Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Proving ownership of Bitcoin-like UTXO's using a zk-SNARK scheme

Semester Thesis

Aaron Menegalli-Boggelli

aaronme@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tejaswi Nadahalli
Prof. Dr. Roger Wattenhofer

June 5, 2021

Abstract

We study the core principles behind zk-SNARK schemes using the Pinocchio protocol [1] as reference. Furthermore, the potential of such schemes is highlighted by addressing the problem of proving ownership of some Bitcoin in a simplified setting using the ZoKrates [2] toolbox.

Contents

Chapter 1

Introduction

1.1 Motivation

The popularity and use of cryptocurrencies and blockchain technologies have accelerated the research in this cryptography field in recent years. Complex concepts that until a few years ago were merely theoretical begin now to see use in real-life applications, driven by this big trend. A promising technology that is having a big impact are so called zk-SNARK proofs (zero-knowledge Succinct Non-interactive ARgument of Knowledge). They are one of the most recent evolution of the concept of zero knowledge proofs, firstly introduced in 1989 in [3] and they are considered by the community a key component for solving the scalability problem that blockchains are currently facing, among having many other applications.

1.2 Goal

The goal of this thesis is to understand what are the different components that allow zk-SNARKs to achieve what they claim and how they are used to achieve it. The Pinocchio Protocol [1] is considered the first proposed scheme to have a functional practical implementation applicable to general computation and will be our reference. Using ZoKrates [2], a toolbox for implementing zk-SNARKs for a specified computation, we will try to use the Pinocchio protocol to prove ownership of some Bitcoin without revealing the relevant UTXO's to highlight a potential problem that can be addressed using such a scheme.

Chapter 2

Zero-knowledge Proofs

2.1 A short introduction to zero-knowledge proofs

The term *zero knowledge* refers to those *interactive* proof systems that reveal nothing more than the the correctness of the statement being proven. All we learn from the proof is just a single bit: "statement true" or "statement false", and nothing more. This is unusual, as often traditional proofs provide arguments from which other information can be learned. For instance, when proving that a graph is connected, we reveal some information about the graph as a part of the arguments of the proof, like some nodes or edges of the connected path. The notion of a zero-knowledge proof seems weird, if not even a bit contradictory. Let us recall what's in a classical proof. Statements which we want to assess being true or false are represented as strings $z \in \{0,1\}^*$ and a language $L \subseteq \{0,1\}^*$ identifies those strings that represent a "valid" or "true" statement. A proof is also modeled as a string $x \in \{0,1\}^*$ and, together with a verification function ϕ that takes as inputs z and x, assesses if $z \in L$. The point is that a classical proof is just a fixed and deterministic string.

By introducing the elements of randomness and interaction between a prover and a verifier, the authors in [3] were able to show that it is possible to convince someone that a statement is true (with high confidence) without revealing anything else. A proof isn't a fixed string given to the verifier anymore, but an interaction between two parties and randomness can be used by the verifier to give "challenges" to the prover.

Informally, a proof qualifies as a *zero-knowledge proof* if it satisfies 3 conditions:

- Completeness: It is possible to convince a verifier of a true statement
- Soundness: It is not possible to convinve a verifier of a false statement
- Zero-knowledge: A (possibly dishonest) verifier learns nothing more than the single bit "true statement"/"false statement" after running the protocol

The introduction of this concept led to an explosion of work in this particular field of cryptography that laid the foundations for the more recent developments on zk-SNARKs schemes, which we investigate in this thesis.

Chapter 3

Pinocchio Protocol

zk-SNARK, an acronym firstly introduced in [4] stands for *zero-knowledge Succinct Non-interactive ARgument of Knowledge* and refers to a set of algorithms through which a prover can convince a verifier of its knowledge about a certain statement and the proof needed to achieve that is significantly smaller than the statement itself, hence *succinct*. No interaction is required to occur between the two parties, hence *non-interactive*, and knowledge about the statement cannot be extracted by artefacts produced by the prover during the protocol, hence *zero-knowledge*. The Pinocchio protocol, introduced in 2013 in [1], is the first scheme to make use of such a system that is applicable in practice to prove correctness of general computation. Its main goal was to provide a scheme for *verifiable computation* (VC). VC enables the outsourcing of a computation by a client to a potentially untrusted server whose output can be verified to be correct in much less time than performing the computation itself. Pinocchio also suggests an extension to make VC zero-knowledge, where the outsourced computation is a function of two inputs, one assigned by the client and one by the server, and the client learns nothing about the server input during verification.

