DECENTRIQ

ACADEMIA

Introduction to zk-SNARKs (Part 1)

In this series of posts, we will look at ZKPs: a family of probabilistic protocols that has garnered increased popularity with the rise of distributed ledger technology (DLT). Let us introduce some of the theory behind this groundbreaking work and the components of its intricate machinery.

10 SEP 2018 · 14 MIN READ

Overview

In this series of posts, we will look at zero-knowledge proofs (ZKPs): a family of probabilistic protocols, first described by Goldwasser, Micali and Rackoff in 1985, which has garnered increased popularity with the rise of distributed ledger technology (DLT). We will start by introducing some of the theory behind this groundbreaking work and showing how to implement the different components of its intricate machinery. Eventually, this will result in an end-to-end zero-knowledge proof for a toy problem, following the Pinocchio protocol and implemented in Python. The main goal is to provide a gentle introduction to the topic, breaking down the terse mathematics involved in ZKPs and discussing some of the intuition behind it. Along the way, we will also touch on some of the exciting applications, enabled by this technology.

So, what is a ZKP and why should you care?

What the secret is. For example, Peggy might want to prove to Victor that she knows the factorization of a very large non-prime number without revealing the factors; or

method of offloading computation from a weak client to a more computationally powerful worker, which enables the client to *cryptographically verify the correctness of the computation* that was carried out by the worker with a much smaller computational effort compared to executing the original program. This is an extremely powerful paradigm, which affects online privacy, scalability of DLT and mobile phone applications, as well as the security of cloud computing, among many other applications.

Another very promising application of ZKPs lies in the field of machine learning. In this context, the technology can be used to prove possession of specific data without the need of full disclosure as well as to enable 3rd parties to verfiy that specific data has been used in the process of model training or prediction. OpenMined is a pioneering project that is actively working on multiple cryptographic primitives to create an ecosystem for private and secure machine-learning.

But what is a ZKP, really?

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.

Non-interactiveness and succinctness

There is a growing body of research in the field of zero-knowledge and an increasing number of protocols have been suggested since the original publication of Goldwasser, Micali and Rackoff: 1, 2, 3, 4, 5.

One of the main differentiators between these protocols is interactiveness. In an interactive zero-knowledge protocol, the prover and the verifier exchange multiple messages with each other, until the verifier is convinced that the prover knows some

and accepted only if it passes all the boolean checks specified by the protocol. The probability of Peggy not knowing the secret and being able to successfully complete

can continue arbitrarily long until Victor is convinced beyond any reasonable doubt that Peggy knows the secret, despite the fact that he doesn't learn anything about it.

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, the contents of which will be discussed in subsequent sections and posts, 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. Consequently, one advantage of the non-interactive approach is that it is verifier independent. Peggy’s prove is universal for the problem instance at hand (e.g. a given Sudoku puzzle) and any verifier, not only Victor, can later verify Peggy’s claim that she has a solution.

Another dimension along which zero-knowledge protocols differ is succinctness, which deals with the size of the proof generated by Peggy. Succinctness plays an important role for ZKPs published on distributed ledgers, for two reasons: storage space and verification time. Storage can be extremely expensive, e.g. at current prices for Bitcoin, Ethereum, NEO and EOS it costs $\$3.83, \$2.86, \$19.38$ and $\$3.00$ to store 1KB of data on each of those ledgers, respectively, and at peak levels the cost can be much higher! The effects of variable price and expensive storage can be offest by using distributed ledgers which offer cheap and constant pricing, such as Factom ($\$0.001$ per 1KB). Nonetheless, it is still desirable to keep the size of the proof as small as possible. In addition, proofs need to be verifiable quickly, otherwise the gain of offloading a computation to a 3rd party diminishes. Therefore, for on-chain applications it’s crucial that the generated proofs are as short as possible and ideally have a size independent of the problem instance as well as constant verification times.

A natural question to ask is: are there cases in which interactive proofs are preferrable? It turns out that many non-interactive protocols suffer from an annoying

public data used by Peggy when computing her proof. We will discuss the setup in greater detail later, but simply speaking the setup relies on a party or a group of

a source of randomness, commonly referred to as “toxic waste”. Anyone having access to the source of randomness can also produce fake proofs that will be accepted by a verifier following the protocol. This is why it is crucial that the “toxic waste” is destroyed by the parties generating the public numbers, which in turn means those parties need to be trusted. This makes the applicability of some non-interactive zero-knowledge protocols tricky in practice.
In many cases, however, we don’t benefit greatly from non-interactivity or succinctness. 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. Succinctness is necessary only if the medium used for storing the proofs is very expensive and/or if we need very short verification times. However, in general storage is cheap, with distirbuted ledgers being the standout exception, and verification time can also be flexible unless there are hard constraints, such as the ones enforced by inter-block time on public blockchains.

