Lesson 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

Lesson Topics

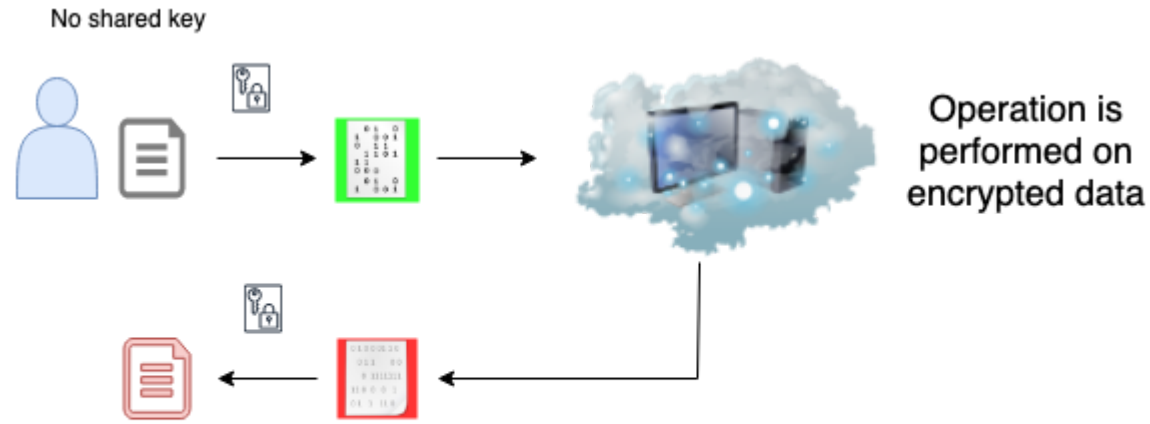
- Cryptographic Background
- Complexity Theory
- ZKP Theory - zkSNARKS
- Zokrates
- zkSNARK process in more details

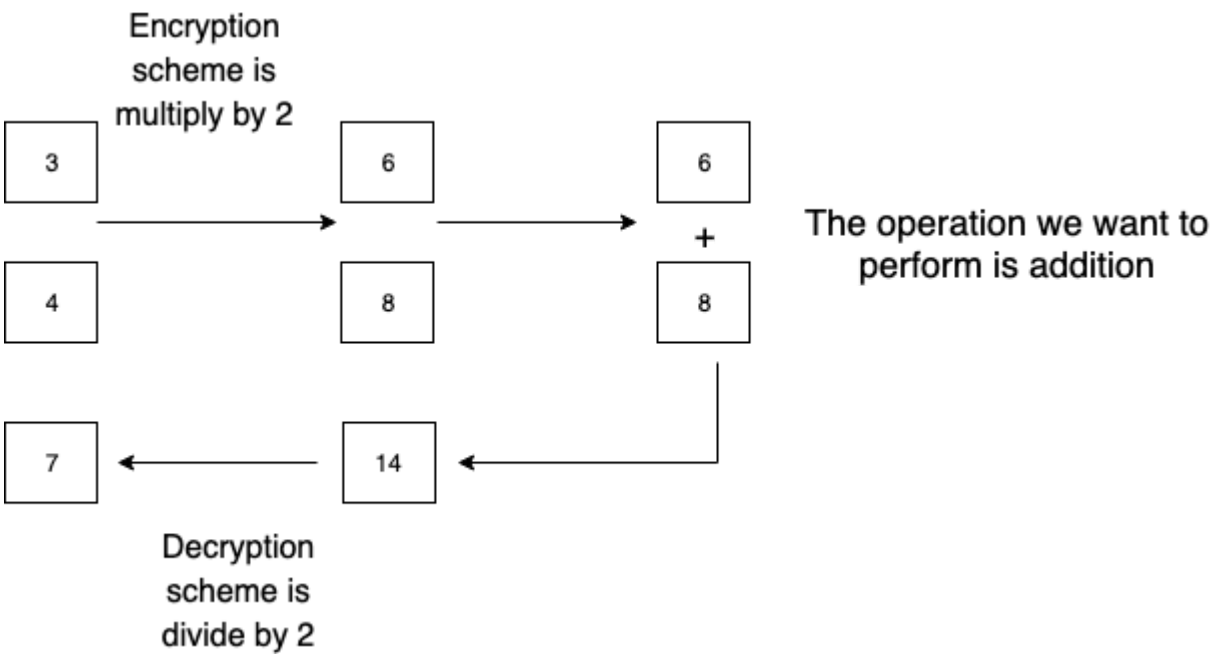
Cryptographic Background

(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.