In this chapter we examine the basic elements that enables such systems to achieve what they claim and give explanations on why they are needed and what their combinations enables.

3.1 Polynomials

Polynomials are at the core of every zk-SNARK protocol. They are a medium that lends itself very well to carrying a proof of knowledge. In a zk-SNARK protocol the computation is first transformed into sets of polynomials and convincing a verifier of the correct execution of it is actually turned into convincing him of the knowledge of a specific polynomial. Polynomials have various useful properties and thanks to polynomial interpolation we can construct and shape them exactly as we want them to be, allowing us to build encoding of other objects using them.

3.1.1 Knowledge of a polynomial

Shared points

A nice property of polynomials is that two polynomials of degree at most d have at most d points in common. It intuitively follows from equating them: the solution of the equation are all the shared points and it can be rewritten as equating another polynomial to zero, which corresponds to finding its roots. Clearly, since there is no multiplication going on in the transformation the degree of the resulting polynomial will be the largest of the degrees of the two polynomial in question, as the following example clarifies:

$$a_3x^3 + a_2x^2 + a_1x + a_0 = b_2x^2 + b_1x + b_0$$
$$a_3x^3 + a_2x^2 + a_1x + a_0 - b_2x^2 - b_1x - b_0 = 0$$

From the Fundamental Theorem of Algebra we know that a polynomial of degree d has exactly d complex roots, or, stated in another way, at most d real roots, convincing ourselves of the initial statement. This property turns out to be useful in convincing someone of the knowledge of a specific polynomial since, given a random evaluation point, the probability that it is shared with another specific polynomial is negligible - if we are dealing in a large space of evaluation points.

Roots

The fundamental element that is exploited in a zk-SNARK scheme is proving that a polynomial has some specific roots. Since a polynomial can be represented as the multiplication of monomials of degree 1:

$$p(x) = (x - r_1)(x - r_2)(x - r_3)...$$

where $r_1, r_2, r_3, ...$ are its roots, in order to test $p(x)$ on having roots r_a and r_b we can construct another polynomial $t(x)$ as follows:

$$t(x) = (x - r_a)(x - r_b)$$

and check if it divides $p(x)$ without a remainder. In other words, if there exists another polynomial $h(x)$ such that $h(x) = \frac{p(x)}{t(x)}$, then it is true that r_a and r_b are roots of $p(x)$. $t(x)$ is often called the *target polynomial* and if it divides $p(x)$ its factors are also cofactors of $p(x)$.

Prover-verifier interaction

Let's exploit the above presented properties and imagine that a prover knows a polynomial $p(x)$ and wants to convince a verifier that it has some specific roots. As a first elementary scheme the verifier can proceed as follows:

- construct the polynomial $t(x)$ containing the monomials with the roots claimed by the prover
- Draw an evaluation point s at random
- Evaluate $t = t(s)$ and send to the prover s

The prover on its side should:

- Compute $h(x) = \frac{p(x)}{t(x)}$
- Evalutate $p(x)$ and $h(x)$ on s: $p = p(s)$, $h = h(s)$
- Send to the verifier p and h

The verifer should check if $p = h \cdot t$ holds. Recall that the evaluation of a multiplication of polynomials on a point x is equivalent to multyplying the single evaluations of the single multiplicands polynomials on point x. Cool, if the prover acted honestly, then the final check really tells the verifier that $p(x)$ has in fact the claimed roots, but can we be convinced of this just by checking that a simple multiplication holds? Of course not. It can easily be cheated. Since the prover knows s, he can also derive $t = t(s)$, pick a random h and set $p = ht$ and still pass the verification equation. Clearly we need something to restrict the prover from being able to do this.

3.1.2 Obscure evaluation

The problem is that the prover knows the evaluation point s and can consequently derive $t(s)$. We want to avoid this, but still let the prover evaluate its polynomials on the randomly drawn evaluation point. Seems quite an hard task but homomorphic encryption, as we will shortly introduce, will allow us to achieve that.