This means that for B2B applications, where two entities are communicating over a secure channel such as HTTPS, we can live with interactive proofs. The companies can put on the masks of Peggy or Victor depending on the use case, with the verifier being responsible for carrying out the trusted setup. Since the interaction is 1-on-1 by nature, the verifier has no interest in faking proofs, it’s not necessary to store the proofs on-chain and Victor can afford to spend more time carrying out the verification, such applications can work in an interactive setting.

Following this small detour, we go back to the main subject of interest for this series -- proofs which are non-interactive and succinct. In particular, we are interested in zero-knowledge **S**uccinct **N**on-interactive **AR**guments of **K**nowledge, a.k.a. zk-SNARKs. zk-SNARKs are the most widely used zero-knowledge protocols, with the anonymous cryptocurrency Zcash and the smart-contract platform Ethereum among the notable early adopters.

QAP: from computation to polynomials

order for them to be applicable the original problem has to be completely transformed

Consider the scenario in which Peggy wants to prove to Victor that she knows a solution to the equation

$$x^2 - 4 = 0$$

without revealing what this solution is. In this case, the solutions are trivial and in reality one doesn't need a zero-knowledge proof for such problems, as anyone would be able to derive the answer. Nonetheless, for the sake of simplicity, this is the example that we will be looking at during this series, keeping in mind that instead of a second degree polynomial this could just as easily be a polynomial of degree 1 million.
So, how do we go about encoding this problem?

A detailed description of the process is available in Vitalik Buterin's excellent post on Quadratic Arithmetic Programs (QAPs), which we highly recommend as an additional reading. Here we provide a shorter summary of the transformation.

We start by representing the problem in code, e.g. in Python we can define the following simple function encoding the computation:

```
1 def f(x):
2     y = x ** 2 - 4
```

Remember that the goal is for Peggy to evaluate the above function at some secret value x known only to her, with the evaluation leading to y being set to 0. Clearly, we have at least two problems:

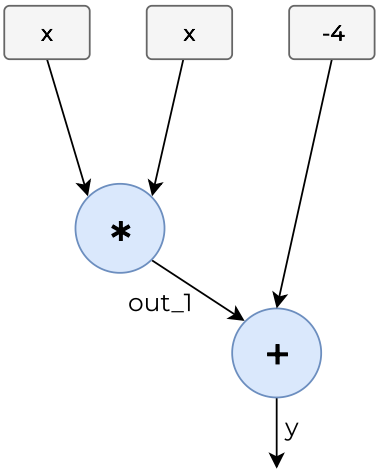
- we need to find a way to allow Victor to get a "peek" of the value of y at the end of the function execution, without him also learning the value of x
- since Peggy is the one running the program, in the current formulation of the code Victor has no control over what code she runs exactly, e.g. she could cheat by just running the code $y = 0$ which would satisfy the condition

conversion to a QAP. This is a three step process: code flattening, conversion to a rank-1 constraint system (R1CS) and finally formulation of the QAP.

arbitrarily complex statements and expressions, into a sequence of statements that are of two forms:

- $x = y$ (where y can be a variable or a number)
- $x = y\ (op)\ z$ (where $op \in (+, -, *, /)$ and y and z can be variables, numbers or themselves sub-expressions)

You can think of these statements as gates of an arithmetic circuit. The circuit for our example program would look like this:



The corresponding flattened code is:

```
1 def f(x):
2     out_1 = x * x
3     y = out_1 - 4
```

The next step is to convert the above to a R1CS. A R1CS is a list of triplets of vectors $(\overrightarrow{a_i}, \overrightarrow{b_i}, \overrightarrow{c_i})$ and its solution is a vector $\vec{s}$, such that:

$$\langle \overrightarrow{a_i}, \vec{s} \rangle * \langle \overrightarrow{b_i}, \vec{s} \rangle - \langle \overrightarrow{c_i}, \vec{s} \rangle = 0 \quad \text{for} \quad \forall i$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product of two vectors. The triplets can be interpreted as defining constraints, which need to be satisfied by finding $\vec{s}$ such that the equations above hold.

we define a vector, which will hold the state of our program, i.e. all variables which are used in the program. Each component of this vector will be one variable and the

The variable vector for our program is:

$$\vec{s} = \begin{pmatrix} 1 \\ x \\ out\_1 \\ y \end{pmatrix}$$

Note that the inclusion of 1 as the first component is done in order to encode constants, as we will see shortly.

Having defined this vector, we continue by processing each line of the flattened code sequentially and defining suitable $\overrightarrow{a_i}$, $\overrightarrow{b_i}$ and $\overrightarrow{c_i}$ to encode the computation contained in the current line.