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

"Screenshot 2022-03-01 at 11.04.49.png" is not created yet. Click to create.

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.

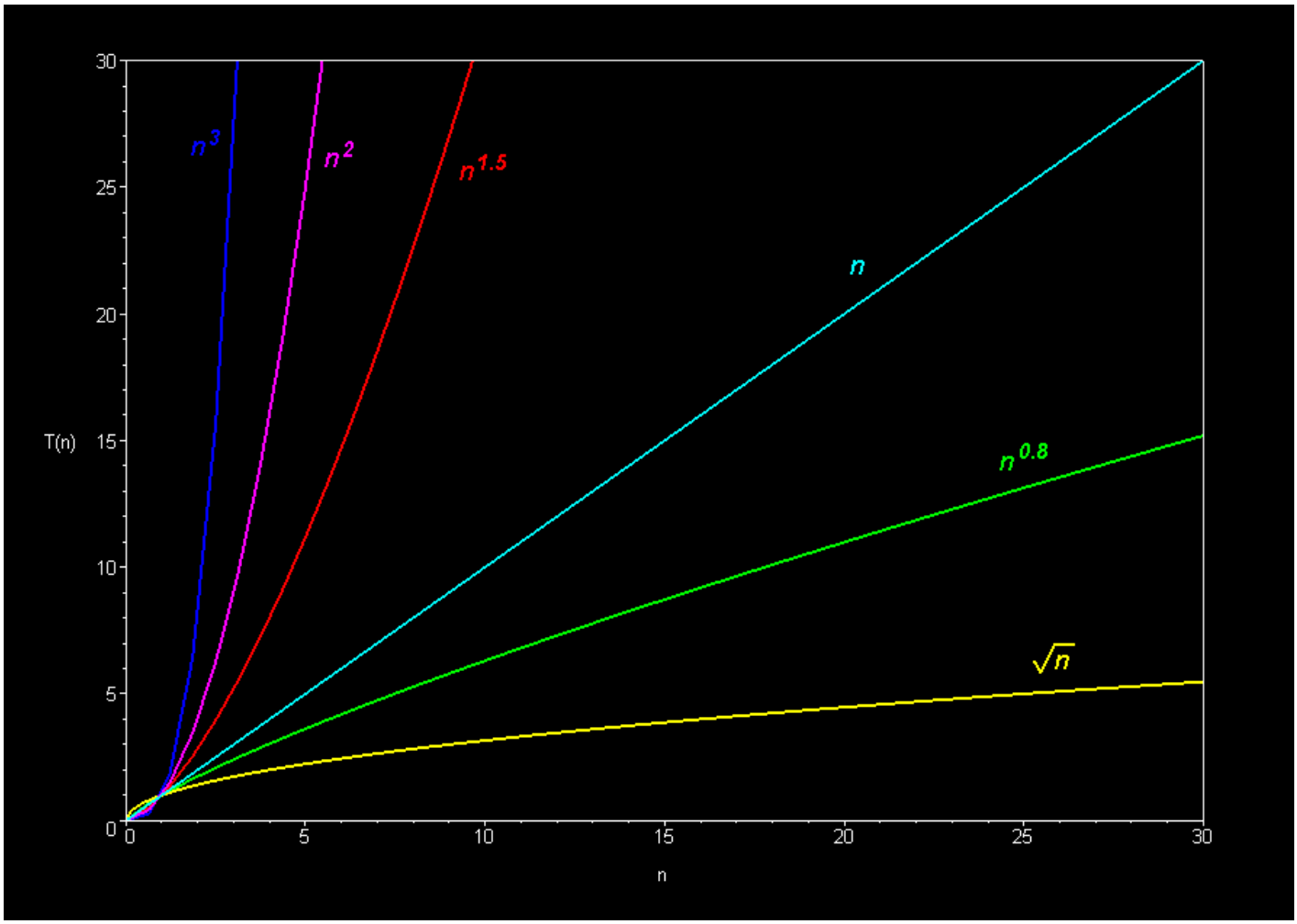
From "Everything provable is provable in zero knowledge"
https://dl.acm.org/doi/pdf/10.5555/88314.88333

"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."