Homomorphic encryption

Homomorphic encryption refers to those encryption schemes that not only encrypt data, but also allow computation to be done on the encrypted data. Basically, when an encryption scheme has an homomorphic property, it means that specific arithmetic operations can be applied to the ciphertext, resulting in the encryption of the same operation applied to the plaintext. There exists various types of homomorphic encryptions, which differentiate themselves in what types of arithmetic operation are allowed to be applied. In particular, fully homomorphic encryption allows any efficiently computable function f to be applied to the ciphertext. To continue our study of the basic elements of the Pinocchio protocol and solve the problem that arose in the previous section we introduce a simple encryption scheme that allow multiplication with an unencrypted value.

As many encryption scheme do, it will be based on the hardness of the discrete logarithm problem (DLP).

Definition 3.1 (DLP)**.** Let p and q be large primes such that q divides $p-1$, let $G_q = \{1, g, g^2, g^3, ..., g^{q-1}\}$ be a set of powers of g mod p, forming *cyclic group* of prime order q with generator p. Set $y = g^x$ mod p where x is a uniformly random value in $\{0, 1, ..., q-1\}$. **Find x**.

If we choose big enough p and q we make the problem in definition 3.1 sufficiently hard for an encryption scheme to be secure. Encrypting a value x form the set $\{0, 1, ..., q-1\}$ is then just raising the generator to the power of x modulo p:

$$E(x) = g^x \; mod \; p$$

Since we are not bothered with formality but more with giving the intuition behind the concept, we will drop the modulo p in the rest of this chapter and assume we are working with values of the group. Multiplying a ciphertext with an unencrypted value c is just exponentiating the encryption to c:

$$(g^x)^c = g^{cx}$$

We can now make use of such encryption to evaluate a polynomial on an encrypted point by knowing the encryptions of every power of the evaluation point up to the degree of the polynomial. For the evaluation point s and the polynomial $p(x) = a_2x^2 + a_1x + a_0$, by knowing (g^1, g^s, g^{s^2}), we can compute the encrypted evaluation of $p(s)$ through exponentiation with its coefficients:

$$E(p(s)) = \left(g^{s^2}\right)^{a_2} \cdot (g^s)^{a_1} \cdot \left(g^1\right)^{a_0} = g^{a_2s^2+a_1s+a_0} = g^{p(s)}$$

Going back to our first elementary scheme, we have now a way of hiding the evaluation point from the prover but still letting him do computation with it thanks to the homomorphic property of the proposed encryption. For a claimed polynomial of degree d, the scheme can be adapted as follows:

- Verifier
  - Construct target polynomial $t(x)$ with the claimed roots
  - Draw random evaluation point s
  - Evaluate $t(s)$ and send to the prover the encryptions $g, g^s, ..., g^{s^d}$
- Prover
  - Compute $h(x) = \frac{p(x)}{t(x)}$

  - Evaluate $p = g^{p(s)}$, $h = g^{h(s)}$ and return them to the verifier

- Verifier
  - Check the if the following equation holds: $p = h^{t(s)}$
  - This is equivalent to the multiplication check but in encrypted space:

$$g^{p(s)} = \left(g^{h(s)}\right)^{t(s)} \iff g^{p(s)} = g^{h(s)t(s)}$$

The prover is now restricted from forging fake p and h like before since he doesn't know the secret evaluation point s and can't therefore derive $t(s)$. Still the scheme is not yet uncheatable. Observe that nothing prevents the prover from computing $g^{t(s)}$, as he is in possession of the target polynomial $t(x)$ and the encryptions of the powers of the evaluation point $g^s, g^{s^2}, ..., g^{s^d}$. This is unacceptable as it enables him to use another workaround and fool the protocol. He can just pick a random r, encrypt it as g^r and set $p = (g^{t(s)})^r$ and $h = g^r$. The verification equation will pass. Clearly we must add another piece to the puzzle, something that restrict the prover in returning powers of the provided encryptions and not something picked at random like r.

Knowledge of exponent assumption

What we want to achieve is to obligate the prover to return only an exponentiation of the provided encrypted values and nothing else, which means being able to tell if the he returned another random encrypted value, like g^r in the previous example. The Knowledge of Exponent Assumption (KoEA) is a widely known cryptographic assumption often used to prove the security of cryptographic protocols. It was introduced by Damgård in [5]. Informally, it states that in a prime order group G with generator g, given as input g^a, the only way to output another group value and its exponentiation with a, namely $(g^r, (g^r)^a)$, for a picked r, is to compute g^r and then $(g^a)^r$. There is no other way to do it. Without losing ourselves in the details on why this is guaranteed, let's see how it is exploited.