For example, to encode the first line in the function body we set:

$$\overrightarrow{a_1} = (\,0\;1\;0\;0\,)$$
$$\overrightarrow{b_1} = (\,0\;1\;0\;0\,)$$
$$\overrightarrow{c_1} = (\,0\;0\;1\;0\,)$$

To encode the second line we use:

$$\overrightarrow{a_2} = (\,-4\;0\;1\;0\,)$$
$$\overrightarrow{b_2} = (\,1\;0\;0\;0\,)$$
$$\overrightarrow{c_2} = (\,0\;0\;0\;1\,)$$

Let's check the expansion of the encodings for the two lines, while also noting how in order to represent the constant -4 in the second line we made use of the 1 in the first component of $\vec{s}$ for the definition of $\overrightarrow{a_2}$:

Line 1:

$$\langle \overrightarrow{a_2}, \vec{s} \rangle * \langle \overrightarrow{b_2}, \vec{s} \rangle - \langle \overrightarrow{c_2}, \vec{s} \rangle = \sum_{i=1}^{n} a_{2,i} s_i * \sum_{i=1}^{n} b_{2,i} s_i - \sum_{i=1}^{n} c_{2,i} s_i = out\_1 - 4 - y = 0$$

computer code to a set of constraints! The logic of the code has been "transferred" to the triplets of constraint vectors $(\overrightarrow{a_i}, \overrightarrow{b_i}, \overrightarrow{c_i})$. Running the program is equivalent to finding an assigment of $\vec{s}$, which satisfies the R1CS equations
$\langle \overrightarrow{a_i}, \vec{s} \rangle * \langle \overrightarrow{b_i}, \vec{s} \rangle - \langle \overrightarrow{c_i}, \vec{s} \rangle = 0 \quad \text{for} \quad \forall i.$

To complete the transformation to a QAP we need to switch from vectors to polynomials. We start by defining polynomials $A_i(x)$, $B_i(x)$ and $C_i(x)$ for $i \in [1, N]$ where N is the number of elements in any of the constraint vectors (in our case 4). We construct these polynomials by requiring that $A_i(n) = \overrightarrow{a_{n,i}}$ and similarly for $B_i(n)$ and $C_i(n)$ and doing a Lagrange interpolation based on these points.

For our example we require that:

$A_1(1) = 0$
$A_1(2) = -4$
$A_2(1) = 1$
$A_2(2) = 0$

and similarly for B_i and C_i. Doing a Lagrange interpolation, we arrive at the polynomials:

$A_1(x) = -4x + 4$
$A_2(x) = -x + 2$
$A_3(x) = x - 1$
$A_4(x) = 0$

$B_1(x) = x - 1$
$B_2(x) = -x + 2$
$B_3(x) = B_4(x) = 0$

$C_1(x) = C_2(x) = 0$
$C_3(x) = -x + 2$
$C_4(x) = x - 1$

$$A(x) * B(x) - C(x) = H(x) * Z(x)$$

$$\overrightarrow{A(x)} = (\ A_1(x)\ A_2(x)\ A_3(x)\ A_4(x)\ )$$
$$\overrightarrow{B(x)} = (\ B_1(x)\ B_2(x)\ B_3(x)\ B_4(x)\ )$$
$$\overrightarrow{C(x)} = (\ C_1(x)\ C_2(x)\ C_3(x)\ C_4(x)\ )$$
$$A(x) = \langle \vec{A}, \vec{s} \rangle$$
$$B(x) = \langle \vec{B}, \vec{s} \rangle$$
$$C(x) = \langle \vec{C}, \vec{s} \rangle$$
$$Z(x) = (x-1)(x-2)$$

and

$H(x)$ needs to be such that $A(x) * B(x) - C(x) = H(x) * Z(x)$ holds for $\forall x$.

Still here? Great! At this point, you might be wondering:

- why on Earth did we just do all of this
- what is $Z(x)$
- what is $H(x)$ and who should come up with it

To answer these questions, we first note that, by definition:

$$\overrightarrow{A(1)} = \overrightarrow{a_1}$$
$$\overrightarrow{A(2)} = \overrightarrow{a_2}$$

and similarly for $\overrightarrow{B(x)}$ and $\overrightarrow{C(x)}$. Together with the definition of $A(x)$, $B(x)$ and $C(x)$, this implies that the original R1CS system is equivalent to:

$$A(x) * B(x) - C(x) = 0 \quad \text{for} \quad x \in \{1, 2\}$$

The only way for this to hold is if the polynomial $A(x) * B(x) - C(x)$ is divisible without remainder by $(x-1)(x-2)$, i.e.:

This claimes what $H(x)$ and $Z(x)$ are, but who computes $H(x)$? It must be the prover because by computing $H(x)$ as $(A(x) * B(x) - C(x))/Z(x)$ they show that they

We still haven't answered the most important question: why do we need all of this?

