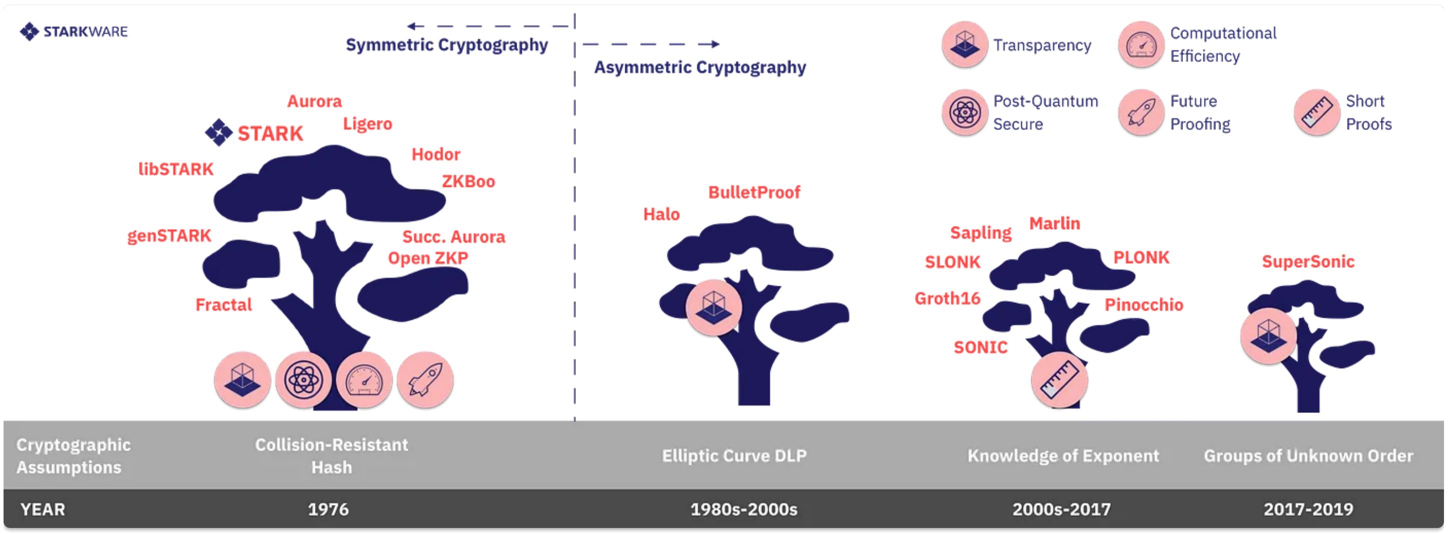
Actors in a Zero Knowledge Proof System

- Creator - optional, maybe combined with the prover
- Prover - often called Peggy
- Verifier - often called Victor

Proof of correctness of computation

Quote from Vitalik Buterin
"You can make a proof for the statement "I know a secret number such that if you take the word 'cow', add the number to the end, and SHA256 hash it 100 million times, the output starts with `0x57d00485aa`". The verifier can verify the proof far more quickly than it would take for them to run 100 million hashes themselves, and the proof would also not reveal what the secret number is."

ZKP 'Families'



from

[The Cambrian Explosion]

Proving Systems

A statement is a proposition we want to prove. It depends on:

- Instance variables, which are public.
- Witness variables, which are private.

Given the instance variables, we can find a short proof that we know witness variables that make the statement true (possibly without revealing any other information).

What do we require of a proof ?

- Completeness: there exists an honest prover P that can convince the honest verifier V of any correct statement with very high probability.
- Soundness: even a dishonest prover P running in super-polynomial time cannot convince an honest verifier V of an incorrect statement. Note: P does not necessarily have to run in polynomial time, but V does.

To make our proof zero knowledge we also need 'zero knowledginess'

To oversimplify: represented on a computer, a ZKP is nothing more than a sequence of numbers, carefully computed by Peggy, together with a bunch of boolean checks that Victor can run in order to verify the proof of correctness for the computation.

A zero-knowledge protocol is thus the mechanism used for deriving these numbers and defining the verification checks.

Interactive v Non Interactive Proofs

Non-interactivity is only useful if we want to allow multiple independent verifiers to verify a given proof without each one having to individually query the prover.

In contrast, in non-interactive zero knowledge protocols there is no repeated communication between the prover and the verifier. Instead, there is only a single "round", which can be carried out asynchronously.

Using publicly available data, Peggy generates a proof, which she publishes in a place accessible to Victor (e.g. on a distributed ledger). Following this, Victor can verify the proof at any point in time to complete the “round”. Note that even though Peggy produces only a single proof, as opposed to multiple ones in the interactive version, the verifier can still be certain that except for negligible probability, she does indeed know the secret she is claiming.

Succint v Non Succint

Succinctness is necessary only if the medium used for storing the proofs is very expensive and/or if we need very short verification times.

Proof v Proof of Knowledge

A proof of knowledge is stronger and more useful than just proving the statement is true. For instance, it allows me to prove that I know a secret key, rather than just that it exists.

Argument v Proof

In a proof, the soundness holds against a computationally unbounded prover and in an argument, the soundness only holds against a polynomially bounded prover.
Arguments are thus often called "computationally sound proofs".

The Prover and the Verifier have to agree on what they’re proving. This means that both know the statement that is to be proven and what the inputs to this statement represent.