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 😏

Zero Knowledge Proof theory part 1 - zkSNARKS

Currently zkSNARKS are the most common proof system being used, for example they form the basis for the privacy provided in ZCash.

The process of creating and using a zk-SNARK can be summarised as

A zk-SNARK consists of three algorithms C, P, V defined as follows:

The Creator takes a secret parameter lambda and a program C, and generates two publicly available keys:

- a proving key pk
- a verification key vk

These keys are public parameters that only need to be generated once for a given program C. They are also known as the Common Reference String.

The prover Peggy takes a proving key pk, a public input x and a private witness w.
Peggy generates a proof $pr = P(pk, x, w)$ that claims that Peggy knows a witness w and that the witness satisfies the program C.

The verifier Victor computes $V(vk, x, pr)$ which returns true if the proof is correct, and false otherwise.
Thus this function returns true if Peggy knows a witness w satisfying

$C(x, w) = true$

Trusted Setups and Toxic Waste

Note the secret parameter lambda in the setup, this parameter sometimes makes it tricky to use zk-SNARK in real-world applications. The reason for this is that anyone who knows this parameter can generate fake proofs.
Specifically, given any program C and public input x a person who knows lambda can generate a proof $pr2$ such that $V(vk, x, pr2)$ evaluates to true without knowledge of the secret w.

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.

Polynomials 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

In general, there is a rule that if a polynomial P is zero across some set
$S = x1, x2 \ldots xn$ then it can be expressed as
$P(x) = Z(x) * H(x)$, where
$Z(x) = (x - x1) * (x - x2) * \ldots * (x - xn)$ 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.

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.

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 explanation.

Zokrates - a toolbox for zkSNARKs on Ethereum.

See repo
Zokrates was the first project to allow (easy) creation of proofs on Ethereum

ZoKrates helps you use verifiable computation in your DApp, from the specification of your program in a high level language to generating proofs of computation to verifying those proofs in Solidity.
Documentation : Zokrates

A preview of the process flow in Zokrates

We can see this workflow in Zokrates :

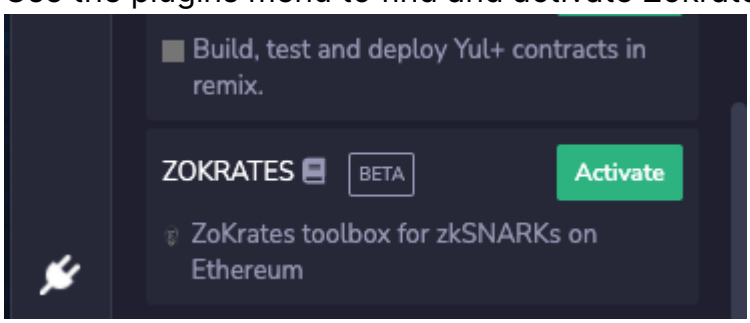
-The Creator writes and compiles a program in the Zokrates DSL

- The Creator / Prover generates a trusted setup for the compiled program
- The Prover computes a witness for the compiled program
- The Prover generates a proof - Using the proving key, she generates a proof for a computation of the compiled program
- The Creator / Prover exports a Verifier - Using the verifying key she generates a Solidity contract which contains the generated verification key and a public function to verify a solution to the compiled program

We can use Zokrates in Remix , it is also available as a program on Linux / Mac / Windows, or a docker container

