zk-SNARKs
Analysis and Implementation on Ethereum

MISTIC: Inter-university Master's Degree in Security of Information and Communication Technologies

Master's thesis report for the studies of the Inter-university Master's Degree in Security of Information and Communication Technologies presented by Alberto Ballesteros Rodríguez and directed by Jordi Herrera Joancomartí.

Universitat Oberta de Catalunya, Madrid, 2020

Abstract

Given the high complexity of understanding zero-knowledge SNARK proofs, this document provides a relaxed guide to introduce anyone who can understand how cryptosystems work into zk-SNARKs. Essentially, this master's thesis aims to be a reference document about zk-SNARKs covering the mathematical composition of these proofs clearly and their application in a general purpose public blockchain such as Ethereum. In this dissertation there is no discussion about security or optimization. Also, real world applications schemes are proposed in order to illustrate the applicability of these proofs.

Keywords: Zero Knowledge Proofs, SNARK, QAP, Quadratic Arithmetic Programs, Arithmetic Circuits, Verifiable Computation, Blockchain, Ethereum, ZoKrates, Nightfall, AZTEC

Contents

List of Figures

Acronyms

CRS	Common Reference String
DLT	Distributed Ledger Technology
ECC	Elliptic Curve Cryptography
EVM	Ethereum Virtual Machine
NIZK	Non-Interactive Zero-Knowledge
PCP	Probabilistically Checkable Proof
QAP	Quadratic Arithmetic Programs
QSP	Quadratic Span Programs
RISC	Reduced Instruction Set Computer
SNARG	Succinct Non-Interactive Argument
SNARK	Succinct Non-Interactive Argument of Knowledge
ZKP	Zero-Knowledge Proof or Protocol

CHAPTER 1

Introduction

Although there are different blockchains focused on privacy nowadays such as Monero or ZCash, their current protocol implementation does not support smart contracts.

Other protocols that allow developers coding decentralized applications, where the logic is executed from smart contracts, such as Ethereum, and where transactions information is public, are in current development to add a privacy layer, which is a major technological challenge.

Implementations that accept the use of zero-knowledge proofs are continuously being developed, specifically the zk-SNARKs already used by ZCash.

With the success of this project, the main idea is to acquire a knowledge base about the zk-SNARKs and their implementation in a public blockchain, like Ethereum, and to analyze their impact in different applications, e.g. financial services or digital identity.

To accomplish this goal, it will be necessary to carry out a detailed technical analysis of the behaviour of the zk-SNARKs, describing mathematically their composition. In continuation some of the most popular implementations, such as toolboxes or protocols on the Ethereum network will be examined. At the same time, existing examples will be used to illustrate the performance of these tools.

Some of these implementations will allow to describe use cases at a high level view, where all the actors involved will be correctly identified. The purpose is to examine the possibilities from a perspective of execution and visibility within the blockchain.

Finally, as mentioned above, with our knowledge it will be possible to draw up a set of conclusions from all the study process related to the application of zk-SNARKs in different applications to add a privacy layer.

1.1 Goals

As previously commented, our main goal is to provide a reference document based on the knowledge we have about zk-SNARKs on the Ethereum network. So, it is important to define the following list of requirements.

- Analyze and understand how zero-knowledge proofs work, especially zk-SNARKs.
- Get familiar with the existing protocols, networks and tools that use this cryptographic technique in the blockchain ecosystem.
- Analyze and understand the performance of different protocols and toolboxes for the application of zk-SNARKs proofs on Ethereum.
- Exemplify using existing implementations the operation of the zk-SNARKs on Ethereum.
- Study of the most common use cases on which the use of this cryptographic technique could be applied.

1.2 Outline

An overview of the main points of the present thesis are given to clarify the structure based on the following chapters:

I. *Introduction* gives the first approximation to the concept of zk-SNARKs on blockchains, and besides, the goals and the outline are presented.

II. *State of The Art* reviews the literature related to zk-SNARKs and list some of the solutions that apply zero-knowledge techniques on Ethereum.

III. *Basic concepts* provide a complete knowledge base on cryptography in order to be able to understand zk-SNARKs.

IV. *zk-SNARKs* is considered one of the main chapters of the thesis where it is defined and explained how a zk-SNARK proof is constructed from a cryptographic point of view.

V. *zk-SNARKs on Ethereum* covers the analysis of some toolboxes or protocols that implements zk-SNARKs on Ethereum, then use cases derived from the analysis are proposed.

VI. *Conclusion* exposes the ideas acquired throughout the project and proposes future works.

1.3 Planning

In this section a planning of the tasks carried out during this thesis is presented. A Gantt chart is used to illustrate the project schedule based on the goals defined in the previous section. Note that some bars of the chart are only referenced by a letter because the task definition is too large, so an explanation of these tasks is provided below.

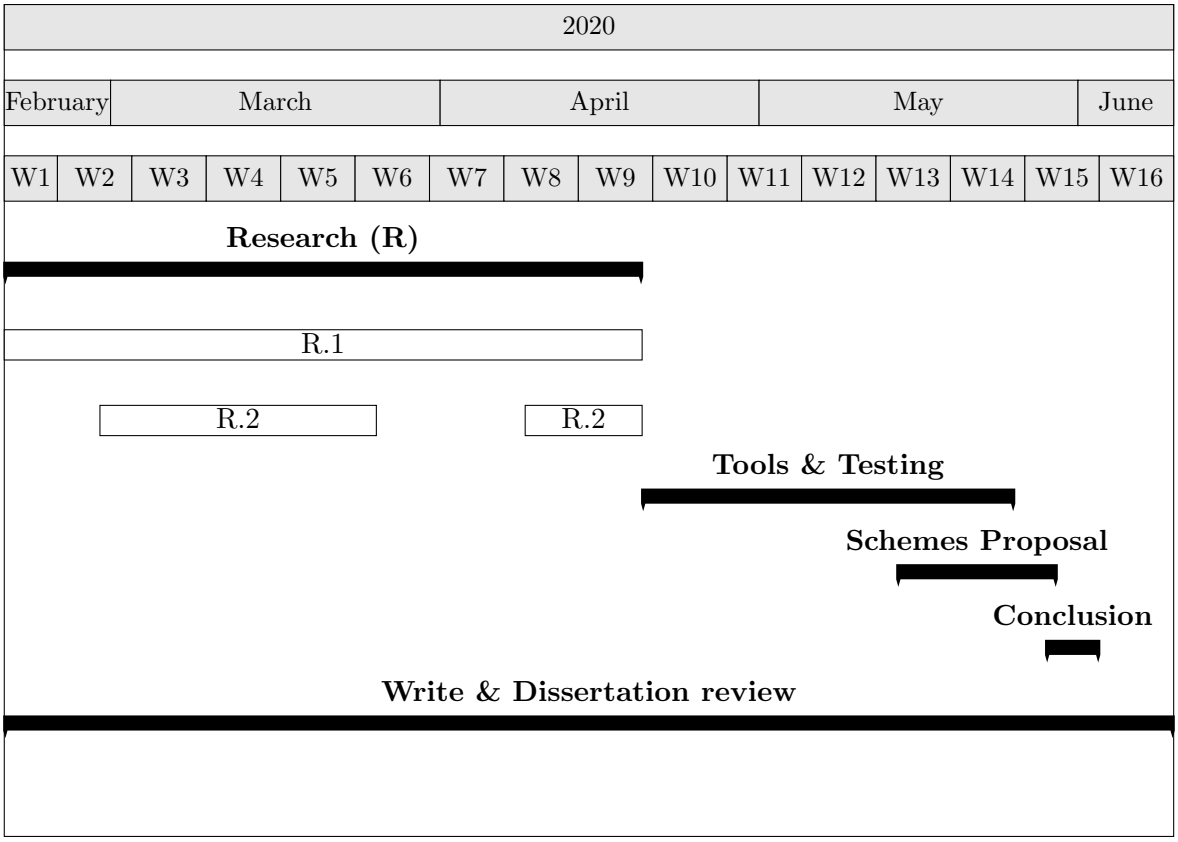
The *Research* task is associated with all the pieces of work that are related to study, analyze and understand the zk-SNARK proofs. Due to the high complexity of these proofs, the job of analyzing and studying them has taken a large amount of time and is represented in the chart by the letter R.1. However, there are some basic concepts that are needed to correctly understand this cryptographic technique, the task of study them is represented by the letter R.2.

The next and more important job during this thesis is the task named as *Tools & Testing*. It is associated with all the job related to the installation, learning, use and programming process in applications that implement zk-SNARKs on Ethereum or pretend to do it in the future.

Taking advantage of all the work done up to this point, some real use cases where zk-SNARK proofs could be applied on decentralized applications are represented in the chart by the task *Schemes Proposal*.

Finally, the *Conclusion* and *Write & Dissertation review* tasks are illustrated in the graph. The former, as it can be seen, has been carried out throughout the project life for being a research work.

Relevant comments regarding some tasks should be mentioned because during the project some difficulties have appeared. On one hand, the time of the *Research* task has been longer than expected because the literature about zk-SNARKs is not easy to understand although one has previous knowledge about cryptography. On the other hand, in the *Tools & Testing* task due to the high requirements of EY Nightfall, the suite could not be tested, but this has not a negative effect in the project because a demonstration of how to use Nighftall is not necessary since the only way to use it is through a user interface which is really easy to interact with.



CHAPTER 2

State of The Art

Despite the reach of this project, from a theoretical point of view it is not a new topic or a recently emerged one. Rather, the rise of distributed ledger technology (DLT) it is the reason of the increased popularity of the zero-knowledge protocols.

A revision of the State of Art is provided from the most relevant works about zero-knowledge proofs and SNARKs. Later a review of some existing solutions such as protocols or toolboxes will be carried out to give a wider vision of the current state of the ecosystem.

2.1 Literature Review

Beginning with the first document, zero-knowledge proofs were described by Shafi Goldwasser, Silvio Micali and Charles Rackoff [17, 18]. In this paper the authors introduced a new procedure to prove a theorem by communicating probabilistic and interactive proofs previously defined and showed that it is possible to prove that a number is a quadratic non residue mod m when the additional knowledge communicated is zero (zero-knowledge).

The next improvement was to investigate the possibility of share some information previously between the Prover and the Verifier in order to avoid the interaction to achieve a zero-knowledge proof. Manuel Blum, Paul Feldman and Silvio Micali, showed in [8] a first approximation that interaction can be replaced by sharing a common, short, random string, naming this new variant as non-interactive zero-knowledge proofs. In 1990, Manuel Blum, Alfredo De Santis, Silvio Micali and Giuseppe Persiano based on the previous extended abstract, investigated the possibility of removing the interaction between the Prover and the Verifier using a short random string previously shared [7].

Then, during the 1990s S. Arora and S. Safra in [4] introduced the definition of the probabilistically checkable proofs (PCPs), which are proofs that can be checked with a random algorithm by reading a limited number of bits of the proof using randomness.

In the next decade, while the idea of outsource computing was arising and with the research for efficient Provers and Verifiers, a Prover for outsourced computation that could run in polynomial time [16] was presented by Shafi Goldwasser, Yael Tauman Kalai and Guy N. Rothblum.

A few years later, in view of the above and with the problem of dishonest workers, the concept of verifiable computation using non-interactive proofs [13] was formalized by Rosario Gennaro, Craig Gentry and Bryan Parno.

Jens Groth, in 2010 constructed a non-interactive zero-knowledge argument [19] (without using probabilistically checkable proofs directly) in the common reference string for circuit satisfiability with perfect soundness, completeness and zero-knowledge, but both the CRS size and the Prover computational complexity were quadratic. Helger Lipmaa [28] reduced the complexity of the CRS size to quasilinear, but the Prover computation was still quadratic.

From span programs [24] defined in 1993, an extension of these was defined by the characterization of the NP complexity class, called Quadratic Span Programs (QPSs), which allows to construct efficient SNARKs without using probabilistically checkable proofs [14]. Thus, from the above and with the constructions of Groth and Lipmaa, a non-interactive zero-knowledge argument can be constructed with a linear computational complexity in the CRS size and quasilinear for the Prover computation, that are quick to construct and verify. With this approach they showed how to encode computations as quadratic programs to obtain efficient verifiable computations and zero-knowledge verifiable computations schemes.

So, at this time SNARKs had been defined and demonstrated their applicability to multiple forms of delegating computation, to succinct non-interactive zero-knowledge arguments and to succinct secure two-party computation.

SNARKs [6] came up from the modification of the protocol [11] described by Di Crescenzo and Lipmaa in 2008. Extractable collision-resistant hash functions (ECRHs) were defined and if exist, with the modification, arguments that satisfy the features introduced were called succinct non-interactive arguments of knowledge (SNARK).

Due to all of this, a system to achieve efficient verifiable computation relying only on cryptographic assumptions called Pinocchio [33] was introduced by B. Parno, J. Howell, C. Gentry and M. Raykova, combining QSPs [14] with systems engineering to produce a set of programming tools for verifying computations.

Pinocchio protocol is described as the first general-purpose system to demonstrate verification cheaper than native execution (for some applications). SNARKs implementations and constructions are practical and usable thanks to this protocol, which is the most used currently.

Finally, at the end of 2013 and with the implementation of the Pinocchio protocol, a zk-SNARK implementation supporting program executions on a universal von Neumann RISC machine or architecture [5] was constructed based on two components, an universal circuit generator and a high-performance implementation of a zk-SNARK for verifying satisfiability of arithmetic circuits.

Despite of the first publication dated from 2013, the paper was updated in 2019 due to an error in the original document. A recall of the zk-SNARK protocol proposed in the Pinocchio original paper [33] it is presented on Appendix B [5].

Therefore, even though there are more papers than the listed above, the collection proposed could be considered enough to get a rough idea of the research done until now to understand how zk-SNARKs came about.

2.2 Real-World Applications

The literature review showed in the previous section shows that there is a lot of research related to zero-knowledge proofs. Many solutions or companies have been created from these investigations. In this section a list of some of those ones that use this cryptographic technique somehow will be provided.

There are multiple solutions that use zero-knowledge proofs in the blockchain ecosystem but the ones listed above are those that are on Ethereum or use its standards.

AZTEC:

Functionality: Protocol

https://www.aztecprotocol.com

The Anonymous Zero-knowledge Transactions with Efficient Communication protocol commonly known as AZTEC, is implemented on the Ethereum public blockchain and describes a set of zero-knowledge proofs defining a confidential transaction protocol that can be used on blockchains that supports Turing-complete computation, such as Ethereum.

ZoKrates:

Functionality: Toolbox

https://zokrates.github.io

Toolbox for using zk-SNARKs on the Ethereum network. ZoKrates allows developers to verify proofs in Solidity from the specification of an off-chain program coded in a high level language, giving the possibility of using verifiable computation in the decentralized application (DApp) by linking them to the Ethereum blockchain.

EY nightfall:

Functionality: Tools Suite

https://github.com/EYBlockchain/nightfall

Nightfall is an open source suite of tools that combine ZoKrates and a set of smart contracts, specifically the ERC-20 and ERC-721 Ethereum token standards, designed for a fully privacy token transactions over the Ethereum public blockchain. Other standards could be added to the specification in the future if their use is extended.

Loopring:

Functionality: DEX Protocol

https://loopring.org

An open source protocol for decentralized exchange over the Ethereum network based on the interoperability among decentralized applications with exchange funcionalities allowing their users to trade their assets. In the third version of the protocol, the use of zk-SNARKs was included to improve the throughput.

Tornado:

Functionality: Mixer

https://tornado.cash

Tornado is an Ethereum privacy solution based on zk-SNARKs to ensure privacy transaction, acting like a mixer or a proxy because it uses a smart contract, which accepts Ether or ERC-20 tokens allowing the withdraw to a different address, removing the reference between the withdrawal and the deposit of the original transaction as a privacy preserving mechanism.

CHAPTER 3

Basic Concepts

This chapter covers the knowledge base in order to correctly understand the whole document, but taking into account the nature of this project it is recommended to have previous knowledge about the following concepts:

- Cryptography Elements: Public/Private key and Symmetric/Asymmetric cryptography. About these concepts there are different books [30] or publications, so the reader can choose one of their preference.
- Modular Arithmetic: Finite fields computation and their properties. The reading of the following material [35] is recommended to have a complete understanding about this topic.
- Homomorphic Encryption: Simply know the main idea of this mechanism to be capable to understand a derived technique from this type of encryption. For more information, this paper [34] was the first publication that defined homomorphic encryption.
- Computational Complexity: Although it is not strictly necessary, the classes of problems P and NP are strongly recommended to have a look [3]. It could be useful to have some notions of complexity theory in order to see which problems and computations zk-SNARKs can be used for.

