Lesson 2

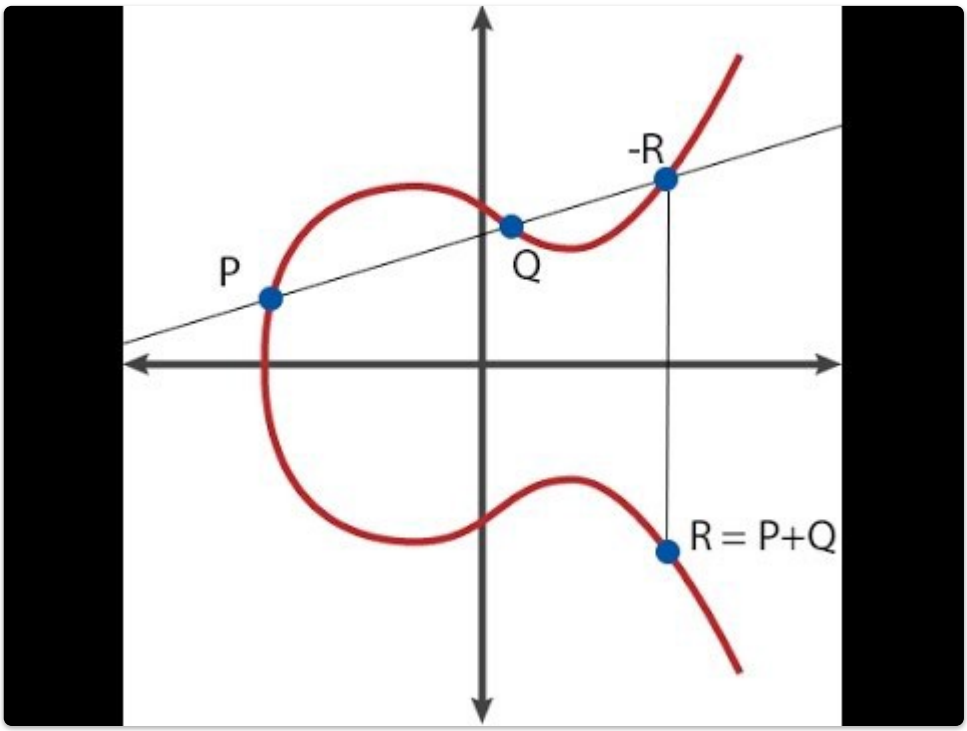
Elliptic Curves

The defining equation for an elliptic curve is for example $y^2 = x^3 + ax + b$
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.

We often use 2 families of curves :

MONTGOMERY CURVES



For example curve 22519 with equation $y^2 = x^3 + 486662x^2 + x$

Curve 25519 gives 128 bits of security and is used in the Diffie–Hellman (ECDH) key agreement scheme
BN254 / BN_128 is the curve used in Ethereum for ZKSNARKS
BLS12-381 is the curve used by ZCash

EDWARDS CURVES

The general equation is $ax^2 + y^2 = 1 + dx^2y^2$ with a = 1
for some scalar d which can be 0 or 1.

If a <> 1 they are called Twisted Edwards Curves
Every twisted Edwards curve is birationally equivalent to a Montgomery curve

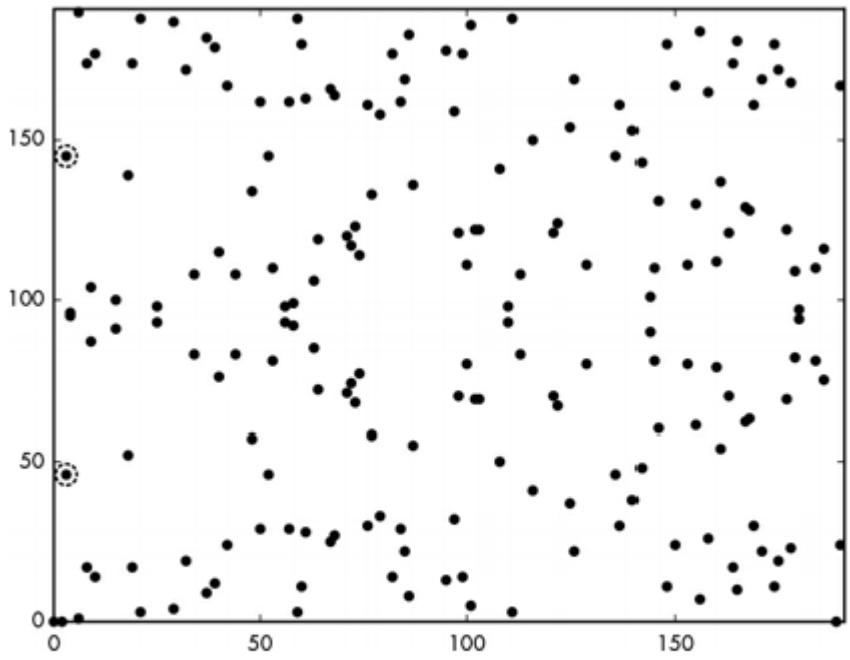


Figure 12-2: The elliptic curve with the equation $y^2 = x^3 - 4x$ *over* $\mathbf{Z}_{191}$*, the set of integers modulo 191*

From
Serious Cryptography Jean-Philippe Aumasson

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.

Given an input value x, the knowledge of the secret key SK allows one to compute $y = F_{SK}(x)$ together with the proof of correctness π_x. This proof convinces every verifier that the value $y = F_{SK}(x)$ is indeed correct with respect to the public key of the VRF. We can view VRFs as a commitment to a number of random-looking bits

The owner of a secret key can compute the function value as well as an associated proof for any input value. Everyone else, using the proof and the associated public key or verification key can check that this value was indeed calculated correctly, yet this information cannot be used to find the secret key.