Assuming that the KoEA holds, requiring the prover to return only an exponentiation of a provided value g^a, and nothing else, is accomplished by the Pinocchio protocol in the following way: the pair $(g^a, g^{\alpha a})$, for a randomly drawn α that is kept as a secret, is sent to the prover, which is then asked to return the same exponentiation of both elements with a value c of its choice, as $((g^a)^c, (g^{\alpha a})^c)$. The following verification:

$$(g^{\alpha a})^c = ((g^a)^c)^\alpha$$

which requires the knowledge of the secret α, holds if and only if the prover acted accordingly. Since the prover isn't in possession of the secret α, there is no other way for him to fulfill the verification equation other than responding with exponentiating the two received values with the same exponent.

We have now a way to make the proposed elementary scheme invulnerable against the demonstrated cheats. In addition to sending to the prover the encrypted powers of the secret evaluation point, the verifier randomly samples a secret value α and also sends $g^{\alpha}, g^{\alpha s}, g^{\alpha s^2}, ..., g^{\alpha s^d}$. With these values the prover can calculate $g^{\alpha p(s)}$ and the verifier can perform the verification on the provided $g^{p(s)}$, ensuring that no other values was used to construct it other than $g, g^s, g^{s^2}, ..., g^{s^d}$.

3.1.3 Non-interactivity

So far we studied the key basic elements needed to construct a robust scheme that allow to prove knowledge of a polynomial with some specified roots, but which required interactivity between the two parties. In this section we explain what are the needed principles to get rid of this requirement and make the scheme non-interactive. It is clear that in the proposed interactive setting only the verifier and nobody else should trust the proof provided by the prover. If the verifier forwards the received proof to someone else, the latter cannot exclude the verifier shared the secret parameters with the prover or that the verifier himself forged a fake proof. A different interaction for every user would be required to convince many parties simultaneously, but this is of course inefficient.

If we can make the secret parameters (s, α) reusable by many verifiers, then a single proof, using just those parameters, would be enough to convince anyone. This means that, in order for anyone to be able to perform the verification of the proof, the parameters should be made public. Obviously, due to the possible cheats presented earlier, we can't let this happen. The clear answer is to first encrypt them and then make them public, making them available to anyone but impossible to use to forge a malicious proof or to tamper with. With this comes a problem though. Recall that the verification requires to multiply in encrypted space the secret parameters with the received values from the prover, but this is no more possible if s and α aren't known anymore. We therefore need a way to allow multiplication of two encrypted values.

Pairing-based cryptography

Pairing based cryptography is what allows us to solve the above presented problem and achieve non-interactivity. An encryption scheme based on *bilinear pairings* achieves the desired property of being homomorphic under the multiplication operation, in addition to the addition operation. This is referred to as double homomorphic encryption. Informally a bilinear pairing on (G, G_T) is a map:

$$e : G \times G \mapsto G_T$$

where G and G_T are cyclic group of the same order, and it satisfies some specific conditions. We won't present the single conditions here since it is not useful to introduce these formalities to understand the key point behind why this concept is fundamental to the scheme, but we highlight what is the important property which is exploited. Basically, the pairing function e allows to represent the encrypted multiplication of two group values in a *target* group G_T, taking as input they're encrypted values in the original group G:

$$e(g^a, g^b) = e(g, g)^{ab}$$

The output of e can be interpreted as the encrypted multiplication of the two encrypted inputs, but under a different generator. Furthermore it still maintain the homomorphic property for the addition operation:

$$e(g^a, g^b) \cdot e(g^c, g^d) = e(g, g)^{ab+cd}$$

Pairings are constructed using elliptic curves groups and there are specific curves created to efficiently support them, but we won't explore this field and stick to our simple example. Using pairing based cryptography allow now any verifier to perform the verification stage with the encrypted secret parameters:

$$e(g^{p(s)}, g) = e(g^{t(s)}, g^{h(s)})$$
$$e(g^{\alpha p(s)}, g) = e(g^{p(s)}, g^{\alpha})$$

Trusted Party Setup and CRS

Protocols like Pinocchio need to have a so called trusted party setup. We need to have a third party that everyone trust to generate the secret parameters, encrypt them and then, importantly, erase them. The secret parameters, once generated, are referred to as *toxic waste*, as it is really important that they are removed once all the other element are derived, such that the risk of the prover getting access to them is zero. At the beginning of the protocol, in what is called the *setup phase*, a trusted party randomly samples the secret parameters and construct the public evaluation key, containing the elements needed by the prover to construct the proof, and a public verification key, containing the elements needed to verify a proof. These two keys are glued together in the so called *Common Reference String* (CRS). Afterwards the trusted party erase the sampled parameters.