With all this, it is expected that the reader has a basis to address the following sections, which will be related to elliptic curve cryptography and homomorphic encryption. The rest of relevant and necessary concepts for the project will be treated in their corresponding chapters.

3.1 Use of Elliptic Curves in Cryptography

Victor S. Miller [31] and Neal Koblitz [25] independently proposed in 1985 the use of the set of points on a defined elliptic curve over a finite field as the basis for cryptographic systems based on the difficulty of breaking the discrete logarithm.

Let C an elliptic curve over a finite field $\mathbb{F}_q$ where $q = p^m$ and p is prime. The *elliptic curve discrete logarithm problem* (ECDLP) is defined as, supposing two points $P, Q \in C$ over $\mathbb{F}_q$ determine an integer k such that $kP = Q$, if it exists. If P and k are known Q can be computed in an efficient way, but if P and Q are known it is a hard computational problem to find k.

Elliptic curve cryptography (ECC) depends on the theory that the elliptic curve discrete logarithm problem is extremely hard to compute. Note that not all elliptic curves are suitable for cryptography, there are some cases in which the discrete logarithm problem could be solved efficiently using different methods or algorithms, such as:

- Baby-step giant-step (BSGS)
- Index calculus
- Pohlig–Hellman
- Pollard's rho
- Pollard's kangaroo

These are only some of the most common methods, but there are more methods based on summation polynomials that have suggested improved results for ECDLP on elliptic curves. In addition, there are other algorithms based on point decomposition.

In order to choose an elliptic curve over a finite field $\mathbb{F}_q$ and a subgroup of order n, to set up an elliptic curve cryptosystem there are some criteria [27]:

- To avoid Pohlig-Hellman attack, n should be prime.
- To avoid baby-step giant-step and Pollard's attacks, in practice q should be of the order of 160 bits or more.
- To avoid Semaev-Smart-Araki attack, the number of rational points on $\mathbb{F}_q$ should not be equal to the prime number q, meaning that the curve should not be anomalous.
- To avoid Menezes-Okamoto-Vanstone (MOV) attack, the embedding degree k should be large, meaning that the curve should not be supersingular ($k \leq 6$).

In FIPS 186-4 [32] NIST recommends fifteen elliptic curves for use in ECC. The ANSI X9.62 [23] and X9.63 [22] provide specifications of elliptic curve cryptographic protocols.

Some of the cryptographic protocols based on elliptic curves are Elliptic-curve Diffie–Hellman, ElGamal, RSA and Elliptic Curve Digital Signature Algorithm (ECDSA).

3.1.1 Elliptic Curve Basics

To completely grasp zero knowledge-proofs it is important to understand how elliptic curves work. This section will summarize the mathematics involved as a reminder to the reader.

An elliptic curve C is a plane algebraic non-singular curve defined by the equation:

$$C : y^2 = x^3 + ax + b \tag{3.1}$$

To check algebraically if the curve is non-singular, that is the graph has no cusps, isolated points or self-intersections, we can calculate the discriminant defined as:

$$\Delta = -16(4a^3 + 27b^2) \tag{3.2}$$

And if the result is not equal to zero ($\Delta \neq 0$), that indicates that the curve does not have singular points.

A typical representation of an elliptic curve could be seen in Figure 3.1:

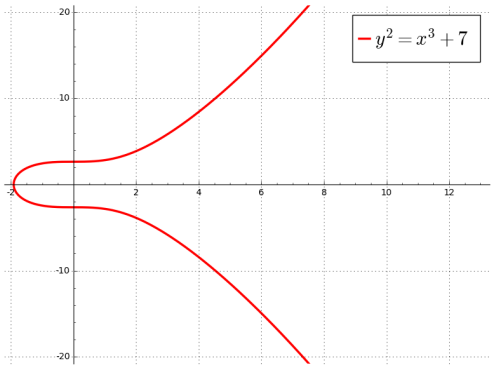


Figure 3.1: Elliptic curve $C : y^2 = x^3 + 7$ over $\mathbb{R}$

However, if the curve is defined over a finite field of prime order ($\mathbb{F}_p$) instead of over the real numbers (as the one in the Figure 3.1) then, it will appear a pattern of dots scattered in two dimensions, but despite the representation is different, all the maths are exactly the same. The same elliptic curve over a finite field of prime order 17 is shown in Figure 3.2.

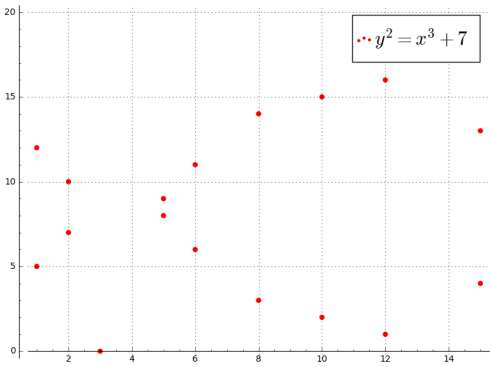


Figure 3.2: Elliptic curve $C : y^2 = x^3 + 7$ over $\mathbb{F}_{17}$

From Figure 3.2 we can guess that the curve will have more points while the number of elements in the field grows, and to know how many points are in the curve it is possible to use different algorithms, such as:

- Naive approach
- Baby-step giant-step
- Schoof's algorithm
- Schoof–Elkies–Atkin algorithm

This number of points is better known as the order of the elliptic curve.

Elliptic curve points are represented in a conventional way as two coordinates (x, y). All these points and the point at infinity (O) must satisfy the Equation 3.1 with the constants a and b.

Besides the point at infinity (O) there is also another point called *generator point*, commonly referenced by the letter G, which can be interpreted or considered as the number 1. All points in a prime order curve, excepting the point at infinity O can be a generator point G, but needs to be standardized for practical applications.

Mathematically it is possible to define the infinite group of points on an elliptic curve on the continuous plane (representation in $\mathbb{R}^2$) but facing an elliptic curve cryptosystem (ECC) will produce a *round-off* error, for that reason these systems use finite fields or Galois fields.

3.1.2 Arithmetic Operations

The most commonly operations for elliptic curve points will be defined below.

The point at infinity (O) is the neutral or identity element of elliptic curve arithmetic, which has the following properties:

- Adding any point P to O will result in: $P + O = P$.
- Adding it to a point $P = O$ will result in: $P + O = O + O = O$.
- Adding any point P and the negative of P will result in: $P + (-P) = O$.

Negating a point P on an elliptic curve over the real numbers and over a finite field is calculated as:

$$-P = -(x_p, y_p) = (x_p, -y_p) \tag{3.3}$$

Note that if the point is over a finite field modular arithmetic must be used.

Given the points P and Q where $P \neq Q$ (and $P \neq -Q; P, Q \neq O$), addition is defined as $R = P + Q$, graphically defined as the negation of the resulting point from intersecting a straight line defined by the points P and Q and the curve C.

$$P(x_p, y_p) + Q(x_q, y_q) = R(x_r, y_r) \tag{3.4}$$

The following formulas can be used to determine the resulting point of the addition:

$$\lambda = \frac{y_q - y_p}{x_q - x_p} \tag{3.5}$$

$$x_r = \lambda^2 - x_p - x_q \tag{3.6}$$

$$y_r = \lambda(x_p - x_r) - y_p \tag{3.7}$$

The addition is graphically shown in Figure 3.3 for the real case and in Figure 3.4 over the finite field $\mathbb{F}_{17}$:

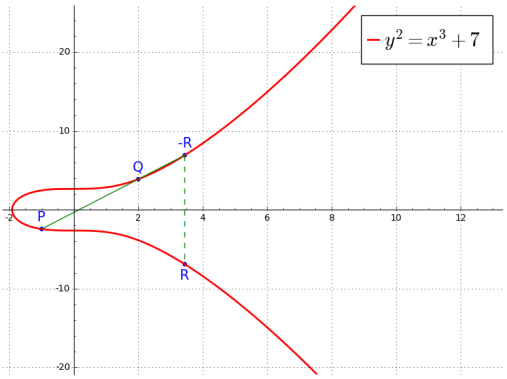


Figure 3.3: Addition of P and Q points over the real numbers $\mathbb{R}$

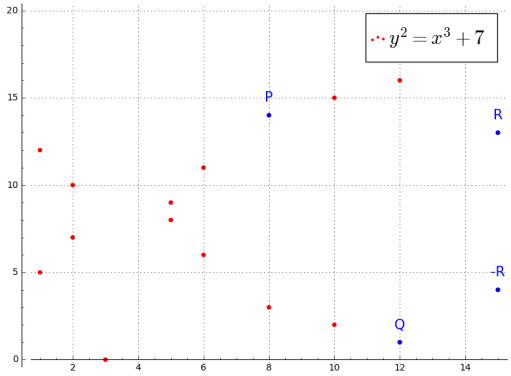


Figure 3.4: Addition of P and Q points over the finite field $\mathbb{F}_{17}$

Another possible case is when the points P and Q have the same coordinates, the addition is comparable but now there is not a line between two points because both are the same point ($Q = P$) so it is necessary to employ a limit case using the tangent to the curve C at this point.

This operation is called point doubling and viewing the curve C, if the point P has the coordinate $y_p = 0$, the doubling operation will result in the point at infinity O, as it is shown in Figure 3.5.

$$R = 2P(x_p, y_p) = 2P(x_p, 0) = O \tag{3.8}$$

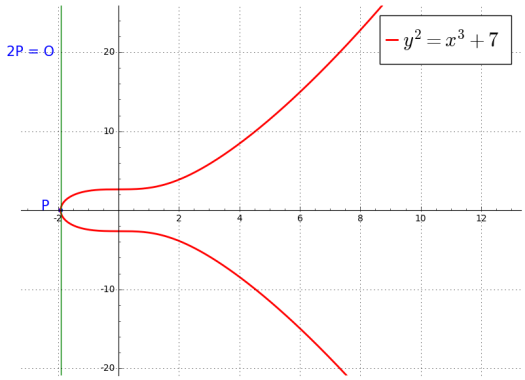


Figure 3.5: Doubling of P with coordinate $y_p = 0$

The doubling operation ($R = 2P$) is graphically shown in Figure 3.6 for the real case and in Figure 3.7 over the finite field $\mathbb{F}_{17}$:

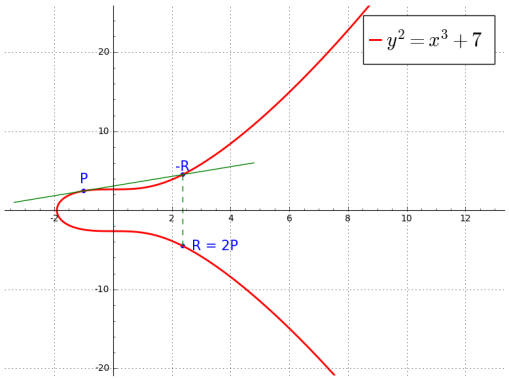


Figure 3.6: Doubling of P over the real numbers $\mathbb{R}$

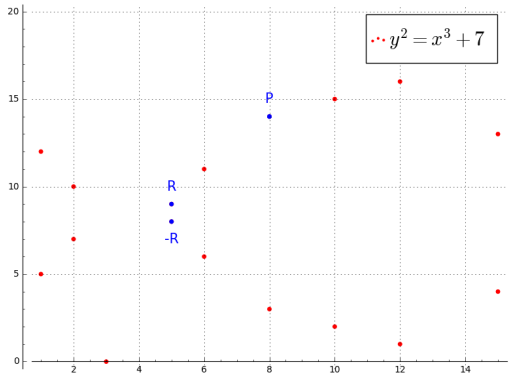


Figure 3.7: Doubling of P over the finite field $\mathbb{F}_{17}$

The calculation is similar to the addition, but changing the value of the slope (λ) with the following formula:

$$\lambda = \frac{3x_p^2 + a}{2y_p} \tag{3.9}$$

In the doubling operation like in negation and addition you must be careful if you are operating over a finite field and use modular arithmetic to calculate the resulting point.

Now that addition and doubling operations are defined, it is possible to define scalar multiplication extending these operations. Scalar multiplication is graphically shown in Figure 3.8 for the real case and in Figure 3.9 over the finite field $\mathbb{F}_{17}$:

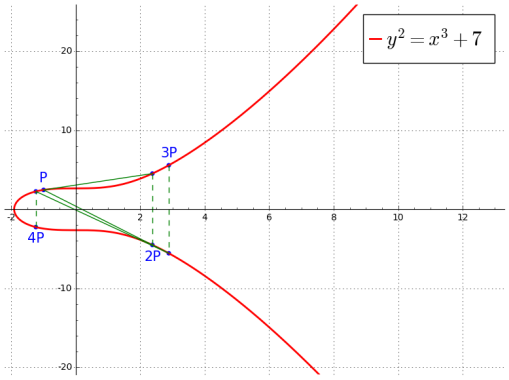


Figure 3.8: Scalar multiplication of P over the real numbers $\mathbb{R}$

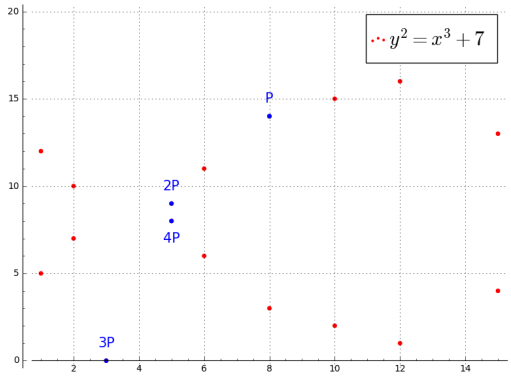


Figure 3.9: Scalar multiplication of P over the finite field $\mathbb{F}_{17}$

Note that if the point is over a finite field modular arithmetic must be used.

In fact, only addition it is necessary to define scalar multiplication for a point P on the elliptic curve C by repeating it k times (see Equation 3.10).

$$R = k \cdot P = \underbrace{P + P + P + ... + P}_{k \text{ times}} \tag{3.10}$$

Some of the algorithms used for scalar multiplication are: Double-and-add (Binary method), Addition Subtraction using NAF (*Non-Adjacent Form*), Windowed method or Montgomery ladder.

3.1.3 Elliptic Curve Pairings

Pairings are bilinear maps defined on pairs of points of an elliptic curve points taking values in a certain multiplicative Abelian group. Let $(M, +)$ and $(R, \cdot)$ be two Abelian groups, the following map $\langle \cdot, \cdot \rangle : M \times M \to R$ is called bilinear if the following conditions are satisfied $\forall x, y, z \in M$:

- $\langle x + y, z \rangle = \langle x, z \rangle \cdot \langle y, z \rangle$
- $\langle x, y + z \rangle = \langle x, y \rangle \cdot \langle x, z \rangle$

But it is possible, and advantageous, that the two arguments in the map not come from the same group and use cyclic groups of the same order. Thus, in cryptography, the commonly notation for bilinear pairings is:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{3.11}$$

That must satifies the bilinear property and two conditions, to be non-degenerate and efficiently computable.

Bilinearity:

$$e(P, Q + R) = e(P, Q) \cdot e(P, R), \qquad \forall P \in \mathbb{G}_1,\ \forall Q, R \in \mathbb{G}_2$$
$$e(P + Q, R) = e(P, R) \cdot e(Q, R), \qquad \forall P, Q \in \mathbb{G}_1,\ \forall R \in \mathbb{G}_2$$

Note that from the bilinearity of e is deducted that:

$$e([a]P, [b]Q) = e(P, [b]Q)^a = e([a]P, Q)^b = e(P, Q)^{ab} = e([b]P, [a]Q) \qquad \forall a, b \in \mathbb{Z}_p$$

Non-degeneracy:

$$\forall P \in \mathbb{G}_1, P \neq 0, \quad \exists Q \in \mathbb{G}_2 : e(P, Q) \neq 1$$
$$\forall Q \in \mathbb{G}_2, Q \neq 0, \quad \exists P \in \mathbb{G}_1 : e(P, Q) \neq 1$$

Efficient computability: There must exist an algorithm to compute pairings, otherwise they are only considered as a theoretical concept. For example, there must exists a polynomial time algorithm to compute e.

There are different types of pairings:

- *Symmetric pairings*: when $\mathbb{G}_1 = \mathbb{G}_2$.
- *Asymmetric pairings*: when $\mathbb{G}_1 \neq \mathbb{G}_2$ and it exists an efficiently computable group isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$.
- *Asymmetric pairings*: when $\mathbb{G}_1 \neq \mathbb{G}_2$ and it does not exist an efficiently computable group isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$.

There are different types of pairings such as the essentials like the Weil pairing and the Tate pairing, but there are other types such as the Eta, Ate and Omega pairings that have an auxiliar nature and can only be applied to certain curves.

To formally define a pairing it is important to introduce a brief definition from the theory of algebraic function fields that is essential to understand pairings, this is called theory of divisors of an elliptic curve.