Algorand have released it as an extension to the libsodium library

Such functions are ideal to find block producers in a blockchain in a trustless verifiable way.

zkSNARKS

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.

Zokrates

ZoKrates is a toolbox for zkSNARKs on Ethereum. It 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

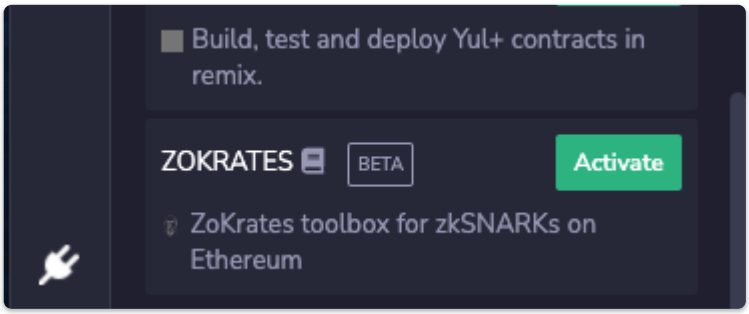
We can see this workflow in Zokrates :

-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

Use the plugins menu to find and activate Zokrates



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

TYPES

-field (positive integer values)
-bool
-u8/u16/u32/u64 (unsigned integers of various lengths)
-Static Arrays - can be multi dimensional
-Structs

That gives us an overview of the process, lets now turn to the theoretical underpinnings, starting with a look at what it means to have a proof.

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.

zkSNARKS

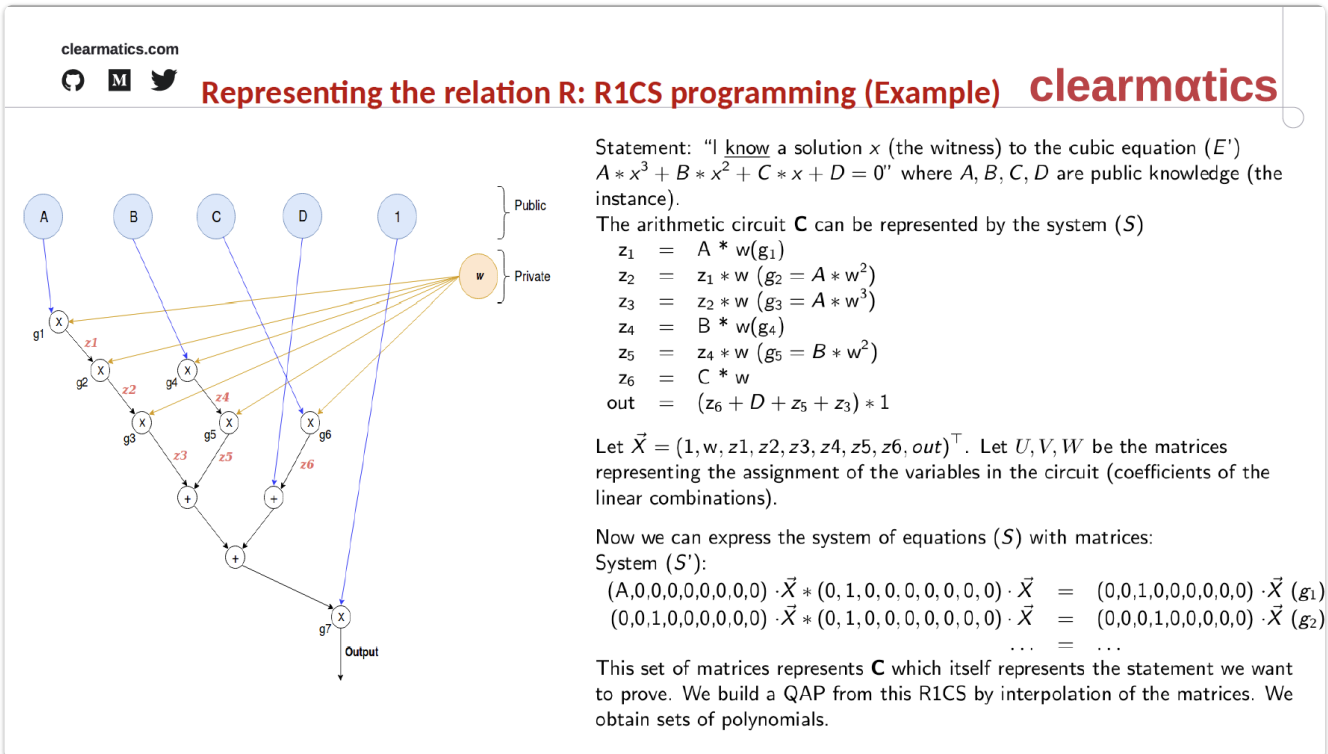
Currently zkSNARKS are the most common proof system being used, they form the basis for the privacy provided in ZCash.

zkSNARK stands for zero knowledge Succint Non interative Argument of Knowledge. The features of succintness (they are small in size and the amount of computation required) and Non Interaction (a single step is sufficient to complete the proof) has helped their adoption in applications.

Simplified Overview of the process needed to create a zkSNARK

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 spcify 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 contrains the allowed inputs that will lead to a correct proof.



3. Further Mathematical refinement
   The circuit is then turned into 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.

Real Life ZKP choices

- We don't always need snarks,

Other useful techniques

- Blind signatures
- Accumulators
- Pedersen commitment
- Sigma protocols

Non SNARK proof of age > 18

See Proving age with hash chains

Problems

There maybe a problem trusting the witness , so how do we know that the witness value is true in a real world sense, we may need a combination of other sites and oracles