3.2 Quadratic Arithmetic Program

In section 3.1 we introduced the basic building blocks of a zk-SNARK protocol and explained, using a simple example scheme, how they operate to enable a prover to convince a verifier of its knowledge of a polynomial with some specified roots. We now move closer to the real Pinocchio protocol introducing *Quadratic Arithmetic Programs* (QAPs), which essentially are systems for encoding a general computation with polynomials and that can be used to build a VC protocol using the techniques presented in the previous section.

3.2.1 Transforming computation into polynomials

A general computation function F can be broken down into a set of elementary operations of the form:

$$l \text{ } operator \text{ } r = o$$

where l, r, o encode the left operand value, right operand value, the output value and *operator* a mathematical operation like addition or multiplication. Let's give a simple example. The following computation:

$$y = 3x^2 + 2x + 1$$

can be expressed as the following set of elementary operations:

$$\begin{aligned}
1: &\quad & x \cdot x &= out_1 \\
2: &\quad & 3 \cdot out_1 &= out_2 \\
3: &\quad & 2 \cdot x &= out_3 \\
4: &\quad & out_2 + out_3 &= out_4 \\
5: &\quad & out_4 + 1 &= y
\end{aligned}$$

Knowing an input and the corresponding output for the presented computation means being able to assign values to the different variables of the set of elementary operations such that all the equations hold. The idea now is to construct three polynomials $l(x)$, $r(x)$ and $o(x)$ through which, in a way that will be explained later, we can assign values to the different variables in the different operations.

To better explain what are we aiming at let's consider only one elementary operation, for instance:

$$1: \qquad v_1 \cdot v_2 = v_3$$

What we want to achieve is that $l(x)$, $r(x)$ and $o(x)$ evaluate to the value assigned to the variables present in this specific operation at the evaluation point $x = 1$ (the index of the operation), namely, if $v_1 = 1,\ v_2 = 2,\ v_3 = 3$, then $l(1) = 1,\ r(1) = 2,\ o(1) = 3$, which make the following equation holds:

$$\begin{aligned}
& l(1) \cdot r(1) = o(1) \\
\iff & l(1) \cdot r(1) - o(1) = 0
\end{aligned}$$

If we create construction rules for polynomials that achieve this, we can check if, with the assigned values, operation 1 holds by verifying that the polynomial

$$p(x) = l(x) \cdot r(x) - o(x)$$

has a root r_1 at $x = 1$.

Imagine now that we construct the three polynomials $l(x),\ r(x)$ and $o(x)$ such that this holds for all the elementary operations, i.e the operation at index 2 will be encoded with $l(2),\ r(2),\ o(2)$ and so on for all the d operations. Furthermore assume that all the operations use multiplication as the operator. Then, to verify that every operation holds, we can construct a target polynomial in the following way:

$$t(x) = \prod_{j=1}^{d} (x - j)$$

and check if it divides $p(x)$. Doing this, as introduced in the previous section, verifies that all the operation indices are roots of the polynomial $p(x)$, which in turn, due to the rules with which the polynomials were constructed, verifies that all operations hold, meaning that the assigned values are correct for the given computation. Note that we didn't explained the rules according to which one should construct $l(x), r(x)$ and $o(x)$ yet, but we just outlined what is the goal of such construction. If we can accomplish this, then, like it was demonstrated, we can turn the proving of a correct computation for a given input/output to proving that a polynomial has some specific roots. This is the ultimate goal of a QAP, because we can use all the techniques presented in the previous section and achieve a protocol for proving VC. As mentioned, all the operations should be multiplications, but this is not a problem, since, as we will see, the way we construct $l(x),\ r(x)$ and $o(x)$ assumes a slightly different elementary operation structure than the presented one, which uses multiplication and encodes addition in another way.