Let C be an elliptic curve over a finite field $K = \mathbb{F}_q$, the group of divisors D of C is a set of formal sum of points on C:

$$D = \sum_{P \in C} n_P(P) \mid n_p \in \mathbb{Z}, n_p = 0 \text{ for all but finitely many} \tag{3.12}$$

Note that in this definition the sum notation used must not be confused with a formal sum of points on an elliptic curve (using the group law). In addition, points $P \in C$ must not be confused with divisors (P), which are a formal symbol. Thus, the set of all divisors forms an Abelian group and is denoted by $\mathrm{Div}_K(C)$.

Given two divisors D and D' defined as $D = \sum_{P \in C} n_p(P)$ and $D' = \sum_{P \in C} n'_p(P)$ the addition formula $D + D' = \sum_{P \in C}(n_p + n'_p)(P)$ vest an Abelian group structure to the set of divisors.

A divisor D has a degree defined as:

$$\mathrm{Deg}(D) = \sum_{P \in C} n_P \tag{3.13}$$

Also has a support denoted by:

$$\mathrm{supp}(D) = P \in C \mid n_P \neq 0 \tag{3.14}$$

In order not to distort the document and since it is not required to fully understand the theory of divisors to understand zk-SNARKs, the following references [29] [36] are left in case the reader wants to continue advancing in the learning of divisors.

Elliptic curve pairings are used in cryptography in two ways. The first one is with a destructive proposal, to design attacks for the discrete logarithm problem. Attacks based on pairings have as a consequence that supersingular curves are not considered secure to implement cryptosystems and cryptographic protocols based on the discrete logarithm, but on the other hand supersingular curves are appropriate for Identity-based cryptography. So, from a constructive point of view pairings constitute a basic tool of this new paradigm.

3.2 Homomorphic Hidings

In SNARK zero-knowledge proofs constructions, a *one-way deterministic function* known as Homomorphic Hiding is used. However, this term is not formally used in cryptography literature.

The most related concept within the cryptographic primitives could be the *Commitment Scheme*, which consists in committing one value (or statement) by Alice, who has the ability to reveal this value later to others, keeping it hidden.

This function $f(x) = HH(x)$ must satisfy the following properties:

- Easy to compute $f(x) = y$, but hard to find x from $f(x)$.
- Different inputs produce different outputs, $x \neq y \Rightarrow f(x) \neq f(y)$.
- Linearity, $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$.

The last property gives the name of "homomorphic" to this function. We can see an example of the homomorphic property with the exponential function.

If we define $HH(x) = e^x$ then, if $x = \alpha x_1 + \beta x_2$ then:

$$HH(x) = HH(\alpha x_1 + \beta x_2) = e^{\alpha x_1 + \beta x_2} = e^{\alpha x_1} \cdot e^{\beta x_2} = HH(x_1)^\alpha \cdot HH(x_2)^\beta$$

However, such naive definition does not provide an homomorphic hiding since as shown above, it is possible to calculate the logarithm base e of $HH(x)$ to get x, so the first property does not hold.

What we need here is a group $\mathbb{Z}_p^*$, so regardless of the inputs used, the result of the multiplication operation will be in the set $\{1, .., p-1\}$, where p is a large prime number. This group must satisfy the following properties:

- Be a cyclic group, having an element g in $\mathbb{Z}_p^*$ called generator of the group, which can express all the elements as g^k in the set for all integers $k \in \{0, ..., p-2\}$. The first element is $g^0 = 1$.
- Given an element h in $\mathbb{Z}_p^*$ it must be hard to find the integer k such that $g^k = h \pmod p$.
- Taking two elements of the set $a, b \in \{0, ..., p-2\}$, $g^a \cdot g^b = g^{(a+b) \bmod p-1}$.

Using these properties it is possible to define a function HH, such as $HH(x) = g^x$ that must satisfy the properties at the beginning of this point:

- From $HH(x) = g^x$ it is practically impossible to find x.
- Getting two elements $a, b \in \mathbb{Z}_{p-1}$, the result will be mapped to different outputs, such as $g^a \neq g^b$ but $g^a, g^b \in \{1, ..., p-1\}$.
- From $HH(x)$ and $HH(y)$ it is possible to compute $HH(x+y)$.

$$HH(x+y) = g^{(x+y) \bmod p-1} = g^x \cdot g^y = HH(x) \cdot HH(y)$$

With this, we can consider $HH(x)$ a one-way homomorphic hiding function.

CHAPTER 4

zk-SNARKs

In this chapter the zk-SNARKs will be analyzed starting from the definition of the zero-knowledge proofs from a theoretical point of view and then it will be shown how to construct a zk-SNARK proof.

4.1 Zero-Knowledge Proofs

A zero-knowledge proof or protocol (ZKP) is a cryptographic method defined as an interaction between two parties where one - the Prover - works to convince to another party - the Verifier - that some statement is true or some information is known without revealing the statement or the information to the Verifier, that it will not learn anything.

There are three properties that any zero-knowledge proof must satisfy:

- *Completeness*: If the Prover acting honestly sends a true input statement, the honest Verifier following the protocol correctly will be convinced of this statement. This property is often easy to check.
- *Soundness*: If the Prover tries to cheat sending a false statement, the Verifier will not be convinced that the statement is true, rejecting the proof with high probability.
- *Zero-knowledge(ness)*: If the statement sent by the Prover is true, the Verifier learns no information beyond the fact that the statement is true. So, the proof is zero-knowledge if whatever the Verifier learns, it could be learned without interacting with the Prover. The Verifier only needs to know the statement because using a polynomial time algorithm (called *simulator*), a transcription that seems the interaction between the honest Prover and the Verifier can be produced.

Let us see an example based on the Quadratic Residuosity protocol or QR protocol for proving quadratic residuosity. Remember that $x \in \mathbf{Z}_N^*$ is a quadratic residue mod N if there is some s such that $s^2 \equiv x \pmod{N}$. Without knowing the factorization of N, it is nearly impossible to tell whether x is a quadratic residue mod N.

One-Time Setup:

- A trusted third party T publishes $N = pq$, without revealing p and q.
- Prover selects a secret w coprime to N, such that $1 \leq w \leq N - 1$.
- Prover computes $v = w^2 \pmod{N}$.
- Prover registers v with T as its public key.

Protocol messages:

There are three messages for each t rounds, independently and sequentially. The Verifier only accepts the proof if all t rounds succeed.

$$\begin{array}{lcc} P \to V: & x = r^2 \pmod{N} & \text{Witness} \\ V \to P: & b \in \{0,1\} & \text{Challenge} \\ P \to V: & y = r \cdot w^b \pmod{N} & \text{Response} \end{array}$$

Protocol actions:

Iterated t times, independently and sequentially.

i. Prover chooses a random number $r \overset{\text{R}}{\leftarrow} \mathbf{Z}_N^*$, and send the witness $x = r^2 \pmod{N}$ to the Verifier.

ii. Verifier chooses a random number $b \overset{\text{R}}{\leftarrow} \{0,1\}$ and send the challenge b to the Prover. With this challenge requires that Prover be able to answer two questions: (i) to demostrate the knowledge of the secret w and (ii) to prevent cheating.

iii. Prover computes $y = r \cdot w^b \pmod{N}$:

   - If $b = 0$, Prover sends $y = r$.
   - If $b = 1$, Prover sends $y = r \cdot w \pmod{N}$.

The verification is done by the Verifier, if $y = 0$ the proofs is rejected, otherwise the proof is accepted if and only if the equation $y^2 \equiv x \cdot v^b \pmod{N}$ is true. To decrease probability of cheating it is recommended to use an acceptably value of t.

We denote by $r \overset{R}{\leftarrow} \mathbf{Z}_N^*$ the operation of selecting a random value from $\mathbf{Z}_N^*$ and naming it r.

To check the completeness, soundness and zero-knowledge(ness) properties defined above, we denote the Verifier's output after the interaction with the Prover on input v as $V_{P(v)}$.

- *Completeness*: Means that if v is a quadratic residue and the Prover and the Verifier follow the instructions of the algorithm, the Verifier will accept the proof with probability 1.

$$\Pr\left[\ V_{P(v)}(v) \rightarrow \text{accepts} \ \middle|\ \forall v \in \mathbf{Z}_N^*\ \right] = 1 \tag{4.1}$$

- *Soundness*: Means that if v is not a quadratic residue, then for every cheating Prover (P^*), the Verifier will reject the proof with probability at least $1/2$ for t iterations.

$$\Pr\left[\ V_{P^*(v)}(v) \rightarrow \text{accepts} \ \middle|\ \forall v \notin \mathbf{Z}_N^*\ \forall P^*\ \right] \leq \left(\frac{1}{2}\right)^t \tag{4.2}$$

- *Zero-knowledge(ness)*: In this case, a cheating Verifier (V^*) is very limited because can only send $b = 0$ or $b = 1$ and receives a random element in $\mathbf{Z}_N^*$, without learning no information about the quadratic residue v.

4.2 Succinct Non-interactive Arguments of Knowledge

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge is a novel zero-knowledge cryptographic method, that inherits the definition of being a proof with which one party - called Prover - can prove the knowledge of some information without revealing it to the second party - called Verifier.

Before focusing on other individual aspects of the proof, it is important to recall the definition of succinct non-interactive arguments (SNARGs) from Gentry and Wichs [15].

A SNARG system Π consists of three efficient machines $\Pi = (Gen, \mathcal{P}, \mathcal{V})$. The generation algorithm $(\texttt{crs}, \texttt{priv}) \leftarrow Gen(1^n)$ produces a common reference string $\texttt{crs}$ along with some private verification state $\texttt{priv}$. The prover algorithm $\pi \leftarrow \mathcal{P}(\texttt{crs}, x, w)$ produces a proof π for a statement x using a witness w. The verification algorithm $\{0,1\} \leftarrow \mathcal{V}(\texttt{priv}, x, \pi)$ decides if π is a valid proof for x, using the private verification state $\texttt{priv}$.

We say that $\Pi = (Gen, \mathcal{P}, \mathcal{V})$ is a succinct non-interactive argument for an $\mathcal{NP}$ language L with a corresponding $\mathcal{NP}$ relation R, if it satisfies the completeness, soundness and succinctness properties.

Completeness: $\forall (x,w) \in R$, $\Pr\left[\ \mathcal{V}(\texttt{priv}, x, \pi) = 0 \ \middle|\ \begin{array}{c}(\texttt{crs}, \texttt{priv}) \leftarrow Gen(1^n) \\ \pi \leftarrow \mathcal{P}(\texttt{crs}, x, w)\end{array}\ \right] = \text{negl}(n)$

Soundness: $\forall$ efficient $\bar{\mathcal{P}}$, $\Pr\left[\ \begin{array}{c}\mathcal{V}(\texttt{priv}, x, \pi) = 1 \\ \wedge\ x \notin L\end{array}\ \middle|\ \begin{array}{c}(\texttt{crs}, \texttt{priv}) \leftarrow Gen(1^n) \\ (x, \pi) \leftarrow \bar{\mathcal{P}}(1^n, \texttt{crs})\end{array}\ \right] = \text{negl}(n)$

Succinctness: The length of a proof is given by $|\pi| = \text{poly}(n)(|x| + |w|)^{o(1)}$.

Note that a function f is called negligible if $f(n) = \frac{1}{n^{w(1)}}$ denoted by $\text{negl}(n)$ where n refers to a security parameter and a simulate adversary is denoted by $\bar{\mathcal{P}}$. The notation $\text{poly}(n)$ is used to identify efficient algorithms [15].

From SNARGs, a definition of SNARKs was formalized in [14].

A succinct non-interactive argument of knowledge (SNARK) is a SNARG that comes together with an extractor $\mathcal{E}$. In particular, for any statement x, we require that there be a polynomial-time extractor $\mathcal{E}_x$ such that, for any $\pi \leftarrow \mathcal{P}(\texttt{crs}, x, w)$, $w \leftarrow \mathcal{E}_x(\texttt{priv}, \pi)$.

A SNARK for an $\mathcal{NP}$ language L with a corresponding $\mathcal{NP}$ relation R is zero-knowledge, if there exists a simulator (S_1, S_2) such that S_1 outputs a simulated `crs` and a trapdoor τ, $(\texttt{crs}, \tau) \leftarrow S_1(1^n, R)$, and S_2 takes as input `crs`, a statement x and τ and outputs a simulated proof π, $\pi \leftarrow S_2(\texttt{crs}, x, \tau)$.

Hence, the formal definition of a SNARK is provided, but we need to define the meaning of each of the individual aspects of the acronym zk-SNARK:

- **z**ero-**k**nowledge: As previously defined in Section 4.1, being a zero-knowledge proof, the Verifier will not learn anything during the interaction with the Prover.
- **S**uccinct: The length of the proof π for a variable statement will only have a few hundred bytes even for complex computations, so the verification could be done in milliseconds.
- **N**on-interactive: For zk-SNARKs there is no interaction between the Prover and the Verifier, instead there is a setup phase and the proof consists of a single message from the Prover to the Verifier, allowing the arguments to be verified by all parties without interacting again, that is the *public verifier* property of SNARKs. Formally, we say that a SNARK is *publicly verifiable* if the private verification state is $\texttt{priv} = \texttt{crs}$.
- **AR**guments: A proof with perfect completeness and perfect soundness is called an argument and protects the Verifier against malicious Provers. Creating incorrect proofs or arguments is computationally hard in contrast to create correct proofs or arguments, which is computationally feasible.
- of **K**nowledge: It is impossible for the Prover to construct arguments or proofs without knowing the witness, there must be one or more inputs that will be used in the verification process.

Let C be an arithmetic circuit such that $C : \mathbb{F}^n \times \mathbb{F}^{n'} \rightarrow \mathbb{F}^l$ and an $\mathcal{NP}$ language L which consists of statements with valid witnesses defined as $L := \{x \in \mathbb{F}^n : \exists\ w \in \mathbb{F}^{n'}\}$ for relation $R := \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^{n'}\}$ where w is the witness and x the statement. A non-interactive zero-knowledge argument for the relation R consists of the triple of polynomial time algorithms: Generation (Gen), Prover ($\mathcal{P}$) and Verification ($\mathcal{V}$).

- A security parameter n and a relation R are taken by Gen that outputs a `crs`: $(\texttt{crs}) \leftarrow Gen(1^n, R)$.
- Prover $\mathcal{P}$ outputs an argument or proof π using the `crs` for the relation R, a statement x and a witness w as inputs $(x, w) \in R$: $\pi \leftarrow \mathcal{P}(\texttt{crs}, x, w)$.
- Verification $\mathcal{V}$ accepts or rejects the proof or argument π using the `crs`, a statement x and the proof π as inputs: $\{0, 1\} \leftarrow \mathcal{V}(\texttt{crs}, x, \pi)$.

