Demonstrate how Zero-Knowledge Proofs work without using maths

Conference Paper · February 2019

CITATION
1

READS
733

2 authors, including:

Konstantinos Chalkias
University of Macedonia
46 PUBLICATIONS **300** CITATIONS

SEE PROFILE

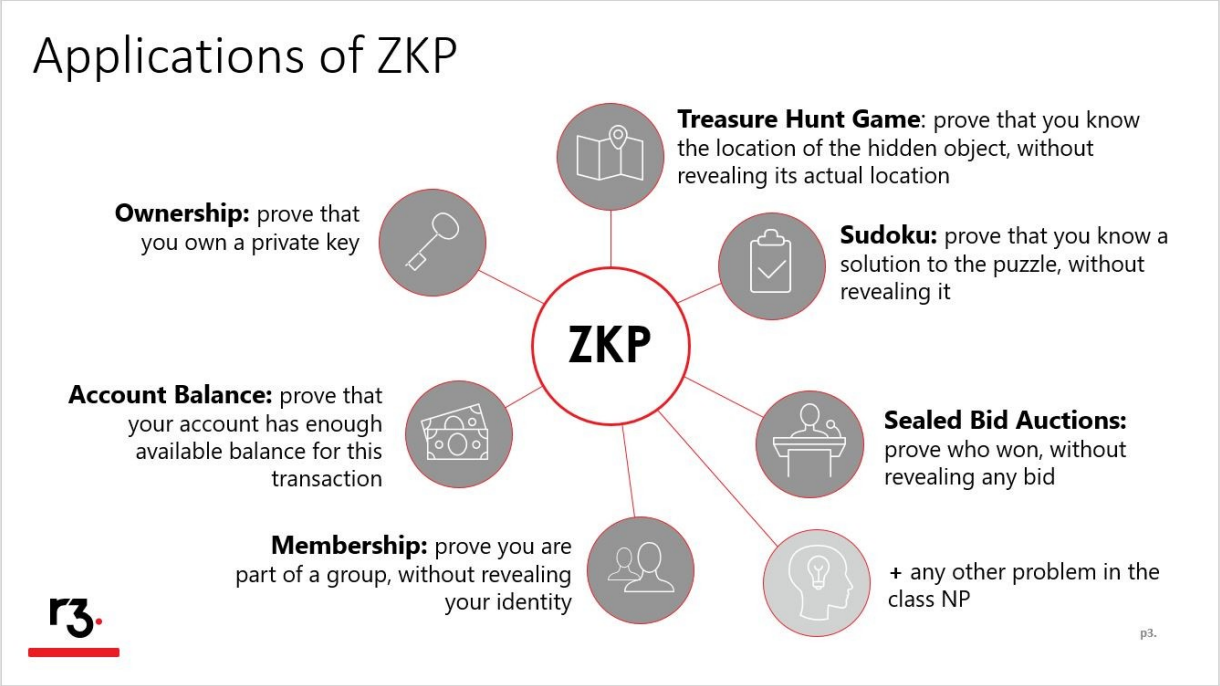
CORDACON 2017 conference

Demonstrate how Zero-Knowledge Proofs work without using maths

Konstantinos Chalkias
R3 / Corda research
konstantinos.chalkias@r3.com

Mike Hearn
R3 / Corda research
mike@r3.com

There are various good articles and examples trying to explain the logic behind Zero Knowledge Proofs (ZKP). If you are not familiar with the concept, the notion of ZKP is that "a Prover wants to convince a Verifier that a statement is true without revealing any further information". And if this is not enough for you to understand why a ZKP is so important and what are its potential applications, a picture is worth a thousand words:



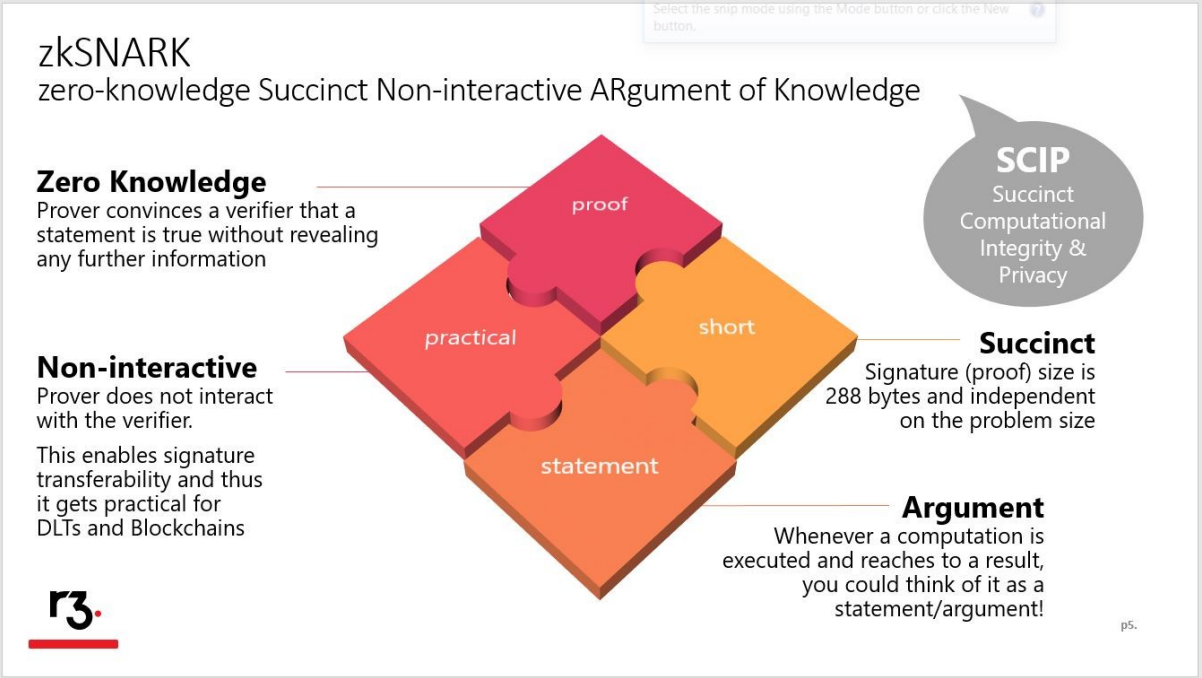
It's amazing that ZKP was initially proposed 30 years ago by MIT researchers Shafi Goldwasser, Silvio Micali and Charles Rackoff and as with many breakthrough ideas, their original paper was rejected a couple of times... and then they won the Gödel Prize. We should also highlight that Goldwasser and Micali recently received the ACM Turing Award for their pioneering work in the field of provable security and interactive proofs, which laid the mathematical foundations that made modern cryptography possible.

It's also true that up until recently, ZKP was not really practical and most of the suggested solutions required multiple interaction rounds between parties or huge proofs. Fortunately, everything changed after the work of Groth and Sahai (2007), Gennaro et al. (2012) and Eli Ben-Sasson et al. (2013). Succinct Non-Interactive Zero Knowledge Proofs (zkSNARKs) are now available and they are the backbone of numerous blockchain solutions these days.

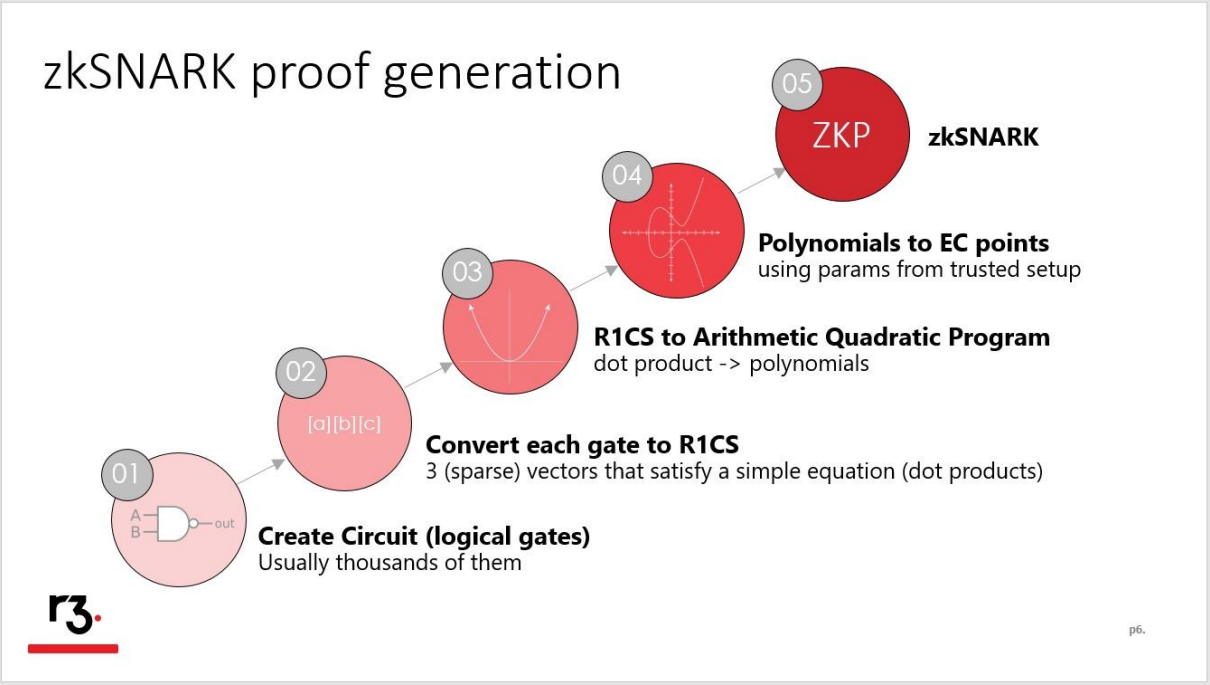
Using zkSNARKs, one can get proofs of any statement in a very efficient manner that is shorter