Construction of operand polynomials

For every variable v_k (for $k \in [n]$) present in the set of elementary operations we construct three polynomials $l_{v_k}(x),\ r_{v_k}(x)$ and $o_{v_k}(x)$ as follows:

$$l_{v_k}(j) = \begin{cases} 1 & \text{if } v_k \text{ present at operation } j \text{ as left operand} \\ 0 & \text{otherwise} \end{cases}$$

for $j = 1 \ldots d$, for d operations. The same applies for $r_{v_k}(x)$ and $o_{v_k}(x)$ if v_k is present as right operand or output. Observe the important following point: with such a construction, assigning a value c_k to a variable v_k, in all operations where v_k is present as a left operand, is equivalent to multiplying c_k with the polynomial: $c_k l_{v_k}(x)$. The same applies for assigning variables present as right operands or as outputs, using $r_{v_k}(x)$ and $o_{v_k}(x)$.

Using such constructed polynomials it is possible to add any number of present variables in a single operand for an operation. We simply add together all the variable polynomials (with assigned value), getting the final operand polynomial:

$$L(x) = l_0(x) + \sum_{k=1}^{n} c_k \cdot l_{v_k}(x)$$

Similarly for $R(x)$ and $O(x)$. This enables to represent an elementary operation as follows:

$$(v_1 + v_2 + ... + v_n) \cdot (v_1 + v_2 + ... + v_n) = (v_1 + v_2 + ... + v_n)$$

and consequently to encode addition. Any constant will be encoded using variables polynomials $l_0(x), r_0(x)$ and $o_0(x)$, which won't have values assignments as they "hardcode" constants in every operation. With values $\{c_1, \dots, c_n\}$ assigned to the variables we have:

$$\left(l_0(x) + \sum_{k=1}^{n} c_k l_{v_k}(x)\right) \cdot \left(r_0(x) + \sum_{k=1}^{n} c_k r_{v_k}(x)\right) = \left(o_0(x) + \sum_{k=1}^{n} c_k o_{v_k}(x)\right)$$

Finally, this is equivalent to:

$$L(x) \cdot R(x) = O(x)$$

and if all the variables were assigned in such a way that all the elementary operations encoding the computation hold, again, meaning that the computation was carried out correctly, then there exists a polynomial $h(x)$ such that:

$$H(x) = \frac{L(x) \cdot R(x) - O(x)}{t(x)}$$

for $t(x) = \prod_{j=1}^{d}(x - j)$, or, in other words, $p(x)$ has all the roots corresponding to the factors of $t(x)$.

3.2.2 Arithemtic Circuits

We introduced the notion of QAP trying to illustrate the intuition behind the construction of the polynomials composing it. More formally, as defined in [1], a QAP over a field $\mathbb{F}$ is a set of three polynomials $V = \{v_k(x)\}$, $W = \{w_k(x)\}$ and $Y = \{y_k(x)\}$, for $\{k = 0 \dots n\}$ and a target polynomial $t(x)$. Note that $v_k(x)$, $w_k(x)$ and $y_k(x)$ are just the $l_{v_k}(x)$, $r_{v_k}(x)$ and $o_{v_k}(x)$ presented above. Their construction is explained using the notion of *arithmetic circuit*. Following their explanation an arithmetic circuit is a graph whose nodes represent either an addition or a multiplication gate (modulo p) with two inputs and one output and its edges represent input/output values for all the gates. There are input edges that represent the input values of the circuit and an output edge that represent the output value of the circuit.

Encoding the circuit into polynomials $v_k(x)$, $w_k(x)$ and $y_k(x)$ is illustrated as follows: For each multiplication gate g in the circuit a root $r_g \in \mathbb{F}$ is picked and the target polynomial is then defined as $t(x) = \prod_g (x - r_g)$. An index $k \in [n] = \{1 \ldots n\}$ is associated with every input of the circuit and every output of a multiplication gate. These are the variables of the elementary operations set of before. As stated "*addition gates will be compressed into their contribution to the multiplication gates*", like demonstrated before. Polynomials in V, W and Y are then used to encode the left and right input and the output of a gate g. More specifically, if wire k is a left input to a gate g, then $v_k(r_g) = 1$, otherwise $v_k(r_g) = 0$, and similarly for the polynomials in W and Y for the right input and the output. With some intuition it can be recognized that this construction achieves the properties presented in the previous section and indeed the polynomial $p(x)$ is constructed as:

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

for assigned values $(c_1, \ldots, c_n)$.