A non-interactive zero-knowledge proof or argument π for the relation R must satisfy the following properties [14] listed in Section 4.1:

- *Completeness*: The Prover acting honestly always provide a valid proof or argument π for a statement $x \in \mathbb{F}^n$ with a witness $w \in \mathbb{F}^{n'}$ with $(x, w) \in R$ and should be able to convince an honest Verifier.

$$\Pr\left[\begin{array}{c|c} (\mathtt{crs}) \leftarrow Gen(1^n, R); & \\ \pi \leftarrow \mathcal{P}(\mathtt{crs}, x, w) & \mathcal{V}(\mathtt{crs}, x, \pi) = 1 \text{ if } (x, w) \in R \end{array}\right] = 1 \tag{4.3}$$

- *Soundness*: If an adversary tries to cheat giving a proof or argument π for a false statement $x \notin R$ the verification algorithm $\mathcal{V}$ will reject the proof or argument with high probability. So, for an adversary $\mathcal{A}$ the following holds:

$$\Pr\left[\begin{array}{c|c} (\mathtt{crs}) \leftarrow Gen(1^n, R); & \\ (x, \pi) \leftarrow \mathcal{A}(\mathtt{crs}) & \mathcal{V}(\mathtt{crs}, x, \pi) = 1 \text{ and } (x, w) \notin R \end{array}\right] \leq \text{negl}(n) \tag{4.4}$$

  And, if there exist and extractor $\mathcal{E}$ such that computes the witness $w \leftarrow \mathcal{E}_{\mathcal{A}}(\mathtt{crs})$ from and an adversary $\mathcal{A}$ that produces a valid argument $(x, \pi) \leftarrow \mathcal{A}(\mathtt{crs})$, the following holds:

$$\Pr\left[\begin{array}{c|c} (\mathtt{crs}) \leftarrow Gen(1^n, R); & \\ (x, \pi) \leftarrow \mathcal{A}(\mathtt{crs}) & \mathcal{V}(\mathtt{crs}, x, \pi) = 1 \text{ and } (x, w) \notin R \\ w \leftarrow \mathcal{E}_{\mathcal{A}}(\mathtt{crs}) & \end{array}\right] \leq \text{negl}(n) \tag{4.5}$$

- *Zero-knowledge(ness)*: This property guarantees that the Verifier learns no information beyond the fact that the statement is true. For this purpose is used a polynomial time algorithm called simulator to simulate the proof or argument π for a valid statement x without knowing the witness w. From the zero-knowledge SNARK definition commented above: there exists a simulator (S_1, S_2) such that S_1 outputs a simulated $\mathtt{crs}$ and a trapdoor τ, $(\mathtt{crs}, \tau) \leftarrow S_1(1^n, R)$, and S_2 takes as input $\mathtt{crs}$, a statement x and τ and outputs a simulated proof π, $\pi \leftarrow S_2(\mathtt{crs}, x, \tau)$. For all adversaries $\mathcal{A}$ it holds that:

$$\Pr\left[\begin{array}{c|c} (\mathtt{crs}) \leftarrow Gen(1^n, R) & \\ (x, w) \leftarrow \mathcal{A}(\mathtt{crs}) & (x, w) \in R \\ \pi \leftarrow \mathcal{P}(\mathtt{crs}, x, w) & \mathcal{A}(\pi) = 1 \end{array}\right] = \Pr\left[\begin{array}{c|c} (\mathtt{crs}, \tau) \leftarrow \mathcal{S}_1(1^n, R) & \\ (x, w) \leftarrow \mathcal{A}(\mathtt{crs}) & (x, w) \in R \\ \pi \leftarrow \mathcal{S}_2(\mathtt{crs}, x, \tau) & \mathcal{A}(\pi) = 1 \end{array}\right]$$

Note that this and the following sections are based on the protocol defined in [14], but it is possible to find different definitions of the polynomial time algorithms depending on the protocol used. For example, in [10] the protocol is based on a quadruple of efficient algorithms (Setup, Prove, Verify and Simulation), and, as a result, it is an improvement in terms of size and verification cost compare with the Pinocchio protocol [33]. However, in order to avoid adding complexity to the explanation, a simpler but less efficient protocol has been considered to show how zk-SNARKs work.

In next sections we will cover the SNARK construction from arithmetic circuits and quadratic arithmetic programs. For more information about security or optimization of the verification algorithm it is highly recommended to read chapters four and five in [14].

4.3 Quadratic Arithmetic Programs for Arithmetic Circuits

As previously commented in the State of Art, Section 2, Gennaro, Gentry, Parno and Raykova showed how to encode computations as quadratic programs [14]. They show how to transform any boolean circuit into a Quadratic Span Program (QSP) and any arithmetic circuit into a Quadratic Arithmetic Program (QAP).

Quadratic span program (QSP) is a new characterization of the NP complexity class that allows to construct efficient zk-SNARK proofs without using probabilistically checkable proofs (PCPs), but with the disadvantage that they can only compute boolean circuits, which sometimes is more inefficient and unnatural than arithmetic circuits. Quadratic arithmetic programs (QAPs) are analogous to QSPs but they can compute arithmetic circuits using three sets of polynomials while QSPs only require two sets. In many cases, QAPs are more efficient for the Prover because she only computes cryptographic operations for each mod-p gate and not for each boolean gate [14].

Boolean circuits operate over bits with boolean gates such as AND, OR, XOR, etc. Arithmetic circuits, in contrast, operate values using addition and multiplication gates. Arithmetic circuits, which can be represented in the same way as boolean circuits, are directed acyclic graphs with wires as edges and gates as vertices.

Let $\mathbb{F}$ be a finite field and C a map $\mathbb{F}^n \times \mathbb{F}^{n'} \to \mathbb{F}^l$ that takes $n+n'$ elements as inputs and l element as outputs in $\mathbb{F}$. The map C is an arithmetic circuit if the input elements that pass through wires to gates are manipulated according to the circuit operations (additive, multiplicative or constant gates) determine the output elements.

A tuple $(a_1, ..., a_N) \in \mathbb{F}^N$ where $N = (n + n') + l$ is the number of all inputs and outputs of C is a valid assignment for an arithmetic circuit C if $C(a_1, ..., a_{n+n'}) = (a_{n+n'+1}, ..., a_N)$.

As an example, suppose a map C, such that $C : \mathbb{F}^2 \times \mathbb{F}^2 \to \mathbb{F}^2$ is defined by the following expression:

$$C(a_1, a_2, a_3, a_4) = \big((a_1 + a_2) \cdot (a_2 - a_3), (a_2 - a_3) \cdot (a_4)\big)$$

In the Figure 4.1 the arithmetic circuit C based on the circuit proposed in [14] is shown where the output value of a gate is expressed in terms of the input values of higher gates.

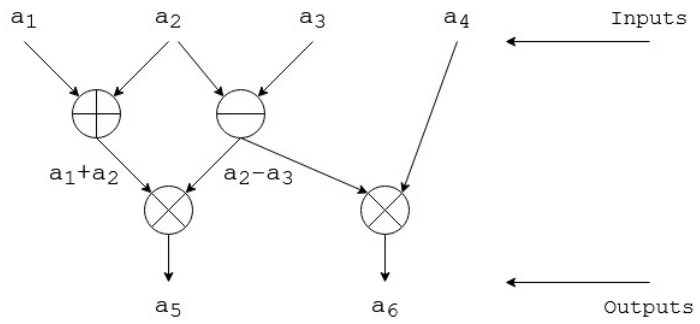


Figure 4.1: Arithmetic circuit example

QAPs are used to make it easier for the Prover to construct a proof π to claim that she knows a valid statement $(a_1, ..., a_N) \in \mathbb{F}^N$ for an arithmetic circuit C.

A Quadratic Span Program Q over field $\mathbb{F}$ contains three sets of polynomials: $\mathcal{V} = \{v_k(x) : k \in \{0, ..., m\}\}$, $\mathcal{W} = \{w_k(x) : k \in \{0, ..., m\}\}$, $\mathcal{Y} = \{y_k(x) : k \in \{0, ..., m\}\}$ and a target polynomial $t(x)$. The positive integer m is the number of inputs into the circuit plus the number of multiplication gates.

Suppose a function f which takes as input $n + n'$ elements of $\mathbb{F}$ and outputs l making a total of $N = n + n' + l$ input/output elements. We say that Q is a QAP that computes f if: $(a_1, ..., a_N) \in \mathbb{F}^N$ is a valid assignment to the input/output variables of f, if and only if there exist $(a_{N+1}, ..., a_m)$ such that $t(x)$ divides $p(x)$ defined as:

$$p(x) = \left(v_0(x) + \sum_{k=1}^{m} a_k \cdot v_k(x)\right) \cdot \left(w_0(x) + \sum_{k=1}^{m} a_k \cdot w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} a_k \cdot y_k(x)\right) \quad (4.6)$$

The above can be interpreted as there must exist a polynomial $h(x)$ such that $h(x) \cdot t(x) = p(x)$.

The size of Q is m, and the degree is defined as $\deg(t(x))$.

As shown in Figure 4.1, input values of a multiplication gate are linear functions of the higher values in the circuit. The equation for a gate g could be formally defined as:

$$a_g = \left(\sum_{k \in \mathcal{I}_{g,L}} a_k \cdot c_{g,L,k}\right) \cdot \left(\sum_{k \in \mathcal{I}_{g,R}} a_k \cdot c_{g,R,k}\right) \quad (4.7)$$

Where a_g refers to the output value of a multiplication gate g, a_k refers to the input value of the gate g which can be the output of a higher multiplication gate or just an input value. $\mathcal{I}_{g,L}$ refers to the subset of values that are outputs of multiplication gates or input values to the circuit that are indirect left inputs to gate g, and analogously $\mathcal{I}_{g,R}$ refers to the subset of indirect right inputs to gate g. Lastly $c_{g,L,k}$ and $c_{c,R,k}$ refer to the scalar with which the input enters gate g from the left and from the right respectively.

To define the target polynomial $t(x)$, let $\mathcal{M}$ be the index of s multiplication gates of the circuit, the roots $\{r_i \in \mathbb{F} : i \in \mathcal{M}\}$ associated with each multiplication gate $g \in \mathcal{M}$ as r_g define the target polynomial as:

$$t(x) = \prod_{g \in \mathcal{M}} (x - r_g) \quad (4.8)$$

Then, the set of polynomials $\mathcal{V}, \mathcal{W}$ known as input polynomials and $\mathcal{Y}$ defined as output polynomial are formed as follows.

$$v_k(r_g) = \begin{cases} c_{g,L,k} & \forall k \in \mathcal{I}_{g,L} \\ 0 & \text{otherwise} \end{cases} \quad (4.9)$$

$$w_k(r_g) = \begin{cases} c_{g,R,k} & \forall k \in \mathcal{I}_{g,R} \\ 0 & \text{otherwise} \end{cases} \quad (4.10)$$

And from the equations 4.9 and 4.10:

$$v(r_g) = v_0(r_g) + \sum_{k=1}^{m} a_k \cdot v_k(r_g) = v_0(r_g) + \sum_{k \in \mathcal{I}_{g,L}} a_k \cdot c_{g,L,k} \quad (4.11)$$

$$w(r_g) = w_0(r_g) + \sum_{k=1}^{m} a_k \cdot v_k(r_g) = w_0(r_g) + \sum_{k \in \mathcal{I}_{g,R}} a_k \cdot c_{g,R,k} \tag{4.12}$$

And, the output polynomial is defined as:

$$y_k(r_g) = \begin{cases} 1 & \forall k \in \mathcal{M} \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

And from the Equation 4.13:

$$y(r_g) = y_0(r_g) + \sum_{k=1}^{m} a_k \cdot y_k(r_g) = a_g \tag{4.14}$$

It is possible to conclude from the above that if and only if $(a_1, ..., a_N)$ is a valid assignment, then:

$$\left(\sum_{k=1}^{m} a_k \cdot v_k(r_g)\right) \cdot \left(\sum_{k=1}^{m} a_k \cdot w_k(r_g)\right) - \left(\sum_{k=1}^{m} a_k \cdot y_k(r_g)\right) = 0 \qquad \forall g \tag{4.15}$$

In order to illustrate all the process we will use the circuit in Figure 4.1 to construct the QAP.

First, $\mathcal{M} = \{5, 6\}$ and the index $k \in [m] = \{1, ..., m\}$ is associated to each input of the circuit and each output from a multiplication gate, so $k \in [6]$.

The roots r_5 and r_6 are associated with a multiplication gate respectively so the target polynomial is defined as:

$$t(x) = \prod_{g \in \mathcal{M}} (x - r_g) = (x - r_5) \cdot (x - r_6) \qquad \forall \{r_5, r_6\} \in \mathbb{F} : r_5 \neq r_6$$

Since there are two roots, the polynomials $\mathcal{V}, \mathcal{W}$ and $\mathcal{Y}$ will be expressed as pairs $(\cdot, \cdot)$ referencing their evaluations at r_5 and r_6.

Left polynomials $v_k(x)$ evaluated at r_5 and r_6, i.e. inputs entering gates 5 and 6 from the left.

$v_k(r_g)$	Description	$v_k(r_5, r_6)$
$v_0(r_5) = 0$ $v_0(r_6) = 0$	Because there is not input constant in gate 5 Because there is not input constant in gate 6	$v_0(r_5, r_6) = (0, 0)$
$v_1(r_5) = 1$ $v_1(r_6) = 0$	Because the input a_1 enters gate 5 Because the input a_1 does not enter gate 6	$v_1(r_5, r_6) = (1, 0)$
$v_2(r_5) = 1$ $v_2(r_6) = 1$	Because the input a_2 enters gate 5 Because the input a_2 enters gate 6	$v_2(r_5, r_6) = (1, 1)$
$v_3(r_5) = 0$ $v_3(r_6) = -1$	Because the input a_3 does not enter gate 5 Because the input a_3 enters gate 6 with factor -1	$v_3(r_5, r_6) = (0, -1)$
$v_4(r_5) = 0$ $v_4(r_6) = 0$	Because the input a_4 does not enter gate 5 Because the input a_4 does not enter gate 6	$v_4(r_5, r_6) = (0, 0)$
$v_5(r_5) = 0$ $v_5(r_6) = 0$	Because inputs a_5 and a_6 are not inputs to any gate	$v_5(r_5, r_6) = (0, 0)$
$v_6(r_5) = 0$ $v_6(r_6) = 0$	Because inputs a_5 and a_6 are not inputs to any gate	$v_6(r_5, r_6) = (0, 0)$

$$v(r_5) = \sum_{k=1}^{6} a_k \cdot v_k(r_5) = a_1 + a_2 \qquad v(r_6) = \sum_{k=1}^{6} a_k \cdot v_k(r_6) = a_2 - a_3$$

Right polynomials $v_k(x)$ evaluated at r_5 and r_6, i.e. inputs entering gates 5 and 6 from the right.

$w_k(r_g)$	Description	$w_k(r_5, r_6)$
$w_0(r_5) = 0$ $w_0(r_6) = 0$	Because there is not input constant in gate 5 Because there is not input constant in gate 6	$w_0(r_5, r_6) = (0, 0)$
$w_1(r_5) = 0$ $w_1(r_6) = 0$	Because the input a_1 does not enter gate 5 Because the input a_1 does not enter gate 6	$w_1(r_5, r_6) = (0, 0)$
$w_2(r_5) = 1$ $w_2(r_6) = 0$	Because the input a_2 enters gate 5 Because the input a_2 does not enter gate 6	$w_2(r_5, r_6) = (1, 0)$
$w_3(r_5) = -1$ $w_3(r_6) = 0$	Because the input a_3 enters gate 5 with factor -1 Because the input a_3 does not enter gate 6	$w_3(r_5, r_6) = (-1, 0)$
$w_4(r_5) = 0$ $w_4(r_6) = 1$	Because the input a_4 does not enter gate 5 Because the input a_4 enters gate 6	$v_4(r_5, r_6) = (0, 1)$
$w_5(r_5) = 0$ $w_5(r_6) = 0$	Because inputs a_5 and a_6 are not inputs to any gate	$w_5(r_5, r_6) = (0, 0)$
$w_6(r_5) = 0$ $w_6(r_6) = 0$	Because inputs a_5 and a_6 are not inputs to any gate	$w_6(r_5, r_6) = (0, 0)$

$$w(r_5) = \sum_{k=1}^{6} a_k \cdot w_k(r_5) = a_2 - a_3 \qquad w(r_6) = \sum_{k=1}^{6} a_k \cdot w_k(r_6) = a_4$$

And finally, the output polynomial $y_k(x)$ evaluated at r_5 and r_6.

$y_k(r_g)$	Description	$y_k(r_5, r_6)$
$y_0(r_5) = 0$ $y_0(r_6) = 0$	Because correspond to an input value of the circuit	$y_0(r_5, r_6) = (0, 0)$
$y_1(r_5) = 0$ $y_1(r_6) = 0$	Because correspond to an input value of the circuit	$y_1(r_5, r_6) = (0, 0)$
$y_2(r_5) = 0$ $y_2(r_6) = 0$	Because correspond to an input value of the circuit	$y_2(r_5, r_6) = (0, 0)$
$y_3(r_5) = 0$ $y_3(r_6) = 0$	Because correspond to an input value of the circuit	$y_3(r_5, r_6) = (0, 0)$
$y_4(r_5) = 0$ $y_4(r_6) = 0$	Because correspond to an input value of the circuit	$y_4(r_5, r_6) = (0, 0)$
$y_5(r_5) = 1$ $y_5(r_6) = 0$	Because correspond to an output value of gate 5	$y_5(r_5, r_6) = (1, 0)$
$y_6(r_5) = 0$ $y_6(r_6) = 1$	Because correspond to an output value of gate 6	$y_6(r_5, r_6) = (0, 1)$

$$y(r_5) = \sum_{k=1}^{6} a_k \cdot y_k(r_5) = a_5 \qquad y(r_6) = \sum_{k=1}^{6} a_k \cdot y_k(r_6) = a_6$$

Concluding the process we have the following polynomials:

$$\begin{aligned} &\text{Gate 5:} \quad p(r_5) = v(r_5) \cdot w(r_5) - y(r_5) = (a_1 + a_2) \cdot (a_2 - a_3) - a_5 \\ &\text{Gate 6:} \quad p(r_6) = v(r_6) \cdot w(r_6) - y(r_6) = (a_2 - a_3) \cdot (a_4) - a_6 \end{aligned}$$

Thus, from the definition of QAP in Equation 4.6 and from the condition in Equation 4.15, $t(x)$ divides $p(x)$ if and only if $a_5 = (a_1 + a_2) \cdot (a_2 - a_3)$ and $a_6 = (a_2 - a_3) \cdot a_4$ or in other words if and only if $(a_1, ..., a_6)$ is a valid assignment for the arithmetic circuit C.