Zokrates in Remix

Use the plugins menu to find and activate Zokrates



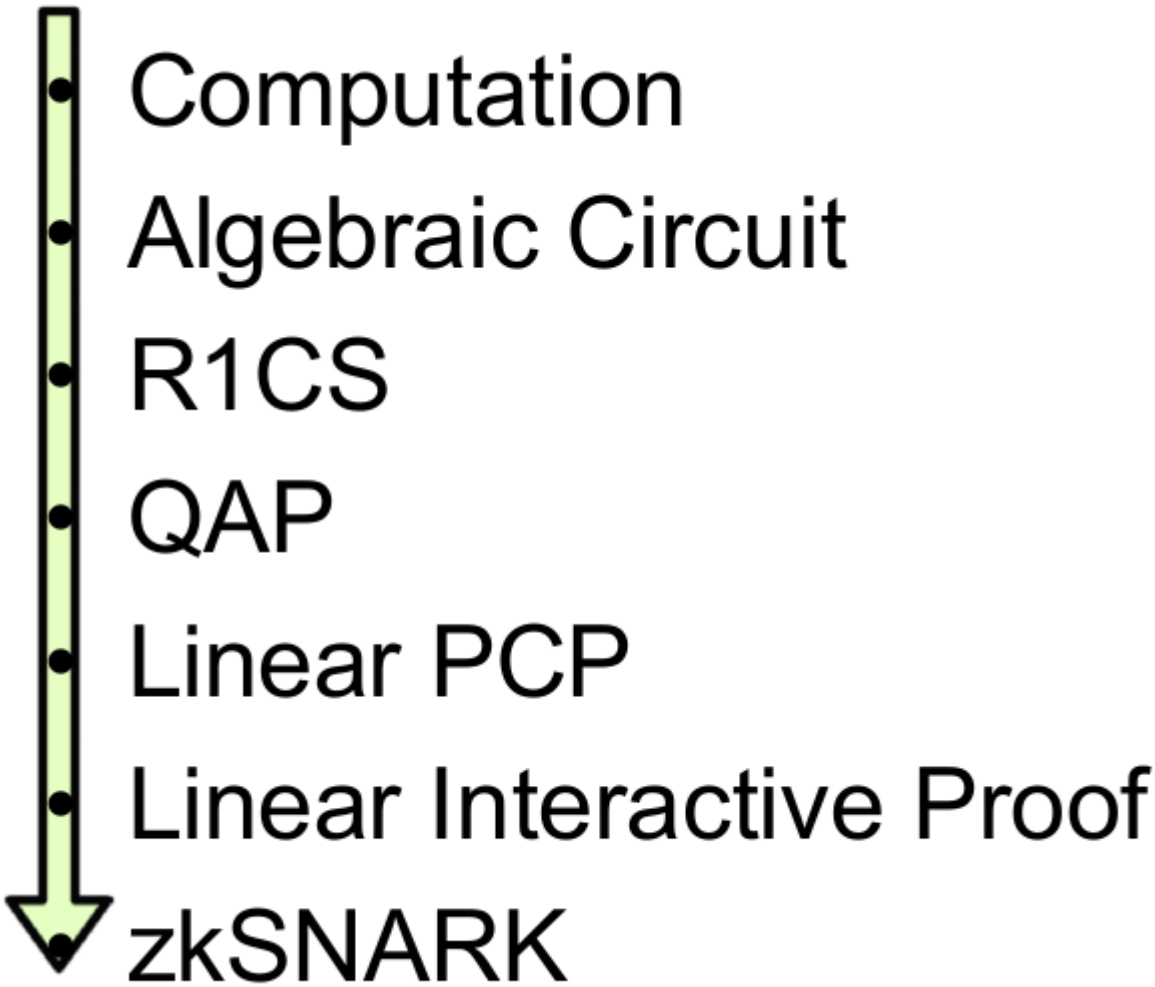
The Zokrates DSL is more limited than say Solidity, see the documentation

ZKSnark Process

General Process

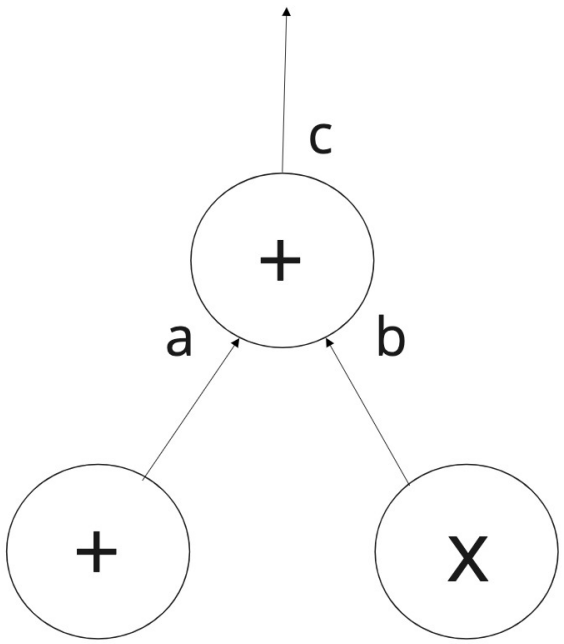
- Arithmetisation
  - Flatten code
  - Arithmetic Circuit
  - R1CS
  - QAP
- Polynomials
- Polynomial Commitment Scheme
- Inner product argument
- Cryptographic proving system
- Make non interactive