than the classical mathematical way. **What is a statement?** At the moment, any computer program that reaches to a result, as long as it's not too many cycles (e.g. more than 1-2M cycles). Probably in the future, zkSNARKs and its variants will be able to prove statements about any computer program being executed correctly. For the time being, **financial applications** tend to be the best application candidates, because

1 they are usually very simple (just addition and subtraction of numbers)
2 due to the hype of the cryptocurrency and blockchain projects and products, there is a lot of economic and reputation incentive to them in order to win the battle of privacy preserving.



But, to give you an idea on the steps required to convert a simple program to a zkSNARK, see the diagram below:



I won't get into the mathematical details and if you need to learn more about it, Vitalik posted a comprehensive, easy to follow article about the math processes involved.

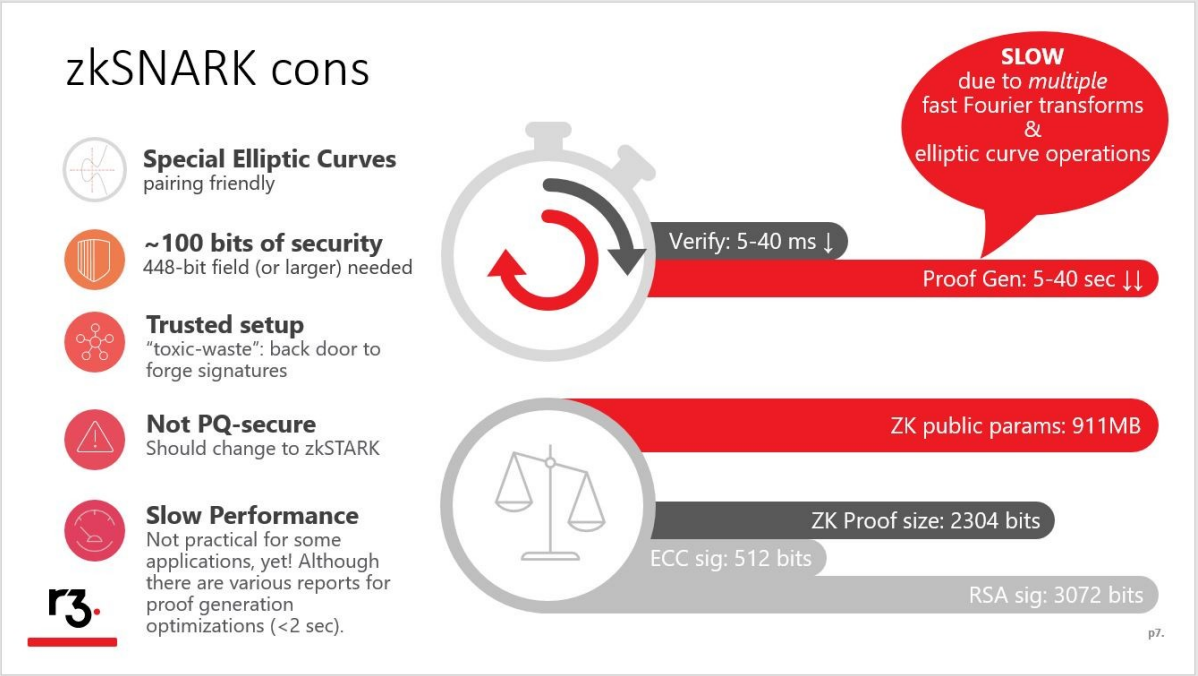
In practice, it might be easier for a blockchain solution to support a concrete circuit to solve one single problem, as one needs to generate the circuit only once, but if you want to generically support numerous or unlimited types of smart contracts, this requires some significant effort and probably dependencies to specialised staff, because:

- Proofs demonstrate solutions to systems of algebraic constraints.
- But equations are not how programmers think of problems.
- Hard to express even simple business logic
- We cannot expect non-specialists to write R1CS

This is how it is to write code in a ZKP manner and it seems it will take a while to get used to it or we will eventually need to create higher-level languages to help developers build secure ZKP applications. But, as Eli Ben-Sasson mentioned, it won't be that complicated in the long run and we've already seen this happening with the advanced operating systems.

```
this->pb.val(ZERO) = FieldT::zero();
// Witness rt. This is not a sanity check.  This ensures the read gadget constrains
// the intended root in the event that both inputs are zero-valued.
zk_merkle_root->bits.fill_with_bits(this->pb, uint256_to_bool_vector(rt));
// Witness public balance values
zk_vpub_old.fill_with_bits(this->pb, uint64_to_bool_vector(vpub_old));
zk_vpub_new.fill_with_bitsthis->pb, uint64_to_bool_vector(vpub_new));
// Witness total_uint64 bits
uint64_t left_side_acc = vpub_old;
for (size_t i = 0; i < NumInputs; i++) left_side_acc += inputs[i].note.value;
zk_total_uint64.fill_with_bits(this->pb, uint64_to_bool_vector(left_side_acc));
// Witness phi
zk_phi->bits.fill_with_bits(this->pb, uint252_to_bool_vector(phi));
// Witness h_sig
zk_h_sig->bits.fill_with_bits(this->pb, uint256_to_bool_vector(h_sig));
for (size_t i = 0; i < NumInputs; i++) {
    // Witness the input information.
    auto merkle_path = inputs[i].witness.path();
    zk_input_notes[i]->generate_r1cs_witness(merkle_path, inputs[i].key, inputs[i].note);
    // Witness macs
    zk_mac_authentication[i]->generate_r1cs_witness();
}
for (size_t i = 0; i < NumOutputs; i++) {
    // Witness the output information.
    zk_output_notes[i]->generate_r1cs_witness(outputs[i]);
}
```

Apart from the complexity, it should be noted that zkSNARKs are still not a panacea and there are a few side-effects to consider before applying this "moon maths" technology. The most important issues, in terms of practicality, are a) the trade-off that the **prover pays a lot** and the verifier pays very little and b) the **trusted initial setup**, a very critical phase that has to be carried out right using multi-party computation and a degree of tolerance on who we trust to run this setup.



A live interactive ZKP example

In case some readers may still find it difficult to understand the concept and follow the details of this post, I will attach the easier in my opinion example on how to demonstrate how a zero knowledge proof works. A google search will show up examples like the Ali Baba cave and some other examples using Sudoku puzzles and coloured graphs. But, what if you are presenting in an event or you are teaching in a CS class or you try to explain the logic in a 8-years old child or you don't have a pen or a computer to draw?

What you need is just two balls with different colours. And if you are in a bar, a glass of white wine and another (same) glass of red wine will do the trick as well.