An example of a valid assignment could be $(0, 1, 1, 1, 0, 0) \in \mathbb{F}^6$ or $(0, 1, 1, 0, 0, 0) \in \mathbb{F}^6$, in addition to many others.

4.4 From Quadratic Arithmetic Programs to zk-SNARKs

Based on the triple of polynomial time algorithms: Generation (Gen), Prover ($\mathcal{P}$) and Verification ($\mathcal{V}$) presented in Section 4.2, now it is possible to define a SNARK system Π such as $\Pi = (Gen, \mathcal{P}, \mathcal{V})$ for Quadratic Arithmetic Programs (QAPs).

Let R be a relation $R := \{(u, w) \in \mathbb{F}^n \times \mathbb{F}^{n'}\}$ where $w \in \mathbb{F}^{n'}$ is the witness and $u \in \mathbb{F}^n$ the statement. This relation R can be verified with an arithmetic circuit C, so it would be an arithmetic function f such that $f(u, w) = 1$ if and only if $(u, w) \in R$, meaning that an arithmetic circuit output is 1 if the input satisfies the arithmetic function f, and 0 otherwise.

$$f(u, w) = \begin{cases} 1 & \forall (u, w) \in R \\ 0 & \text{otherwise} \end{cases}$$

Before starting with the triple of polynomial time algorithms, the SNARK protocol in [14] uses bilinear cryptography concepts explained in Section 3.1.3 and Section 3.2. First, we fix a group $\mathbb{G}$ of order r and a generator g, which is called generator if every element of the group can be written as a power of g, such that $g^0, g^1, ..., g^{r-1}$. From Section 3.2 the encryption is $HH(x) := g^x$.

Let us start with the Gen algorithm, which is in charge of the CRS generation. Using the relation R and the function f from $\mathbb{F}$, from an arithmetic circuit C we construct a QAP Q such that $Q = (\mathcal{V}, \mathcal{W}, \mathcal{Y}, t(x))$ of size m and degree $deg(t(x)) = d$. Since m is relatively large, there are some numbers that do not appear in the output of f for any input, also known as non-input/output-related indices, but these indices are not restricted so let us call them $\mathcal{I}_{mid} = \{N + 1, ..., m\}$ where N is the number of inputs/outputs of the functon f. Then, let e be a bilinear and non-degenerate map such that $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$.

At this point, the Verifier chooses the following secret numbers: $\alpha, s, \beta_v, \beta_w, \beta_y, \gamma \xleftarrow{R} \mathbb{F}$. These numbers are also known as *toxic waste* or the randomness source that the generator of the CRS must remove at all costs.

To simplify the notation, in the details provided from now on, since we use the function $HH(x) := g^x$, the reference function $HH(x)$ will not be provided.

At the beginning, the verifier selects α and s and constructs the following:

$$\{g^{s^i}\}_{i\in[d]} \text{ and } \{g^{\alpha s^i}\}_{i\in[d]}$$

Given the polynomials from the QAP ($\mathcal{V}$,$\mathcal{W}$, $\mathcal{Y}$ and $t(x)$), they are evaluated and published.

$$\begin{array}{llll} g^{t(s)}, & \{g^{v_k(s)}\}_{k\in\mathcal{I}_{mid}}, & \{g^{w_k(s)}\}_{k\in[m]}, & \{g^{y_k(s)}\}_{k\in[m]}, \\ g^{\alpha t(s)}, & \{g^{\alpha v_k(s)}\}_{k\in\mathcal{I}_{mid}}, & \{g^{\alpha w_k(s)}\}_{k\in[m]}, & \{g^{\alpha y_k(s)}\}_{k\in[m]} \end{array}$$

An additional set of secret numbers $(\beta_v, \beta_w, \beta_y, \gamma)$ will be used to verify that those polynomials were evaluated:

$$\begin{array}{llll} g^{\beta_v t(s)}, & g^{\beta_w t(s)}, & g^{\beta_y t(s)}, & \\ \{g^{\beta_v v_k(s)}\}_{k\in\mathcal{I}_{mid}}, & \{g^{\beta_w w_k(s)}\}_{k\in[m]}, & \{g^{\beta_y y_k(s)}\}_{k\in[m]}, & \\ g^{\beta_v\gamma}, & g^{\beta_w\gamma}, & g^{\beta_y\gamma}, & g^{\gamma} \end{array}$$

All these computations are considered as the full common reference string (CRS), but in practical implementations only some elements are used to construct the CRS.

Since in this chapter the explanation is based on the protocol in [14], from the computations above the Verifier computes the public evaluation key EK (also known as CRS or proving key) and the public verification key VK (also known as short CRS):

Evaluation key:

$$\begin{aligned} EK = \big(&\{g^{s^i}\}_{i\in[d]}, \{g^{\alpha s^i}\}_{i\in[d]}, \{g^{v_k(s)}\}_{k\in\mathcal{I}_{mid}}, \{g^{\alpha v_k(s)}\}_{k\in\mathcal{I}_{mid}}, \{g^{\beta_v v_k(s)}\}_{k\in\mathcal{I}_{mid}}, \{g^{w_k(s)}\}_{k\in[m]}, \\ &\{g^{\alpha w_k(s)}\}_{k\in[m]}, \{g^{\beta_w w_k(s)}\}_{k\in[m]}, \{g^{y_k(s)}\}_{k\in[m]}, \{g^{\alpha y_k(s)}\}_{k\in[m]}, \{g^{\beta_y y_k(s)}\}_{k\in[m]}\big) \end{aligned}$$

Verification key:

$$VK = \big(g^1, g^\alpha, g^\gamma, g^{\beta_v\gamma}, g^{\beta_w\gamma}, g^{\beta_y\gamma}, g^{t(s)}, \{g^{v_k(s)}\}_{k\in[N]}, g^{v_0(s)}, g^{w_0(s)}, g^{y_0(s)}\big)$$

These evaluation and verification keys change depending on the protocol as well as the secret numbers selected, as happens in the Pinocchio protocol [33].

Continuing with the algorithms, now the Prover generates a proof from a statement u and a witness w such that $(u, w) \in R \to f(u, w) = 1$ using the public evaluation key (also known as CRS or proving key). The Prover evaluates the QAP Q to obtain $\{a_i\}_{i\in[m]}$ such that:

$$h(x)\cdot t(x) = \left(v_0(x) + \sum_{k=1}^{m} a_k \cdot v_k(x)\right) \cdot \left(w_0(x) + \sum_{k=1}^{m} a_k \cdot w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} a_k \cdot y_k(x)\right)$$

The Prover computes $h(x)$ which is the polynomial such that $h(x)\cdot t(x) = p(x)$ as we see in Section 4.3. Do not forget to take into account the non-input/output-related indices $\mathcal{I}_{mid}$ as before and define $v_{mid}(x) = \sum_{k\in\mathcal{I}_{mid}} a_k \cdot v_k(x)$, $w(x) = \sum_{k\in[m]} a_k \cdot w_k(x)$ and $y(x) = \sum_{k\in[m]} a_k \cdot y_k(x)$.

Thus, the Prover generates the following proof π:

$$\pi = (\underbrace{g^{v_{mid}(s)}}_{\pi_{v_{mid}}}, \underbrace{g^{w(s)}}_{\pi_w}, \underbrace{g^{y(s)}}_{\pi_y}, \underbrace{g^{h(s)}}_{\pi_h}, \underbrace{g^{\alpha v_{mid}(s)}}_{\pi_{v'_{mid}}}, \underbrace{g^{\alpha w(s)}}_{\pi_{w'}}, \underbrace{g^{\alpha y(s)}}_{\pi_{y'}}, \underbrace{g^{\alpha h(s)}}_{\pi_{h'}}, \underbrace{g^{\beta_v v_{mid}(s)+\beta_w w(s)+\beta_y y(s)}}_{\pi_{\beta'}})$$

Note that the Prover can generate all the values knowing the public evaluation key or CRS.

Finally, the Verification algorithm takes as inputs the public verification key or short CRS, the statement u and the proof π. From the statement u the Verifier can compute the missing part of the full sum for $v(s)$, that is: $v_{in}(s) = \sum_{k\in[N]} a_k \cdot v_k(s)$.

Using the pairing function e and the verification key, the Verifier checks the proof π submitted by the Prover.

Extractability check: These four checks verify that the Prover did evaluate a constructed polynomial from the public evaluation key or CRS. The following equalities are immediate.

$$\begin{aligned} &e(\pi_{v_{mid}}, g^\alpha) = e(\pi_{v'_{mid}}, g) \quad && e(\pi_w, g^\alpha) = e(\pi_{w'}, g) \\ &e(\pi_y, g^\alpha) = e(\pi_{y'}, g) && e(\pi_h, g^\alpha) = e(\pi_{h'}, g) \end{aligned}$$

Linear span check: This check verifies that the Prover did not use an arbitrary sets of polynomials instead of the correct ones: $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$.

$$e(\pi_\beta, g^\gamma) = e(g^{\beta_v\gamma}, \pi_{v_{mid}}) \cdot e(g^{\beta_w\gamma}, \pi_w) \cdot e(g^{\beta_y\gamma}, \pi_y)$$

From the following expression, we can see the correctness of such equation when all values are correctly computed:

$$\begin{aligned} e(\pi_\beta, g^\gamma) &= e(g^{\beta_v v_{mid}(s)+\beta_w w(s)+\beta_y y(s)}, g^\gamma) = \\ &= e(g,g)^{\gamma(\beta_v v_{mid}(s)+\beta_w w(s)+\beta_y y(s))} = \\ &= e(g,g)^{\beta_v\gamma v_{mid}(s)} \cdot e(g,g)^{\beta_w\gamma w(s)} \cdot e(g,g)^{\beta_y\gamma y(s)} = \\ &= e(g^{\beta_v\gamma}, g^{v_{mid}(s)}) \cdot e(g^{\beta_w\gamma}, g^{w(s)}) \cdot e(g^{\beta_y\gamma}, g^{y(s)}) = \\ &= e(g^{\beta_v\gamma}, \pi_{v_{mid}}) \cdot e(g^{\beta_w\gamma}, \pi_w) \cdot e(g^{\beta_y\gamma}, \pi_y) \end{aligned}$$

Divisibility check: This check verifies the main condition for the Quadratic Arithmetic Problem (see Equation 4.6).

$$e(\pi_h, g^{t(s)}) = e(g^{v_0(s)} g^{v_{in}(s)} \pi_{v_{mid}}, g^{w_0(s)}\pi_w) / e(g^{y_0(s)}\pi_y, g)$$

From the following expression, we can see the correctness of such equation when all values are correctly computed:

$$\begin{aligned} e(\pi_h, g^{t(s)}) &= e(g^{h(s)}, g^{t(s)}) = \\ &= e(g,g)^{h(s)t(s)} = \\ &= e(g,g)^{(v_0(s)+v_{in}(s)+v_{mid}(s))(w_0(s)+w(s))-(y_0(s)+y(s))} = \\ &= e(g^{v_0(s)} g^{v_{in}(s)} g^{v_{mid}(s)}, g^{w_0(s)} g^{w(s)}) / e(g^{y_0(s)} g^{y(s)}, g) = \\ &= e(g^{v_0(s)} g^{v_{in}(s)} \pi_{v_{mid}}, g^{w_0(s)}\pi_w) / e(g^{y_0(s)}\pi_y, g) \end{aligned}$$

At this point, this SNARK verifiable computation construction is not zero-knowledge, the proof terms may reveal information about the witness w because they are not uniformly distributed. To add zero-knowledge the Prover will randomize its proof by adding new terms while keeping the divisibility property of $p(x)$ by $t(x)$ intact making impossible to extract the witness.

The following terms must be added to the public evaluation key EK or CRS to facilitate the randomization:

$$\left(g^{t(s)}, g^{\alpha t(s)}, g^{\beta_v t(s)}, g^{v_0(s)}, g^{\alpha v_0(s)}, g^{\beta_w t(s))}, g^{w_0(s)}, g^{\alpha w_0(s)}, g^{\beta_y t(s)}, g^{y_0(s)}, g^{\alpha y_0(s)}\right)$$

The Prover acts as before and constructs a proof π, but now selects the following random values: $\delta_{v_{mid}}, \delta_w, \delta_y \xleftarrow{R} \mathbb{F}$ in order to compute random multiples of $t(s)$ to replace the original polynomials $v_{mid}(s)$, $w(s)$ and $y(s)$ with these new randomized terms in the proof, so the polynomial $h(s)$ will be modified accordingly.

$$v'_{mid}(s) := v_{mid}(x) + \delta_{v_{mid}} t(s) \qquad w'(s) := w(s) + \delta_w t(s) \qquad y'(s) := y(s) + \delta_y t(s)$$

Accordingly, the value of $h'(s)$ will be:

$$h'(s) = \frac{(v_0(s) + v_{in}(s) + v'_{mid}(s))(w_0(s) + w'(s)) - (y_0(s) + y'(s))}{t(s)}$$

$$h'(s) = \frac{(\overbrace{v_0(s) + v_{in}(s) + v_{mid}(x)}^{v^\dagger(s)} + \delta_{v_{mid}} t(s))(\overbrace{w_0(s) + w(s)}^{w^\dagger(s)} + \delta_w t(s)) - (\overbrace{y_0(s) + y(s)}^{y^\dagger(s)} + \delta_y t(s))}{t(s)}$$

$$h'(s) = \frac{(v^\dagger(s) + \delta_{v_{mid}} t(s))(w^\dagger(s) + \delta_w t(s)) - (y^\dagger(s) + \delta_y t(s))}{t(s)}$$

$$h'(s) = h(s) + \delta_w v^\dagger(s) + \delta_{v_{mid}} w^\dagger(s) - \delta_y + \delta_{v_{mid}} \delta_w t(s)$$

$$h'(s) = h(s) + \delta_w(v_0(s) + v_{in}(s) + v_{mid}(x)) + \delta_{v_{mid}}(w_0(s) + w(s)) - \delta_y + \delta_{v_{mid}} \delta_w t(s)$$

With these replacements, due to the randomness applied to the values, they become indistinguishable and the proof construction is now zero-knowledge. From the previous calculations, the new computed proof π will be:

$$\pi = \left(g^{v'_{mid}(s)}, g^{w'(s)}, g^{y'(s)}, g^{h'(s)}, g^{\alpha v'_{mid}(s)}, g^{\alpha w'(s)}, g^{\alpha y'(s)}, g^{\alpha h'(s)}, g^{\beta_v v'_{mid}(s) + \beta_w w'(s) + \beta_y y'(s)}\right)$$

As the original document [14] indicates, the above zk-SNARK computation is statistical zero-knowledge, so it is required an additional property known as *re-randomizable*. The Prover is able to re-randomize its proof in order to construct new proof statements, but it is important to notice that anyone can apply this re-randomization. Zero-knowledge systems that allow these transformations are called *malleable*. On the one hand, for that reason, it should be considered to use a non-malleable SNARK construction such as the proposed by Jens Groth and Mary Maller [21]. On the other hand, this malleability can be considered as a useful feature for different proof systems [9].

CHAPTER 5

zk-SNARKs on Ethereum

The knowledge about how Ethereum works is supposed as well as blockchain fundamentals, but if the reader wants a depth understanding consider the following materials about Blockchain [1] and Ethereum [2].

The use of zero-knowledge proofs on the Ethereum blockchain allows to verify arbitrary computations by multiple parties, but the application of zk-SNARKs could achieve that in an efficient way.

Ethereum uses a quasi[1]-Turing-complete state machine called Ethereum Virtual Machine (EVM) in which elliptic curve operations are extremely complex to execute. So to provide this complex functions to the EVM exist the so-called precompiled contracts that are not executed inside the EVM, but implemented and executed by Ethereum clients. The disadvantage of that is the limitation of using a determined elliptic curve and a specific pairing function. In the last Ethereum upgrade (December 2019) called Istanbul[2] the Gas costs of some operations was reduced[3,4] making zk-SNARKs cheaper. The list of Istanbul precompiled contracts can be found in the source code.[5]

In this chapter an analysis of different toolboxes or protocols related to zk-SNARKs on Ethereum is done to understand how this mechanism runs on a general purpose public blockchain. Also, examples are provided when possible and always focused on a possible real-world implementation. Finally, applications derived from the examples are shown in a conceptual way for future works.

[1] Considered quasi-Turing-Complete because computing is intrinsically limited by a factor, Gas, which limits the total amount of computation.