Remember that by expressing our problem as a Python program we ran into a couple of big issues:

- there is no way to verify that Peggy runs the correct program
- it is not clear how to allow Victor to observe the value of y at the end of the execution, without revealing any other information

With the computation expressed as a R1CS, if Victor sees a valid assignment $\vec{s}$, he can be certain that Peggy ran the program corresponding to the R1CS! This already solves one of our main problems.

The reason we transform the R1CS further to obtain a QAP is that mathematically it's much more fruitful to reason about polynomials than R1CSs. In the next section, we leverage our freshly minted problem reformulation to describe one technique central to the construction of zk-SNARKs, which exploits the structure of polynomials.

Evaluation of polynomials at a hidden point

One nifty trick that we can do with polynomials is to evaluate them at a point that is not known to us, a.k.a. blind evaluation of polynomials. Wait, what?! Surely, this must be impossible. Given a polynomial:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

you *must* know x if you want to evaluate the polynomial. Indeed that is the case, but it turns out that if we have access to some additional information, we can evaluate a polynomial without knowing the point of evaluation!

1. one-way: given x it is easy to compute $y = f(x)$, but given y, it is hard to find x

3. $x \neq y \implies f(x) \neq f(y)$

Let's see how such a function can help us do blind evaluation. To make things easier to visualize, assume that Victor wants to ask Peggy to do a blind evaluation of P, which is known to both of them, in such a way that he can verify that Peggy evaluated the polynomial at the point in question. Assume also that f exists and is known to both Peggy and Victor.

First, Victor chooses a point x_0 at which he wants the polynomial to be evaluated. He then feeds it to f to get a new point, which we denote z, and sends z to Peggy. Note that because of the 1st property of f, there is no way for Peggy to find x_0 from z. Furthermore, because of the 3rd property, she cannot come up with $x'_0 \neq x_0$ s.t. $f(x'_0) = z$. Together, these two facts imply that once Peggy receives z from Victor there is no way for her to get back the original x_0.

Consider the case when $P(x) = a_1 x$. Given $z = f(x_0)$, Peggy can trivially compute $P(z) = a_1 z$. Note that because of the linearity of f:

$$P(z) = P(f(x_0)) = a_1 f(x_0) = f(a_1 x_0) = f(P(x_0))$$

This is exactly what we need: by computing $P(z)$, Peggy is effectively computing $f(P(x_0))$, without knowing anything about x_0! Victor can now look at $P(z)$ computed by Peggy and compare it with $f(P(x_0))$ locally, since he knows f, P and x_0. If the two results match, he can be certain that she evaluated P at x_0, since evaluation at $x'_0 \neq x_0$ would have produced a different result.

In order to generalize the above approach to any polynomial, the only thing that needs to change is for Victor to send more information to Peggy. In particular, he sends her:

$$f(1), f(x_0), f(x_0^2), \cdots, f(x_0^d)$$

$$f(P(x_0)) = f(a_0 + a_1 x_0 + \cdots + a_d x_0^d) = f(a_0) + f(a_1 x_0) + \cdots + f(a_d x_0^d)$$

without knowing anything about x_0 and furthermore Victor can check the correctness of the result since he knows f, P and x_0.

Conclusion

In the first post of this series, we discussed some of the high level properties of zero knowledge protocols such as succinctness and interactiveness. Based on that, we introduced the main subject of the series, zk-SNARKs, and showed some of the problems that arise if we try to construct a zk-SNARK for a simple program written in Python. This motivated the introduction of R1CS and QAP and we provided an example of converting computer code to a QAP, which allows us to reason about computer programs in a more formal and mathematically flexible way, bringing us closer to the construction of a zk-SNARK. Finally, we added a powerful tool to our toolbox: blind evaluation of polynomials, which will become central in the subsequent parts of this series.

Next time, we will extend our zk-SNARK Swiss army knife with the tools necessary to construct the "magic" function f, used to evaluate polynomials at a hidden point, and we will provide an implementation of this technique in Python. Until then!

DECENTRIQ

MORE IN ACADEMIA

Membership Inference as a Model Training Metric
13 Mar 2020 – 10 min read

Closing the circle of security in the cloud
15 Oct 2019 – 3 min read

Proving Knowledge of a Hash Pre-Image with ZoKrates
25 Oct 2018 – 23 min read

See all 3 posts →

ACADEMIA

Proving Knowledge of a Hash Pre-Image with ZoKrates

In this blog we implement a problem very typical for blockchain use-cases: proving the knowledge of a pre-image for a given SHA-256 digest. We will begin demystifying this machinery by computing the SHA-256 hash of the number 5. We will navigate through several options using different languages.

You've successfully subscribed to Decentriq!

DECENTRIQ