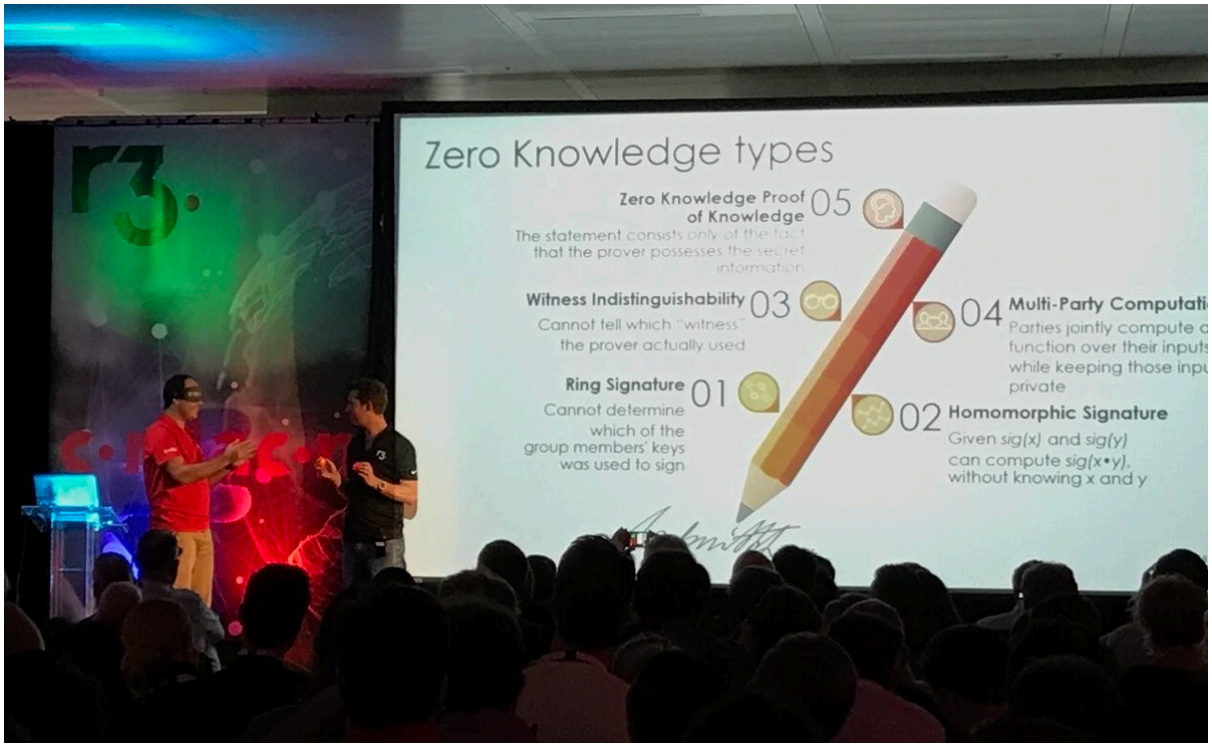
Imagine your friend is colour-blind and you have 2 balls; one red and one green, but they are otherwise identical when you touch them. To your friend they seem completely identical and he is sceptical that they are actually distinguishable. You want to **prove to him they are in fact differently-coloured**, but nothing else, thus **you do not reveal which one is the red and which is the green**.

Here is the proof system. You give the two balls to your friend and he puts them behind his back. Next, he gets one of the balls and brings it out from behind his back. From now on he will always put the ball behind his back and with probability 50% he will reveal one of the two balls. Each time he will ask you "Did I switch the ball?"

By looking at their colours, you can of course say with certainty whether or not he switched them. On the other hand, if they were the same colour and hence indistinguishable, there is no way you could guess correctly with probability higher than 50%.

If you and your friend repeat this "proof" multiple times (e.g. 128), your friend should become convinced that the balls are indeed differently coloured; otherwise, the probability that you would have randomly succeeded at identifying all the switch/non-switches is close to zero.
The above proof is "zero-knowledge" because your friend never learns which ball is green and which is red; indeed, he gains no knowledge about how to distinguish the balls.

CORDACON 2017 conference



How to make it non-interactive?

You might have noticed that the example above is interactive. How could we transform it to be non-interactive as zkSNARKs do? The answer is we could somehow have a **common reference value** that would be used **to simulate your friend's random choices**. Then you (the prover) would send a full answer (proof) including all the simulated challenges and your friend (the verifier) would use the same reference value to re-run the experiment on his side. This common reference value could be a random string, which is drawn from some probability distribution. The assumption is, that these random values are available to all parties, but no party has any influence on their actual choice. And if you simulate the ZK proof with the random choices according to the common reference string, you should get the same result as anyone else validating the protocol's transcript. It's not easy however to construct such a common reference value, but this is what zkSNARKs managed to do.

Why do we care about a proof being interactive or not? This is something that many posts fail to mention, but by making a proof non-interactive we can support proof "transferability", meaning that it can be sent to other verifiers as well. In the above coloured-balls example, you could only convince your friend and no-one else. To avoid repeating these challenges with every party and to verify signatures/proofs of the past (as most if not all of the blockchains require), we need transferable, non-interactive signatures and proofs.

We would like to thank Rayn O'Donnel as his answer in MathOverflow was the inspiration to make the 2-ball example even simpler and Mike Hearn who guided me on this ZKP research and with whom we've presented the shortest ever live ZKP experiment in front of 500 people at CordaCon 2017.