[2] Ethereum Istanbul: https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement

[3] EIP 1108: Reduce alt_bn128 precompile Gas costs https://eips.ethereum.org/EIPS/eip-1108

[4] EIP 2028: Transaction data Gas cost reduction: https://eips.ethereum.org/EIPS/eip-2028

[5] Precompiled contracts Geth code: https://github.com/ethereum/go-ethereum/blob/master/core/vm/contracts.go#L66

5.1 Toolboxes, Networks & Protocols

5.1.1 ZoKrates

As introduced in Section 2.2, ZoKrates claims to be a toolbox for zk-SNARKs on Ethereum. ZoKrates allows developers to create and verify zero-knowledge proofs using Solidity[6] contracts. In the Byzantium hard fork there were introduced some enhancing cryptography changes on Ethereum, such as curve addition, scalar multiplication and pairing checks on the elliptic curve `alt_bn128`[7] in order to perform zk-SNARK verification.

ZoKrates allows to use verifiable computation in a decentralized application (DApp) starting from the definition of a proof using a high level language similar to Python and then using Solidity to verify it. Thus, in this section the usage of ZoKrates will be covered and illustrated with some examples.

It is worth to mention that this section is not intended as a tutorial for using the toolbox, but only a brief introduction to understand the concepts to be able to use it. For that reason, not all the aspects of ZoKrates will be covered.

How it works

First, there are two ways of using ZoKrates, the former is by loading the ZoKrates plugin on Remix[8] and the latter, which is the one we use and recommend, is by installing the ZoKrates CLI.[9]

Once ZoKrates is installed you will be able to use all the CLI commands ZoKrates provide, which are the following (or you can see them by running: `$ zokrates --help`):

- `compile`: Compiles a file written in ZoKrates high level language (`.zok`) into an arithmetic circuit representation. You need to set the path to the file with the flag `-i`.
- `compute-witness`: Generates a witness for the compiled constraint system produced by the previous command from public and private arguments provided by the Prover using the flag `-a arg1 arg2`.
- `export-verifier`: Using a verifying key (`verification.key`) generates a Solidity smart contract (`Verifier.sol`) which contains a public function (`verifyTx`) that accepts a proof and public inputs. The verifying key is hardcoded in the function `verifyingKey()`.
- `generate-proof`: Generates a proof in JSON format for the compiled constraint system and witness using a proving key (`proving.key`).
- `print-proof`: Prints the proof in JSON or Remix format.
- `setup`: Performs a trusted setup for the compiled constraint system and creates the proving (`proving.key`) and verifying (`verification.key`) keys derived from the *toxic waste*.

[6] Solidity is an object-oriented, high-level programming language for developing smart contracts influenced by C++, Python and JavaScript

[7] The `alt_bn128` curve is specified in section E.1 on Appendix E on the Ethereum Yellow Paper[39]

[8] An open source IDE to develop Solidity contracts straight from the browser, http://remix.ethereum.org.

[9] Instructions for installing ZoKrates can be found in: https://zokrates.github.io/gettingstarted.html.

Considerations

As of today (May 2020) ZoKrates is still under development and there are some considerations to take into account when you use it, besides that some details may be subject to change in the future.

ZoKrates has a high level language similar to Python, but when you are developing programs in ZoKrates you need to be aware of: shadowing is not allowed, positive integers are referred as `field` represented in $[0, p-1]$ and overflow[10] at p. Boolean types can only be used in `if` statements and for checking equality or inequality between `field` values. Some other particularities are reviewed in the official documentation.[11]

By default, when the trusted setup is performed or when the proof and the Solidity smart contract are generated, ZoKrates uses the proving schema in [20] which is subject to malleability. To avoid this malleability the developer can use different proving schemes ([33], [21]) or different mechanisms.[12]

Participants

Before start coding and introducing the commands listed above it is important to take into account the parts involved during the process. There are two possible scenarios:

- The Prover and the Verifier are the two parties involved, so the Verifier trusts his own setup phase.
- Based on the *public verifier* property where multiples parties verify the argument of the Prover. For that it is necessary to generalize the setup phase, a process known as *ceremony*, but ZoKrates does not support it yet.

Therefore, for obvious reasons the first option will be used to illustrate the commands typed by each party.

First of all, we focus on the Verifier that can be referred as the trusted party. Suppose that for some reason this party wants a second party to prove the knowledge of some information on the Ethereum blockchain. To start, the Verifier needs to code the program using the ZoKrates high level language. When the program is ready, the Verifier saves it as `test.zok`.

Then, the Verifier type the following to compile the program into an arithmetic circuit:

```
verifier:/$ zokrates compile -i test.zok
```

Here we do not use a specific program with specific arguments, this will be illustrated in the next section with an example.

After that, the Verifier performs the trusted setup and creates the proving and verifying keys running the following command:

```
verifier:/$ zokrates setup --proving-scheme g16 #Default scheme.
```

[10] Large prime number based on the `alt_128` curve, https://github.com/Zokrates/ZoKrates/blob/master/zokrates_field/src/field.rs#L23

[11] ZoKrates programming concepts: https://zokrates.github.io/concepts.html

[12] G16 malleability: https://zokrates.github.io/reference/proving_schemes.html#g16-malleability

The proving key should be shared with the Prover (or make it public) due it is necessary to generate the proof. The verification key is public because it is contained in the Solidity smart contract that will be generated running the command:

```
verifier:/$ zokrates export-verifier --proving-scheme g16 #Default scheme.
```

Finally, the Verifier deploys the `Verifier.sol` contract to the Ethereum blockchain. Now it is the turn of the Prover.

The Prover wants to demonstrate the knowledge of a secret that satisfies the constraints of the compiled program, so the secret is provided along with the following command:

```
prover:/$ zokrates compute-witness -a secret
```

The witness file is created. Then the Prover gets the proving key file which is publically available and generates the proof running the command:

```
prover:/$ zokrates generate-proof --proving-scheme g16 #Default scheme.
```

A JSON proof file is created which contains values that can be accepted in the `verifyTx` function of the Solidity contract along with public inputs. To finalize, the Prover submit its proof calling this function and if it is correct the contract emits an event and return true, otherwise only returns false. The Verifier can see the event triggered and the return statement by the transaction to check if the Prover sent the correct secret within the proof.

Example

In order to use an example as close as possible to a real application, the Prover (named Peggy) will prove to the Verifier (named Victor) that she is over eighteen years without revealing it. This is relatively easy but it is important to add mechanisms that guarantee that no one can use Peggy's proof.

First, Peggy's proof needs to be signed by her at the same time of proving her age. For that there are some limitations but the best way of doing this is using the ZoKrates Standard Library[13] for key derivation, specifically the ECC function `proofOfOwnership.zok` that verifies a given public/private keypair from the equation $p_k = G \cdot s_k$ (see Section 3.1.2).

ECC functions in ZoKrates use the Baby Jubjub [37] elliptic curve which uses the parameters defined in `babyjubjubParams.zok`.[14]

At this point, Victor can forward Peggy's proof to a second Verifier, so an additional mechanism that guarantees that only Victor verify her proof should be implemented. This can be done in a feasible way if Victor only verifies proofs that use his public key as an input.

[13] ZoKrates stdlib: https://zokrates.github.io/concepts/stdlib.html

[14] ZoKrates Baby JubJub parameters: https://github.com/Zokrates/ZoKrates/blob/master/zokrates_stdlib/stdlib/ecc/babyjubjubParams.zok

A possible implementation[15] of the program could be the following.

```
import "ecc/babyjubjubParams" as context
import "ecc/proofOfOwnership" as proofOfOwnership

def main(field[2] publicKeyA, private field secretKeyA, field[2] publicKeyB, field
    todayGMTts, private field birthdayGMTts) -> (field):

    context = context()
    field secretKeyB = 1628047910396079130384481809782300080 39

    field proofAkeypair = proofOfOwnership(publicKeyA, secretKeyA, context)
    field proofBkeypair = proofOfOwnership(publicKeyB, secretKeyB, context)

    field verifyAge = if 568024668 < (todayGMTts - birthdayGMTts) then 1 else 0 fi

    field output = if (proofAkeypair == 1 && proofBkeypair == 1 && verifyAge == 1) then 1
         else 0 fi

    return output
```

This implementation only should be used to illustrate the example. It is a simple program that verifies if the Prover is over eighteen, which equals to 568.024.668 seconds, calculating the difference between the GMT timestamp dates of today and the day of birth. In addition Peggy must submit her public/private keypair and the public key of Victor to prevent that anyone can use the proof of Peggy in other Verifiers.

First, Victor compile the program and get two files: `out` and `out.ztf` which are the compiled binary file and the human readable code respectively. The number of constraints of this program is 8.446.

Using the compiled binary file (`out`) Victor generates the trusted setup and creates two files: `proving.key` and `verifying.key`. Then generates the Solidity smart contract `Verifier.sol`[16] using the verification key.

Peggy wants to prove that she is over eighteen, so first she computes a witness using the parameters below running the command `prover:/$ zokrates compute-witness -a parameters:`[17]

publicKeyA
14897476871502190904409029696666322856887678969656209656241038339251270171395
16668832459046858928951622951481252834155254151733002984053501254009901876174
secretKeyA
19970113589829231689283449921999914806895468376215802393426564332342553790 25
publicKeyB
94403459375576262164081074036464695510891331394660661857227577497996200551 73
71058776159624153149390610717890533477698901334079780012712877853184164773 15
todayGMTts
1589760000
birthdayGMTts
703987200

[15] `identity.zok`: https://gist.github.com/ballesterosbr/6967dbd8b23ebf8d10ef88b534f84ebc
[16] `Verifier.sol`: https://gist.github.com/ballesterosbr/8924b8dc08ff9b217c60967abcb2824c
[17] compute-witness comd.: https://gist.github.com/ballesterosbr/03484f03a85f69235b3c659cdf6f8e23

The timestamps 703.987.200 and 1.589.760.000 correspond to 23th April 1992 and 18th May 2020 respectively so it is obvious that Peggy is over eighteen. With these arguments the program returns as a result the witness `1`, and if Peggy changes the `birthdayGMTts` to 1.082.678.400 that corresponds to 23th April 2004 the program returns the witness `0`.

After that, Peggy constructs the proof which is contained in a JSON file (`proof.json`):

```
{   "proof":
        {
        "a": ["0x0fd5b4919e42654e16e88cc9c09948162c8fdb854073636a09785da425435a47",
              "0x1ca5299759da5f0ef7320f0df03dda2f7b9bf8759d308966c25d113d367430d5"],
        "b": [["0x208718ecf8437acd623eb1ce99c989abad4fc51ac9e0d3166ca8741b56fd0ca8",
              "0x0dbb3305dd8e7a292ded4fd57d931af90923d98f87c128cfb90af06f88d5ba49"],
             ["0x0e08befd28e137af574dab986984cbc34ec766dd4ac8938c80571a709cab3e80",
              "0x268da6cd3395f177ad0265f47eab0930a443716e5152f21e2939cf8c36532c75"]],
        "c": ["0x139dd4a585c409e1e8ffd19cbf589bfb6264ad86a8a414c7656bd1dd63ca0a2e",
              "0x2bd6922db53c3ad30ad7309626a75d901bc031b969550112a26452190de2a88a"]
        },
    "inputs": ["0x20efac506a3d21dc3882103a7a727ad821f2422988783ee8b2f1762ecd0dbb03",
               "0x24da398ed1996eae6dafb3a687806e3a49c3add6949774d6017b30a66b2503ce",
               "0x14df0be4297c9b29b5887e4dea4e755f1c071fbd8ac75fec711298ae1ea21085",
               "0x0fb5c8c1977301b6e0a1eff6077dd1818016fca5d42ac5c6f08dc3da27456c83",
               "0x00000000000000000000000000000000000000000000000000000005ec1d000",
               "0x0000000000000000000000000000000000000000000000000000000000000001"]
}
```

Note that if you convert the input values from hexadecimal to decimal you will realize that the `secretKeyB` and `birthdayGMTts` values are omitted because they were defined as private. Peggy now can submit this proof by calling the function `verifyTx` of the smart contract, which has the following code:

```
function verifyTx(
        uint[2] memory a,
        uint[2][2] memory b,
        uint[2] memory c,
        uint[1] memory input
    ) public returns (bool r) {
    Proof memory proof;
    proof.a = Pairing.G1Point(a[0], a[1]);
    proof.b = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
    proof.c = Pairing.G1Point(c[0], c[1]);
    uint[] memory inputValues = new uint[](input.length);
    for(uint i = 0; i < input.length; i++){
        inputValues[i] = input[i];
    }
    if (verify(inputValues, proof) == 0) {
        emit Verified("Transaction successfully verified.");
        return true;
    } else {
        return false;
    }
}
```

A UML scheme of the smart contract is illustrated on Appendix A.

The smart contract was deployed on a local blockchain (`ganache-cli`) and the Gas usage of the contract deployment was: 1.109.590. The Gas usage of the transaction sent by Peggy when submitting the proof was: 284.090. The boolean output of the transaction was `true`.[18]

[18]Decoded output and logs: https://gist.github.com/ballesterosbr/71a6e2820872b0c5ea0ffb62a804f8dc

5.1.2 EY Nightfall

As defined in Section 2.2, Nightfall is an integration of different tools and a set of smart contracts that uses ZoKrates toolbox to achieve complete private token transactions on the Ethereum public blockchain.

Currently, there are two predominant smart contracts compatible with Nightfall which are the ERC20[19] (Token Standard) and ERC721[20] (Non-Fungible Token Standard) token standards chosen because of their popularity and extended use on Ethereum.

Nightfall integrates different microservices such as database, UI and offchain mechanisms, besides those related to zk-SNARKs. ZoKrates is used in Nightfall to generate zk-SNARKs, so note that if ZoKrates applies improvement updates, Nightfall could perhaps improve too, but there are some differences between the ZoKrates implementation on Nightfall and the previous one used in Section 5.1.1.

- Nightfall uses the so-called proving scheme GM17 [21].
- Nightfall does not use the `verifier.sol` smart contract generated by ZoKrates. Instead uses a verifier smart contract capable of managing verification keys using the scheme in [21] called `Verifier.sol` which corresponds to the EIP1922.[21]

Just like in ZoKrates, a trusted setup is required in Nightfall before anyone can compute proofs for the computations available. In Nightfall there are six main zk-SNARK computations available depending on the token type:

Non-fungible Token (ERC721)	Fungible Token (ERC20)
`non-fungible mint`	`fungible mint`
`non-fungible transfer`	`fungible transfer`
`non-fungible burn`	`fungible burn`

For each different zk-SNARK computation the trusted setup is performed at the beginning and only once. It is important that the generator of this trusted setup deletes the *toxic waste* produced in this process. The files produced by the trusted setup in Nigthfall is a set that corresponds to all the files generated by the Verifier in ZoKrates (i.e. `out`, `out.code`, `proving.key`, `verification.key`, `verifier.sol`) and JSON files related to the specific computation keys.

Non-fungible Token (ERC721) keys pair (main computations)		
`nft-mint-vk.json`	`nft-transfer-vk.json`	`nft-burn-vk.json`
`proving.key`	`proving.key`	`proving.key`

Fungible Token (ERC20) keys pair (main computations)		
`ft-mint-vk.json`	`ft-transfer-vk.json`	`ft-burn-vk.json`
`proving.key`	`proving.key`	`proving.key`

[19] ERC20 standard: https://eips.ethereum.org/EIPS/eip-20
[20] ERC721 standard: https://eips.ethereum.org/EIPS/eip-721
[21] Ethereum Improvement Proposal 1922: https://eips.ethereum.org/EIPS/eip-1922

Once the trusted setup is finished, on one hand the proving keys must be shared using an online service to allow potential Provers to use these files to generate proofs, it is not recommended to use a blockchain to store these files because they are too large. On the other hand, the verification keys are stored on the Ethereum public blockchain.

Note that because of the proving scheme used, the verification and proving keys will be different each time the trusted setup is performed.

As we can see, the verification keys do not use a `.key` format because Nightfall uses a JSON version of these keys and store them within the shield contract. So, when a Prover sends a computed proof to a shield contract, this proof is verified against the corresponding verification key stored in the shield contract by calling a verification function in the Verifier contract.[22] The process[23] is shown in Figure 5.1.

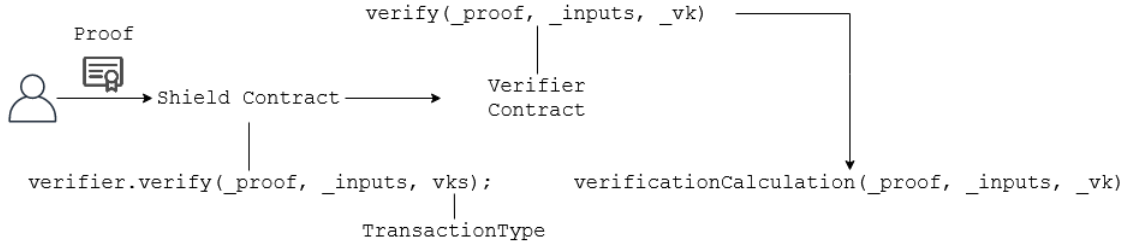


Figure 5.1: Nightfall proof verification flow

After the proving key has been shared, the generator of the trusted setup will deploy the GM17 verifier smart contract instance. At this time, the generator should choose the tokens to shield.