3.3 The protocol

We have now a way to construct a polynomial $p(x)$ in such a way that the target polynomial $t(x)$ divides it if and only if the assigned values $\{c_1, \ldots, c_n\}$ are a correct assignment for the encoded computation. Therefore, by using the elements presented in the first section of this chapter, a protocol for VC can be created. In this section we present a simplified version of the Pinocchio protocol.

3.3.1 Setup phase

For a function F (the computation to be proven) a QAP is constructed, resulting in n variables and d operations. Using the polynomials from the QAP, a trusted party randomly samples s, α, and construct the public evaluation key and the public verification key:

- EK: $\left(\{g^{v_k(s)}, g^{w_k(s)}, g^{y_k(s)}, g^{\alpha v_k(s)}, g^{\alpha w_k(s)}, g^{\alpha y_k(s)}\}_{k \in [n]}, \{g^{s^j}\}_{j \in [d]}\right)$
- VK: $\left(g^{\alpha}, g^{t(s)}\right)$

Note that all the variable polynomials are present also in their version multiplied with the α element.

3.3.2 Proof

The prover uses its input/output variable assignments to compute the corresponding *witness*. A witness is an assignment to all the variables of a QAP; basically the variables obtained from transforming the computation in a set of elementary operations, or an arithmetic circuit. A valid witness, derived from a correct assignment of input/output variables, will generate a valid proof that pass the verification checks. More specifically, the prover, in possess of the witness $\{c_1, \ldots, c_n\}$ proceeds as follows:

- constructs the operand polynomial $V(x) = v_0(x) + \sum_{k=1}^{n} c_k v_k(x)$, and similarly $W(x)$ and $Y(x)$. He then derives

$$H(x) = \frac{V(x) \cdot W(x) - Y(x)}{t(x)}$$

- computes $g^{H(s)}$ using $\{g^{s^j}\}_{j \in [d]}$
- assigns the witness to the encrypted evaluation of variable polynomials

$$g^{V(s)} = \prod_{k=1}^{n} \left(g^{v_k(s)}\right)^{c_k}$$

  and similarly $g^{W(s)}, g^{Y(s)}, g^{\alpha V(s)}, g^{\alpha W(s)}, g^{\alpha Y(s)}$

The proof is then: $\left(g^{V(s)}, g^{W(s)}, g^{Y(s)}, g^{\alpha V(s)}, g^{\alpha W(s)}, g^{\alpha Y(s)}, g^{H(s)}\right)$

3.3.3 Verification

The proof is parsed as $\left(g^V, g^W, g^Y, g^{V'}, g^{W'}, g^{Y'}, g^H\right)$. Then two checks are performed, one to assess the correctness of the witness and one to restrict the prover in using the encrypted evaluated polynomials provided in the evaluation key:

$$e(g^V, g^W) = e(g^{t(s)}, g^H) \cdot e(g^Y, g)$$

$$e(g^V, g^\alpha) = e(g^{V'}, g)$$

, and similarly for g^W and g^Y.

3.3.4 Zero-knowledge

The protocol that was presented until now works for verifiable computation, but it is not completely zero-knowledge yet. It is true that the verifier doesn't learn the witness assigned by a prover in the normal procedure of the protocol, but still, since the received values are the direct encryption of the witness assigned to the variable polynomials, knowledge can be extracted from them. A verifier could for example brute-force different witnesses until he obtains the same values as the received ones and reconstruct the assigned witness. For true zero-knowledge randomness is needed. In fact, to make the scheme zero-knowledge, a modification that involves the prover to sample a random parameter δ and modify its proof with it is required. Some other elements must be included in the evaluation key for this modification to work but we won't present the details here as they are quite tedious and not purposeful to the overall understanding.

Chapter 4

ZoKrates

There exists various practical implementations of different zk-SNARK proving schemes, the most famous one probably being `libsanrk`, written in C++, among a few others ones. This library provides implementation of a few proving schemes, including Pinocchio, starting from a *Rank-1 Constraint System* (R1CS). A R1CS is another mean of representing a computation as a set of elementary operations, or *constraints*, as they have to be satisfied in order for the input witness to be valid for the encoded computation. They are very similar to arithmetic circuits and in fact they can be very easily reduced to them. Consequently, in order to be able to construct a proof for an arbitrary computation of choice, one must be able to work out the transformation and come up with a R1CS that correctly encodes it. In chapter 3 we presented how zk-SNARK protocols work starting from an arithmetic circuit without dwelling on how this representation is achieved, which is not in the objectives of this thesis. Even though `libsnark` has libraries that support you in this task by letting you building R1CS in a modular fashion out of classes that implement elementary computations, it is still quite an involved process, especially considered the level of understanding needed to execute such step.

