Thesis speech

Stefano Trevisani

December 6, 2022

Good afternoon, I would like to thank you all for coming today, I am really glad to be here to present you my thesis: it was quite a long journey, so I will do my best in order to be both exhaustive and concise. In fact, I want to start by thinking for a moment about what we're doing right now: well, I have to convince you, through this presentation, that the underlying work is hopefully good. We are engaging in what is called an *interactive proof system*.

1

Ok, but what is exactly an interactive proof system? Well, in an interactive proof system we have two parties, called the *prover* and the *verifier*. The prover and the verifier interact with each other by exchanging messages back and forth: the prover wants to convince the verifier that some statement is true, therefore he builds a *proof* of its validity and sends it to the verifier. The verifier, after receiving an alleged proof, checks it to assess its soundness, and finally decides whether he believes the statement to be true or false.

In a formal setting, an interactive proof system is modeled by a pair of connected interactive Turing machines which have access to their own source of randomness. For a proof system to make sense, we assume the verifier to be exponentially less powerful than the prover, otherwise he could simply verify the truthiness of the statement by himself. Furthermore, we allow for the verifier to be fooled with negligible probability. In fact, it was shown that the problems which admit interactive proof systems are the same problems that can be solved in polynomial space.

2

An extremely important application, or rather, class of applications, for proof systems is *verifiable computation*.

For example, a low-power device might want to delegate some heavy load to a powerful cloud provider. However, as the user does not fully trust the provider, he requires a proof to be provided along with the result.

Another possible application lies in computing, say, the electricity bills. The user computes the due amount by himself, and of course the electric company needs some proof that the amount is correct.

Finally, a particularly interesting application lies in verifying blockchain transactions, where users must prove that no money was created or destroyed during a transaction.

3

From the examples I just made we can gather some requirements that our proof system should have in order to be useful: for starters, the reason why we need a proof system in the first place is clearly because the verifier does not trust the prover.

Now, we know that the simplest kind of proof is a witness, but a witness often leaks much more information than just the truthiness of the statement, and the verifier might be a curious one. Hence, we need to create more clever proofs, called *zero-knowledge* proofs, from which the verifier cannot derive, in polynomial time, anything more than whether the statement is true or false.

Another important requirement for verifiable computation is that checking the proof should inherently be a faster process than performing the computation, so we need *succinct* proofs. Furthermore, in many instances there are many verifiers that want to check the same proof: setting up a one-to-one protocol with each of them is not really desirable, so we would like our system to be *non-interactive*.

Finally, the prover is not computationally unbounded in practice: the additional power is given by the knowledge of a witness, or of an efficient way to compute one, which is precisely what we want to keep *secret*. Hence, we call the proof an *argument of knowledge*. A proof system with all the properties we just discussed is called a zero-knowledge succinct non-interactive argument of knowledge system, or ZK-SNARK for short.

4

Ok, but how do we actually set up an efficient ZK-SNARK system? Well, the most promising system that has been devised and then greatly improved in the last ten years, works in the following way: any bounded computation, that is any computation with a fixed number of steps, can be represented through an arithmetic circuit where the fundamental operations are addition and multiplication over an adequate prime field.

The constraints between the input variables, the intermediate variables and the output variables are encoded by systems of bilinear equations called Rank-1 constraint systems, or R1CS for short, whose solution vectors correspond to the values induced by a correct computation of the underlying circuit. Such constraints are then translated into a particular kind of polynomial divisibility

problem called Quadratic Arithmetic Program, or QAP for short. Since verifying the divisibility of two polynomials can be done in a point-wise manner, and checking just one point guarantees an overwhelming confidence, this system allows one to have short proofs.

In order to make the protocol non-interactive, the problem is "moved" in the exponent using a private proving key, and applying a scheme inspired by the Fiat-Shamir heuristic for the discrete logarithm problem. By exploiting a special bilinear map over cyclic groups and a public verification key, any verifier can then check the proof in the exponent: to obtain a zero-knowledge protocol, it is sufficient to apply a randomization scheme on the QAP's polynomials that maintains the divisibility property.

It is worth noting that the generation of the randomness required to build the keys is arguably the most troublesome point of SNARKs, and current research has been very active in trying to solve this problem while maintaining practical feasibility.

5

As we said earlier, an archetypical application of ZK-SNARK systems lies in the verification blockchain commitments, say for privacy-preserving cryptocurrencies. In the plain setting, whenever a transaction is performed, details like the pseudonyms of the payer and the recipient, as well as the involved amount of currency, are inserted as leaves of a Merkle Tree data structure.

By applying a cryptographic compression function, the tree is the constructed in a bottom-up fashion, ending at the root, which is then appended to the blockchain. The value contained in the root is a cryptographic *commitment* for all the transactions at the bottom level: even a slight change of the values in a single leaf will produce a "chain-reaction" that will change the contents of the root.

When a verifier in the network wants to check that a payer is behaving honestly, it will ask him to provide the *authentication path* of the values connecting its transaction to the root of the tree. For example, in the picture, the transaction is the green node which, together with the blue nodes, constitutes its authentication path.

Clearly, such a scheme does not provide any kind of privacy, as everyone in the network will get to know the pseudonyms and the amount of money involved in the transaction. In order to make it zero-knowledge, we need to embed the Merkle Tree computation in an arithmetic circuit. Since the only operation being effectively performed in a Merkle Tree is the execution of the underlying compression function, in order to make the scheme efficient in a zero-knowledge setting, we must have a compression function whose constraint system is as compact as possible, without losing security.

6

Technically speaking, a one-way compression function is a function which maps a string of some fixed length to another, shorter string, in such a way that it is very efficient to compute but infeasible to invert or to find collisions.

Traditionally, one-way compression functions are derived from one-way permutations by applying a secure construction scheme, such as the Davies-Meyer or the Sponge construction, which is shown in the figure.

7

The official standard for secure compression functions proposed by the American National Institute of Standards and Technology is the SHA family of hash functions. The SHA transformation is designed to work over extensions of the boolean field, with the underlying operations being bitwise AND, XOR, rotation and addition modulo a power of two.

While this results in extremely efficient implementations both in hardware, such as on FPGAs, and in software, such as on x86 CPUs, it is not ideal for ZK-SNARK designs that work natively over prime fields, and for which the bitwise operations must be emulated through expensive circuitry. In fact, the 256-bit variant of SHA requires about twenty-five thousands R1CS constraints to be represented, which in turn results in a QAP involving huge polynomials.

8

So, it comes naturally to think that in order to obtain efficient cryptographic primitives for SNARK systems, we should work natively in the underlying prime field. Such *arithmetization-oriented* constructions, which at their core consists of keyed permutations, only use addition and multiplication to mix up the message.

Clearly, these transformations can be easily represented by a polynomial: while traditional cryptanalysis, such as meet in the middle and differential attacks, are usually not a problem, algebraic attacks which try to interpolate, factorize or solve the polynomial pose a major threat and their hardness is what influences the number of rounds required to achieve an adequate security level.

9