As previously commented, Nightfall operates with token standards. These tokens could exist or be deployed by the generator, but in the first case it would be necessary that the generator owns a value of tokens (ERC20) or the token (ERC721). To continue with the explanation we will suppose that the generator previously has deployed both type of tokens and would like to shield them.

Because the selection of the generator is to shield the two token standards, it will be necessary to deploy one shield smart contract for each original token with the following constructor parameter.

Shield Token	Constructor parameters	Parameter type
`FTokenShield.sol`	`Verifier.sol`	`address`
`NFTokenShield.sol`	`Verifier.sol`	`address`

Now the generator interacts with the shield contract and adds all the verification keys (`.json` files of the previous page) computed during the trusted setup. It is possible to change the Verifier contract address after the deployment of the shield contracts.

Finally the addresses of `FTokenShield.sol` and `NFTokenShield.sol` can be shared to allow Provers submit proofs.

[22] `Verifier.sol` source code: https://github.com/EYBlockchain/nightfall/blob/master/zkp/contracts/Verifier.sol

[23] Note that as of May 2020 this implementation is not updated in the Nightfall whitepaper.

To complete the Nightfall infrastructure, one user downloads the proving key from the online service used by the generator to store it. Then, this user generates a proof for a zk-SNARK computation and submits it to the corresponding shield token contract. The shield token contract passes the user's zk-SNARK proof to the Verifier contract, which returns true if the proof is verified or false if not. When the user sends the proof, which has the same format as in ZoKrates, the shield contract knows which verification key to use.

To better understand all the Nightfall infrastructure, a scheme is proposed in Figure 5.2.

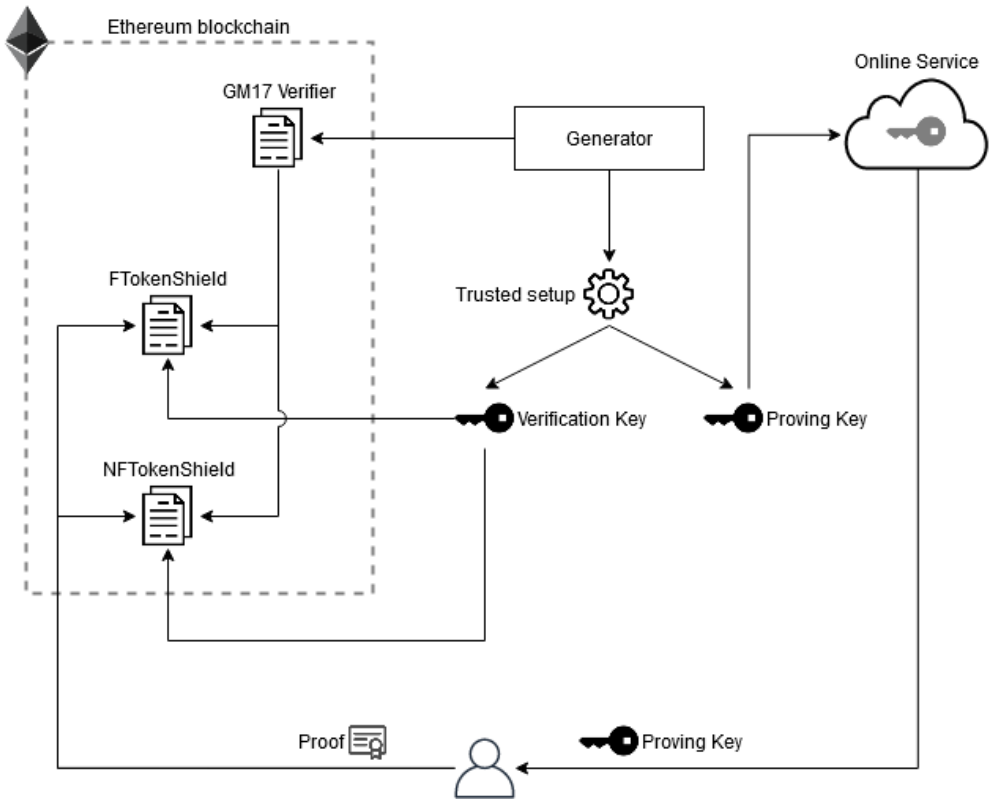


Figure 5.2: Nightfall infrastructure

In the previous scheme the verification flow between contracts is not shown, but it can be reviewed in Figure 5.1.

Some considerations should be taken into account about the verification keys:

- The verification keys stored in the `FTokenShield` contract are associated with the following transaction types: `Mint`, `Transfer`, `Burn`, `ConsolidationTransfer`, `MintRC`, `TransferRC`, `BurnRC` and `SimpleBatchTransfer`.
- The verification keys stored in the `NFTokenShield` contract are associated with the following transaction types: `Mint`, `Transfer` and `Burn`.
- The Prover must indicate the original token address of the ERC20 or ERC721 contract when submitting a proof to the following transaction types:
  - `FTokenShield`: `Mint`, `Burn`, `MintRC` and `BurnRC`.
  - `NFTokenShield`: `Mint` and `Burn`.

To understand transaction types, the interaction with shield tokens it is shown in Figure 5.3 and Figure 5.4. To simplify, only `Mint`, `Transfer` and `Burn` transaction types are considered.

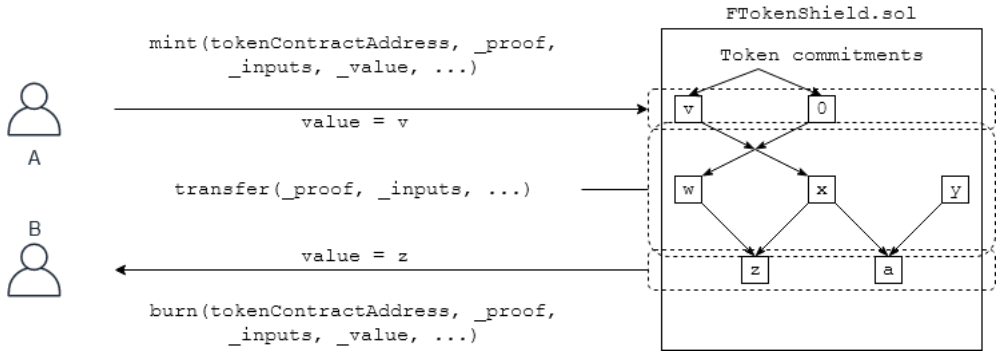


Figure 5.3: `FTokenShield.sol` transaction types

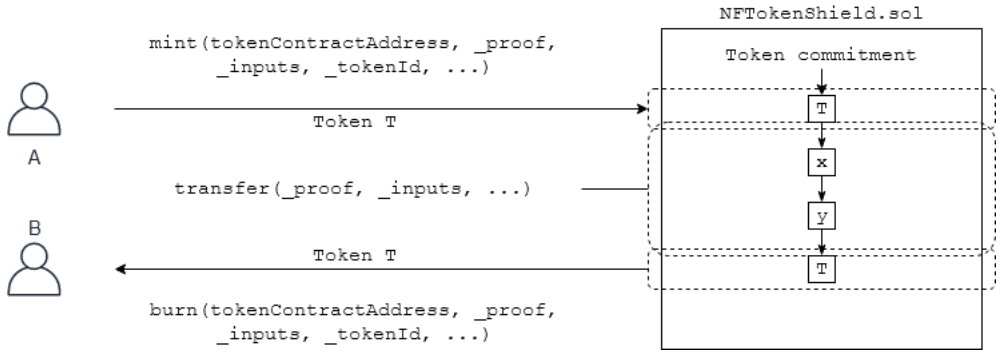


Figure 5.4: `NFTokenShield.sol` transaction types

From the two diagrams, it is easy to notice that the process of minting and burning obviously reveals what it is been transacted and who owns it on the Ethereum public blockchain, but within the shield contracts all transfers are hidden. Note that the burning process is optional and participants could stay in the shield contract environment indefinitely.

Token commitments represent ownership of a token (ERC721) or a particular amount of tokens (ERC20). The transfers are done within the shield contracts by submitting proofs instead of exchanging value directly as happens in the token standards.

For more information about the protocols of the shield tokens it is recommended to read the white-paper [26], but for implementation details check the official repository.[24]

An example of using Nightfall is not illustrated because the application displays an intuitive graphical interface to execute all possible computations. Also, as mentioned, due to the usage of ZoKrates for zk-SNARKs computations, Section 5.1.1 could help to understand how Nightfall works.

[24]EY Nightfall GitHub repository: https://github.com/EYBlockchain/nightfall

5.1.3 AZTEC Protocol

Anonymous Zero-knowledge Transactions with Efficient Communication (AZTEC) exists because it was created to enable privacy on public blockchains. This protocol implements confidential transactions, which hide the transaction value on general-purpose blockchains that support Turing-complete computations.

Considering the most known general-purpose blockchain, that is Ethereum, AZTEC not only allows to hide transaction amounts, but it can be used for confidential transactions of digital assets by attaching AZTEC tokens to existing tokens based on standards such as ERC20.

Different cryptographic methods are used to encrypt transaction values such as homomorphic encryption and range proofs[25], which are better known as zero-knowledge range proofs (ZKRP) and allow a Prover to prove to a Verifier if a number provided by the Prover lies within a determined range.

To understand how AZTEC works it is necessary to explain the concept of *Note*. A *Note* is the core of every transaction of the protocol and represents an encrypted abstract value. It is managed by the *Note Registry*, which exists in any AZTEC digital asset. The information contained in a *Note* is the following:

- **Public**: An Ethereum address of the owner of the note and an encrypted representation of the value.
- **Private**: The value of the note and a *viewing key* that allows the note to be decrypted and used to create join-split proofs.

There is another element known as *spending key* which is used to sign the proofs. The ownership of a note is determined by the possession of the *viewing* and *spending* keys.

Join-split proofs are a type of transaction that combines a set of input notes into a set of output notes in order to achieve confidential transactions and ensuring that the input notes sum is equal to the output notes sum without revealing any value. To better understand this, let us see a straightforward example:

One sender A has a total balance of 50 tokens conformed by two notes and wants to send 10 tokens to a receiver B. First, the sender A creates at least one note owned by the receiver B with a total value of 10 tokens. Then the sender A creates at least one note owned by him with a total value of 40 tokens. After that the sender A constructs a zero-knowledge proof which proves that input and output sums are equal and the balance match. Finally the AZTEC digital asset smart contract acts as a Verifier and validate this zero-knowledge proofs, removes the input notes and generates new output notes in the note registry.

To sum up, 10 tokens to receiver B and 40 tokens to sender A still holds the balance relationship, so AZTEC will replace the input notes with the output notes in the note registry. As you can appreciate the process is really similar to Bitcoin UTXOs.

[25] AZTEC range proofs: https://aztec-protocol.gitbook.io/aztec-documentation/v/master/overview/aztec-under-the-hood-range-proofs

Simultaneously at the moment when the sender A is creating the notes owned by receiver B, it also takes place the construction of the *viewing keys* that receiver B will be able to identify. Sender A (or prover) can use the public key of the receiver B, so the owner of this public key could derive the *viewing key* of the note and decrypt it using its private key. This is possible because every note has an ephemeral key that allows to derive the *viewing key*.

Finally, a note needs to be signed using the *spending key* in order to be spent for a particular address. This is done automatically in join-split proofs with a private key provided by the input note's owner.

For more details about AZTEC protocol notes and proofs it is highly recommended to read the white-paper in [38].

The principal component of the protocol is the AZTEC Cryptography Engine (ACE) which is a smart contract (`ACE.sol`[26]) that managed the state of all AZTEC assets that share a common trusted setup and subscribe to ACE.

This engine is the authority of the submitted proofs correctness by delegating the validation to specific contracts. Join-split proofs are not the only type of proofs supported by AZTEC, others with different functionalities are available with their corresponding validation smart contract.[27] Another functionality of this engine is to update the state in note registries by converting zero-knowledge proof data into instructions from proofs validated successfully.

As commented before, AZTEC protocol currently uses a different zero-knowledge technique called range proofs to provide balance privacy. Developers are working on preserve user privacy by hiding the issuer and the receiver of a transaction.

In March 2020 AZTEC announced their privacy roadmap in which they are working in a new efficient zk-SNARK construction called $\mathcal{P}\mathfrak{lon}\mathcal{K}$ [12] to reduce Gas costs of private transactions on the Ethereum mainnet. For being under development an example will not be provided.

Although AZTEC protocol does not use zk-SNARKs it has been included in this document for being related to zero-knowledge proofs on Ethereum and because they are developing a new zk-SNARK construction.

[26] ACE.sol: https://github.com/AztecProtocol/AZTEC/blob/develop/packages/protocol/contracts/ACE/ACE.sol

[27] ACE validators smart contracts: https://github.com/AztecProtocol/AZTEC/tree/develop/packages/protocol/contracts/ACE/validators

5.2 Use cases

From the analysis of toolboxes and protocols presented in the previous section, it is possible to define some real world use cases schemes based on zk-SNARKs which protect users privacy in different environments.

5.2.1 Digital Identity

Identification or authentication mechanisms are one of the most known possible usages of this cryptographic technique. The main idea here is to preserve user's privacy when he is proving his identity in an authentication process.

Blockchain applications that require identification or authentication could use zk-SNARK mechanisms to allow access without revealing any information of the user. Thus, trust is achieved in a trustless environment such as a public blockchain. Zk-SNARKs allow Verifiers to store the proof computed by the Prover as a transaction on the blockchain.

Using this mechanism avoid revealing any information such as date of birth, ID cards, academic credentials, driver licenses, other sensitive data or even passwords.

A possible authentication scheme is shown in Figure 5.5:

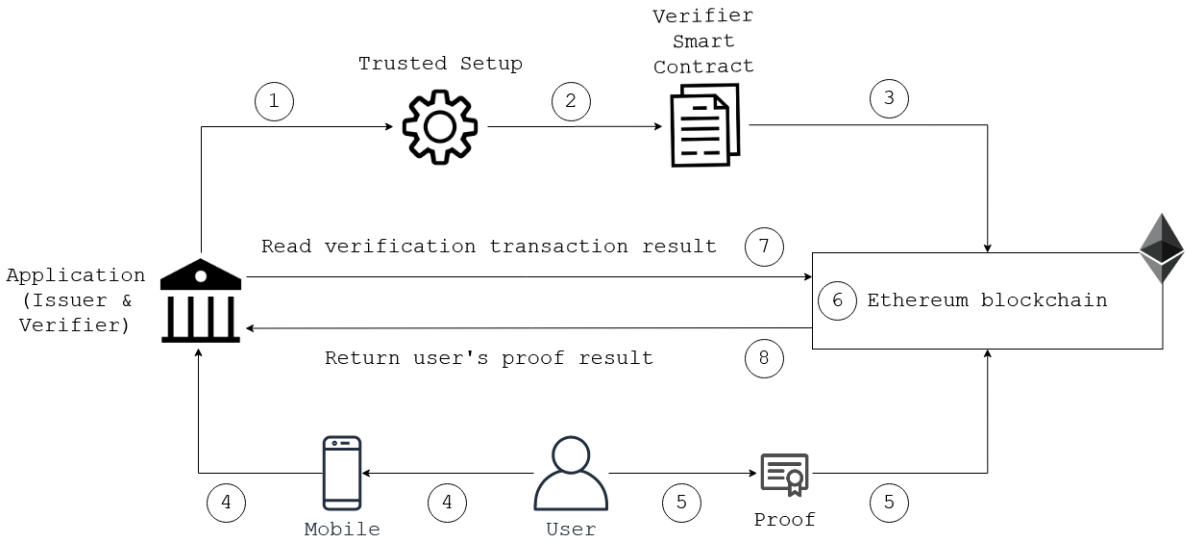


Figure 5.5: Authentication scheme

The flow of this scheme is based on an authentication scheme in an application. First, the application acting as a Verifier executes the trusted setup (1) from a program and generates the Verifier smart contract (2) that it is published on the Ethereum blockchain (3). The program should be related with user's attributes such as age or address also known as credential attributes. Then, the user tries to authenticate (4) and the application asks for a proof. User constructs a proof based on the attributes that the program will verify and submit it to the Verifier smart contract (5). The Verifier smart contract executes the transaction (6) using the user's proof and returns true or false. Finally, the application checks the transaction output (7) to see if the user sent a valid proof and if so grant access to him, otherwise the access will be denied (8).