Transformations in SNARKS



1. Trusted Setup
   ZKSNarks require a one off set up step to produce prover and verifier keys. This step is generally seen as a drawback to zkSNARKS, it requires an amount of trust, if details of the setup are later leaked it would be possible to create false proofs.

2. A High Level description is turned into an arithmetic circuit
   The creator of the zkSNARK uses a high level language to specify the algorithm that constitutes and tests the proof.
   This high level specification is compiled into an arithmetic circuit.
   An arithmetic circuit can be thought of as similar to a physical electrical circuit consisting of logical gates and wires. This circuit constrains the allowed inputs that will lead to a correct proof.



3. Further Mathematical refinement
   The circuit is then turned into a an R1CS, and then a series of formulae called a Quadratic Arithmetic Program (QAP).
   The QAP is then further refined to ensure the privacy aspect of the process.
   The end result is a proof in the form of series of bytes that is given to the verifier. The verifier can pass this proof through a verifier function to receive a true or false result. There is no information in the proof that the verifier can use to learn any further information about the prover or their witness.

Trusted Setups

From ZCash explanation :

"SNARKs require something called “the public parameters”. The SNARK public parameters are numbers with a specific cryptographic structure that are known to all of the participants in the system. They are baked into the protocol and the software from the beginning.

The obvious way to construct SNARK public parameters is just to have someone generate a public/private keypair, similar to an ECDSA keypair, (See ZCash explanation) and then destroy the private key.

The problem is that private key. Anybody who gets a copy of it can use it to counterfeit money. (However, it cannot violate any user’s privacy — the privacy of transactions is not at risk from this.)"

ZCash used a *secure multiparty computation* in which multiple people each generate a “shard” of the public/private keypair, then they each destroy their shard of the toxic waste private key, and then they all bring together their shards of the public key to to form the SNARK public parameters.
If that process works — i.e. if *at least one of the participants* successfully destroys their private key shard — then the toxic waste byproduct never comes into existence at all.

They have recently introduced Halo2 which eliminates the need for a trusted setup

Halo2 uses the curves Pallas and Vesta (collectively Pasta) which are also used by Mina

Pallas: y^2 = x^3 + 5y2=x3+5 over GF(0×40000000000000000000000000000000224698fc094cf91b992d30ed00000001)

Vesta: y^2 = x^3 + 5y2=x3+5 over GF(0×40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001)

The use “nested amortization”— repeatedly solving over cycles of elliptic curves so that computational proofs can be used to reason about themselves efficiently, which eliminates the need for a trusted setup.

Transforming our problem into a QAP

Lets look first at transforming the problem into a QAP, there are
3 steps :

- code flattening,
- conversion to a rank-1 constraint system (R1CS)
- formulation of the QAP.

Code Flattening

We are aiming to create arithmetic and / or boolean circuits from our code, so we change the high level language into a sequence of statements that are of two forms

x = y (where y can be a variable or a number)
and
x = y (op) z

(where op can be +, -, *, / and y and z can be variables, numbers or themselves sub-expressions).