The scheme of the Figure 5.5 could be different if legal entities are added to the environment as credential issuers and without acting as Verifiers, as happens in many situations where credentials are intended for broader adoption as ID cards, driver licenses or academic records. In this situation the Verifier must trust credential issuers, which should be organizations such as banks, healthcare providers, universities, or governments.

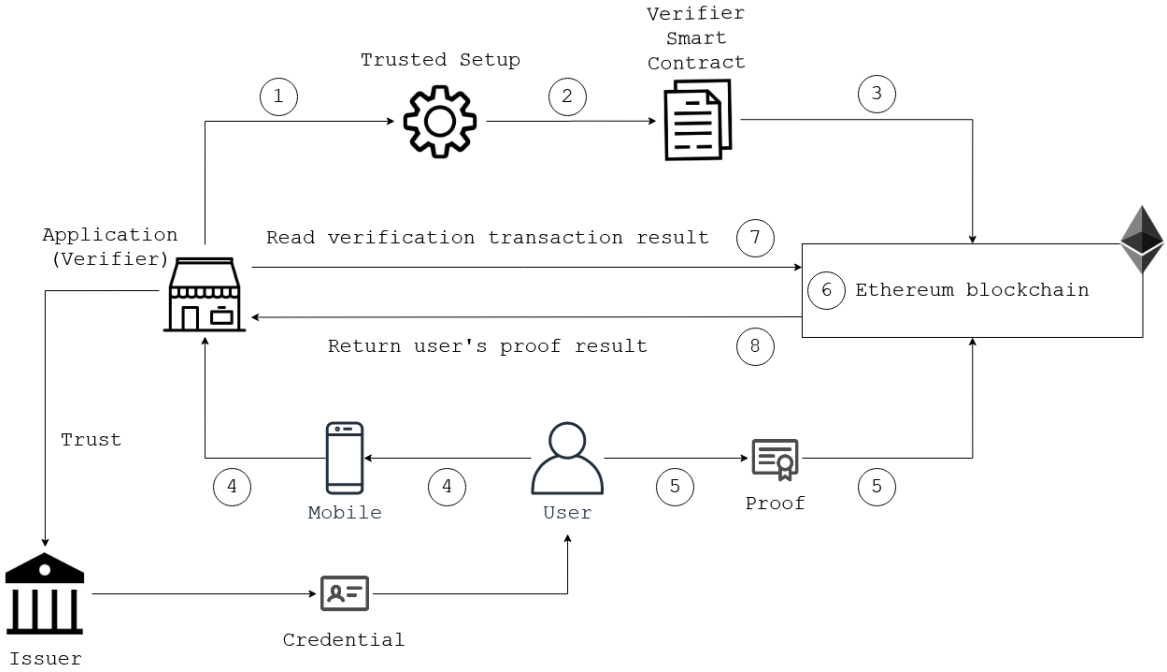


Figure 5.6: Authentication scheme based on legal issuer

As we can see in Figure 5.5 and Figure 5.6 the schemes are not for specific situations, but notice that Verifiers can take many forms and play many roles in different business. On one hand it is possible that credentials are only accepted by the same entity that issue them as happens in Figure 5.5, an example could be an entity loyalty card. On the other hand, some credentials are issued by different entities and should be verified by third parties as happens in Figure 5.6, an example could be ID cards or academic records.

In a compliant GDPR solution the combination of technologies should be applied such as DIDs[28], verifiable credentials[29] and peer-to-peer communications between the user and legal entities. A first approximation to this model is provided by the Sovrin network[30], but it could be replicated on networks capable of supporting DIDs like Ethereum.

5.2.2 Financial Services

Another environment where zk-SNARKs can be applied is the finance world, and actually the digital currency Zcash is the proof of this applicability, but here the focus is on the Ethereum network.

[28]Decentralized Identifiers (DIDs) v1.0: https://www.w3.org/TR/did-core/
[29]Verifiable Credentials Data Model 1.0: https://www.w3.org/TR/vc-data-model/
[30]Sovrin Governance Framework: https://sovrin.org/library/sovrin-governance-framework/

Sometimes, as it happens in the previous section about digital identity, a user wants to preserve his privacy when doing financial transactions or the receiver of a transaction does not want to show his identity, or sometimes the amount transferred is preferred to be hidden.

One of the requirements of financial transactions is that they need to be done in a short time in most cases, so the application of zk-SNARKs suits perfectly here.

As we saw in sections 5.1.2 and 5.1.3 digital assets could be transferred with some confidentiality or complete privacy depending on the token standards and the protocol used on the Ethereum network.

A possible scheme that shows the transfer of digital assets between two parties is shown in Figure 5.7.

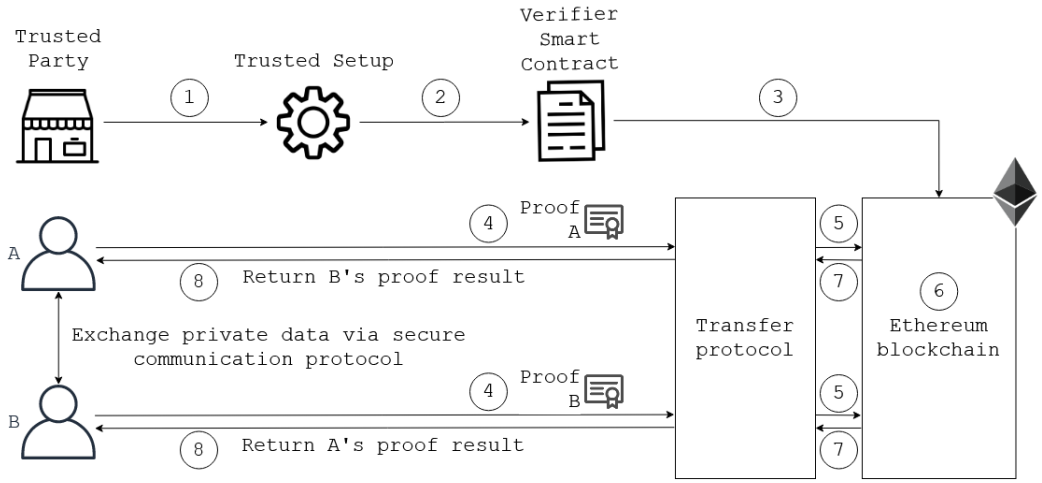


Figure 5.7: Private transactions in a financial scheme

The scheme in Figure 5.7 could be different depending on the protocols or parties involved in the architecture.

The flow of this scheme is based on the transfer of digital assets using computed zk-SNARK proofs. First, we suppose a trusted party who performs the trusted setup (1) from a program and generates the Verifier smart contract (2) that it is published on the Ethereum blockchain (3). Consider that the program is based on a protocol that enables digital asset transfers. Then, two users A and B that trust the trusted party want to exchange digital assets between them so they generate proofs for transfers and submit them to the transfer protocol (4), that calls the Verifier smart contract (5) to see if the proofs are correct or not in order to compute the transfer internally and move the digital assets between accounts. The Verifier smart contract executes the transactions (6) using the proofs, and stores the results that will be requested by the transfer protocol (7) and later consulted by the users (8). During the process, users A and B exchange private information related to the proofs and their accounts, so they can verify that transfers are executed correctly.

CHAPTER 6

Conclusion

The elaboration of the present thesis involves different perspectives related to zk-SNARKs such as their cryptographic definition and their applicability in real world schemes based on the usage of a general purpose public blockchain.

We must recall that zk-SNARKs are quite challenging to grasp besides all the previous concepts needed to understand how these proofs are constructed from a cryptographic point of view.

The main goal of this master's thesis was to provide a reference document about zk-SNARKs on the Ethereum public blockchain. We considered that this goal has been satisfactorily achieved since not only the construction of a zk-SNARK proof has been illustrated, but also the operation of toolboxes and protocols on Ethereum as well as the application to real world use cases.

To be more specific about the goals proposed in the Introduction chapter, Section 1.1, the following conclusions are referenced to them.

A complete construction of a zk-SNARK proof has been proposed to understand how it works. This has been possible due to the several documentation used related to this type of zero-knowledge proofs. Beyond the construction there are more aspects such as security or efficiency that should be taken into account, but they have not been covered in order to avoid a more complex explanation.

Analyses and definitions about different protocols and toolboxes have been done to show how zk-SNARKs work on the Ethereum public blockchain. And we conclude that it is not really necessary to understand the zk-SNARKs maths but it is highly recommended to understand the zk-SNARKs construction in a conceptual way to be able to develop proper programs.

In the case of ZoKrates, an example is provided to show how to implement zk-SNARK proofs based on a possible real application. Inherently in Nightfall all the work shown in the ZoKrates example is done and hidden to the user, and instead a user interface is provided by the suite.

Because of that, anyone can deal with zk-SNARK proofs using the Nightfall suite, but with the limitations that only certain operations related to the token standards are allowed. Meanwhile in ZoKrates it is required a higher programming knowledge, but there are no limitations beyond that those about technical specifications of the toolbox. In contrast, nowadays, it is important to take into account that none of the solutions analyzed in this document should be used in production. This is because all of them are under development and require more time to be considered mature enough to be applied to potential use cases.

Finally, from the schemes proposed about real use cases we can conclude that currently there are some restrictions about the usage of zk-SNARKs, because it does not exist a ceremony process that allows multiples parties to verify proofs, so the process is still complex to apply into a real solution. However the schemes could be applied right now in simpler solutions or considered in future implementations.

6.1 Future Work

Since this project covers multiple aspects related to zk-SNARKs, several research lines can be considered for future works where the present document could be considered as a guide or reference for them.

Although there is an explanation of a zk-SNARK construction presented in this project, there are some related aspects such as optimization or security that could be addressed in future works. This would provide more insights into these zero-knowledge proofs and even a study about execution on the Ethereum network could be formalized. Hence, a work about non-malleable zk-SNARKs could be performed in addition with the usage of secure proving schemes in ZoKrates.

From the above, an analysis of how zk-SNARK proofs impact on a general purpose public blockchain, like Ethereum, in comparison with current networks that use zk-SNARKs would offer a great insight about the applicability of these proofs in a decentralized application (DApp).

Regarding the tools presented in this thesis, future works could be focused on the study or analysis of each one to explain more advanced concepts in order to understand better how they work. This also can be applied to solutions that are only commented in this document. So, new real world application schemes in different environments not covered in this document could be proposed.

Ultimately, research projects can be carried out too, in fact the new SNARK proposal of AZTEC is considered very interesting for future works due to its novelty. Also, different existing zk-SNARK constructions can be studied such as the proposed in [21] or [33].

Bibliography

[1] A. Antonopoulos. "The Blockchain" and "Mining and Consensus", Mastering Bitcoin: Programming the Open Blockchain. O'Reilly, 2017. pp. 211-284.

[2] A. Antonopoulos and G. Wood. "What Is Ethereum?" and "Ethereum Basics", Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly, 2018. pp. 1-40.

[3] S. Arora and B. Barak. "NP and NP completeness", Computational Complexity: A Modern Approach. Cambridge University Press, 2007. pp. 39-55.

[4] S. Arora and S. Safra. "Probabilistic checking of proofs: a new characterization of NP", Journal of the ACM, Vol. 45, No. 1. Princeton, New Jersey and Tel-Aviv, Israel, 1998. pp. 70–122.

[5] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture", SEC'14: Proceedings of the 23rd USENIX conference on Security Symposium. Berkeley, CA, 2014. pp. 781–796.

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again", ITCS '12: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. Association for Computing Machinery, NY, 2012. pp. 326–349.

[7] M. Blum, A. De Santis, S. Micali, and G. Persiano. "Non-Interactive Zero Knowledge", SIAM Journal on Computing, Vol. 20, No. 6. Massachusetts, MA, 1991. pp. 1084–1118.

[8] M. Blum, P. Feldman, and S. Micali. "Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract)", STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing. Association for Computing Machinery, NY, 1988. pp. 103–112.

[9] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn. "Malleable Proof Systems and Applications", Advances in Cryptology – EUROCRYPT 2012. Cambridge, UK, 2012. pp. 281-300.

[10] G. Danezis, Fournet, J. C. Groth, and M. Kohlweiss. "Square Span Programs with Applications to Succinct NIZK Arguments", Advances in Cryptology – ASIACRYPT 2014. Taiwan, R.O.C., 2014. pp. 532-550.

[11] G. Di Crescenzo and H. Lipmaa. "Succinct NP Proofs from an Extractability Assumption", CiE '08: Proceedings of the 4th conference on Computability in Europe: Logic and Theory of Algorithms. Berlin, Heidelberg, 2008. pp. 175–185.

[12] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. $\mathcal{P}\mathtt{lon}\mathcal{K}$: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[13] R. Gennaro, C. Gentry, and B. Parno. "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers", Advances in Cryptology – CRYPTO 2010. Santa Barbara, CA, 2010. pp. 465-482.

[14] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs". Cryptology ePrint Archive, Report 2012/215, 2012.

[15] C. Gentry and D. Wichs. "Separating Succinct Non-Interactive Arguments From All Falsifiable Assumptions", STOC '11: Proceedings of the forty-third annual ACM symposium on Theory of computing. Association for Computing Machinery, NY, 2011. pp. 99–108.

[16] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. "Delegating Computation: Interactive Proofs for Muggles", STOC '08: Proceedings of the fortieth annual ACM symposium on Theory of computing. Association for Computing Machinery, NY, 2008. pp. 113–122.

[17] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)", STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing. Association for Computing Machinery, NY, 1985. pp. 291–304.

[18] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof-Systems", SIAM Journal on Computing, Vol. 18, No. 1. University City Science Center Philadelphia, PA, 1989. pp. 186–208.

[19] J. Groth. "Short Pairing-based Non-interactive Zero-Knowledge Arguments", Advances in Cryptology - ASIACRYPT 2010. Singapore, 2010. pp. 321-340.

[20] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments", Advances in Cryptology – EUROCRYPT 2016. Austria, 2016. pp. 305-326.

[21] J. Groth and M. Maller. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs", Advances in Cryptology – CRYPTO 2017. Santa Barbara, CA, 2017. pp. 581-612.

[22] American National Standards Institute. X9.63 Public Key Cryptography for the Financial Services Industry Key Agreement and Key Transport Using Elliptic Curve Cryptography, 2001.

[23] American National Standards Institute. X9.62 Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[24] M. Karchmer and A. Wigderson. "On Span Programs", [1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference. San Diego, CA, 1993. pp. 102-111.

[25] N. Koblitz. "Elliptic Curve Cryptosystems", Mathematics of Computation, Vol. 48, No. 177, 1987. pp. 203-209.

[26] C. Konda, M. Connor, D. Westland, Q. Drouot, and P. Brody. "Nightfall". EY Global Blockchain R&D, 2019.

[27] Ç. K. Koç. "Good Curves", Cryptographic Engineering. Springer, 2009. pp. 184.

[28] H. Lipmaa. "Progression-Free Sets and Sublinear Pairing-Based Non-Interactive Zero-Knowledge Arguments", TCC 2012: Theory of Cryptography. Italy, 2012. pp. 169-189.

[29] D. Meffert. Bilinear Pairings in Cryptography - Master Thesis, 2009.

[30] A.J. Menezes, P. C. van Oorschot, and S.A. Vanstone. "Overview of Cryptography", Handbook of Applied Cryptography, Fifth Printing. CRC Press, 2001. pp. 1-45.

[31] V. S. Miller. "Use of Elliptic Curves in Cryptography", Advances in Cryptology — CRYPTO '85 Proceedings, 1985. pp. 417-426.

[32] National Institute of Standards and Technology. Digital Signature Standard (DSS), 2013.

[33] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly Practical Verifiable Computation", 2013 IEEE Symposium on Security and Privacy. Berkeley, CA, 2013. pp. 238-252.

[34] R. L. Rivest, L. Adleman, and M. L. Dertouzos. "On Data Banks and Privacy Homomorphisms", Foundations of Secure Computation. Orlando, FL, 1978. pp. 169-179.

[35] B. Schneier. "Mathematical Background", Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C. 20th Anniversary Edition. Wiley, 2015. pp. 233-264.

[36] L. C. Washington. "Divisors", Elliptic Curves: Number Theory and Cryptography, Second Edition. CRC Press, 2008. pp. 339-379.

[37] B. WhiteHat, J. Baylina, and M. Bellés. Baby Jubjub Elliptic Curve, 2018.

[38] Z. J. Williamson. The AZTEC Protocol, 2018.

[39] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, updated 2019.

Appendices

APPENDIX A

Verifier.sol Smart Contract UML

Figure A.1: Verifier.sol UML

Alberto Ballesteros Rodríguez
Madrid, 2020