For example we go from

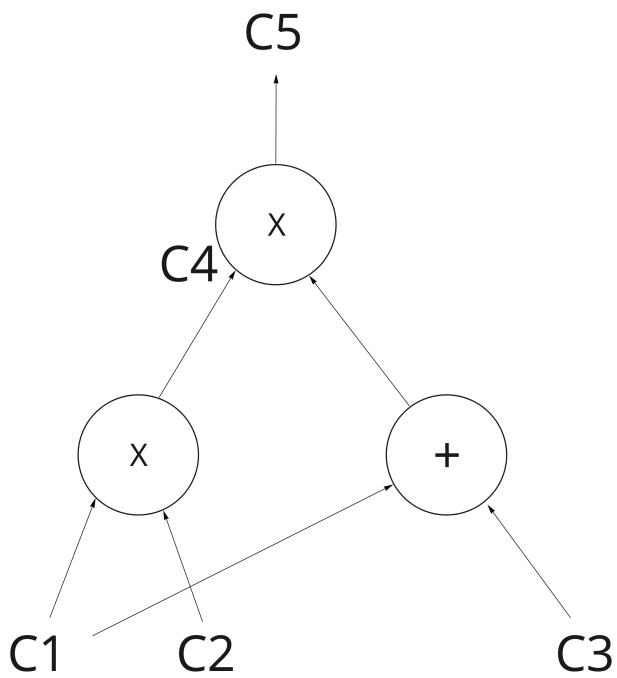
```
def qeval(x):
    y = x**3
    return x + y + 5
```

to

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

Arithmetic Circuit

This is a collection of multiplication and addition gates



Rank 1 Constraint Systems

Constraint languages can be viewed as a generalization of functional languages:

- everything is referentially transparent and side-effect free
- there is no ordering of constraints
- composing two R1CS programs just means that their constraints are simultaneously satisfied.

(From http://coders-errand.com/constraint-systems-for-zk-snarks/)

The important thing to understand is that a R1CS is not a computer program, you are not asking it to produce a value from certain inputs. Instead, a R1CS is more of a verifier, it shows that an already complete computation is correct .

The arithmetc circuit is a compositon of multplicatve sub-circuits (a single multiplcation gate and mutiple addition gates)

A rank 1 constraint system is a set of these sub-circuits expressed as constraints, each of the form:
$AXB = C$
where A, B, C are each linear combinations c1· v1+ c2· v2+ ...
The c_i are constant field elements, and the v_i are instance or witness variables (or 1).

- $AXB = C$ doesn't mean C is computed from A and B just that A, B, C are consistent.

More generally, an implementation of $x = f(a, b)$ doesn't mean that x is computed from a and b, just that x, a, and b are consistent.

Thus our R1CS contains :

- the constant 1
- all public inputs
- outputs of the function
- private inputs
- auxilliary variables

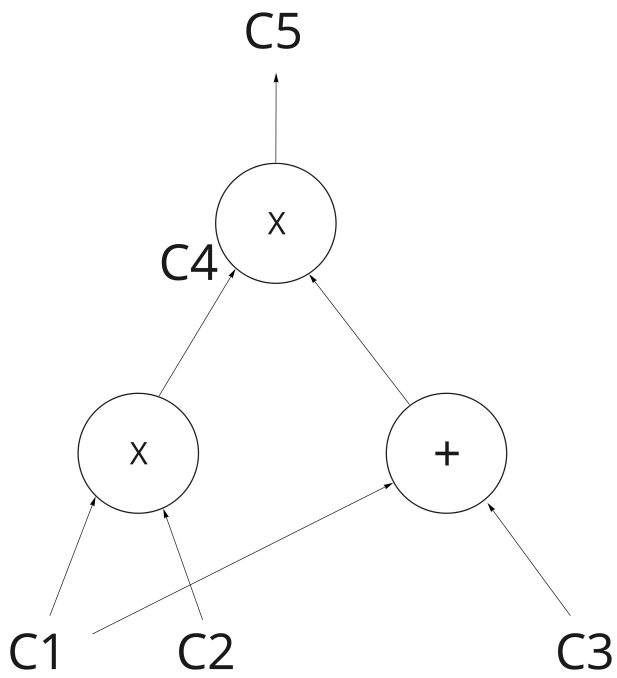
The R1CS has

- one constraint per gate;
- one constraint per circuit output.

Example

Assume Peggy wants to prove to Victor that she knows
$c1, c2, c3$ such that
$(c1 \cdot c2) \cdot (c1 + c3) = 7$

We transform the expression above into an arithmetic circuit as depicted below



A legal assignment for the circuit is of the form:

(c1, . . . , c5), where c4 = c1 · c2 and c5 = c4 · (c1 + c3).