ZoKrates is a library that allows to specify a computation pretty much like in an ordinary programming language and takes care of transforming it into a correct R1CS. It then gives the possibility to choose between different backend libraries (like `libsnark`) to transform the obtained R1CS into a QAP and to run a zk-SNARK protocol supported by the backend library. Its main purpose is to enable the deployment of zk-SNARK proofs on the Ethereum blockchain, contributing to the development of more complex decentralized applications with less cost (in terms of fees) and more privacy.

ZoKrates was chosen as the tool to experiment with the Pinocchio protocol as it offers the more direct access to it, taking care of writing circuits for us.

4.1 Writing circuits with ZoKrates

In the following we will illustrate how we write a ZoKrates program, which is the starting point from which a R1CS is constructed. As already mentioned it is really similar to an ordinary programming language and the idea is that the program itself is the computation for which you want a proof of correct execution to be constructed. For this reason it is not that complicated and there is not much to explain, except for an important detail regarding input variables.

A ZoKrates program

A simple ZoKrates program is listed in 4.1. Input variables, declared as the main function arguments, can be specified to be private. Private variables are assigned by the prover during the proof generation stage and the verifier learns nothing about them besides the output of the computation performed. Public variables, declared without the specification "`private`", instead don't need to remain secret and the verifier learns what the prover assigned to them. Output variables are declared after the right arrow.

Listing 4.1: Simple ZoKrates program

```
def main(private field x, field y) -> field:

        field z = 3*x + 7*y - 2

        return z
```

Generating the inputs

Often the most involved part of using ZoKrates is not so much writing the program itself, but more computing a correct input assignment for it. Of course this is not the case for the listed example program as the computation is trivial and a proof of correct computation makes not much sense on it. In practice though, zk-SNARK schemes are used for more complex computations, like a blockchain state transition function, or, in the case of outsourcing computation to a cloud server, a quite complex computation that can not be easily carried out on a mobile device.

4.2 Running the protocol

After ZoKrates compiles the program and construct a R1CS out of it, a chosen protocol supported by the chosen backend library can be run. Running a protocol is intuitive as it follows the steps outlined in chapter 3. First we should run the setup phase: The QAP is generated and the proving (evaluation) and verification keys are constructed. By specifying the values we want to assign to the input variables ZoKrates let us (impersonating a prover) compute the witness. Using the witness and the proving key we can then run the proof generation stage where the proof file is generated. Finally the verification step can be preformed using the proof file generated by a prover and the verification key.

Chapter 5

Proving ownership of Bitcoin

Finally, to put a zk-SNARK scheme into action we tackle the originally posed problem. We emphasize right away that ours is more of a mental exercise to show an example of an interesting problem that can potentially be addressed using these types of protocols and by no means intend to deliver a practically implementable solution. In fact we will simplify parts of the problem and use procedures that have the same purpose, but are actually implemented differently in the real world.

We want to prove ownership of some Bitcoin without revealing the public key that can spend them. In the Bitcoin setting, unspent Bitcoin are stored as *Unspent Transaction Output*s (UTXO), which contain the hash of the public key corresponding to the private key that can unlock the UTXO and spend the stored Bitcoin. Furthermore, in a second phase, we will also extend the problem to not disclosing which is the UTXO that is controlled, and really just prove the statement *"I can spend at least x Bitcoin"*, without revealing more information. An interesting scenario where this could be useful is, for instance, when more parties are trying to to join together in a trustless setting, in order to create a CoinJoin transaction and everybody has to prove ownership of some Bitcoin, but want to keep privacy on the address and on the amount.

Let's say a user wants to prove that he owns at least 1 bitcoin. To simplify the situation let's consider the simple case where his amount of Bitcoin is stored in a single UTXO. Then to convince someone of this he must prove three things:

1. He controls the private key that can spend that UTXO.
2. The UTXO is not already spent and it is part of the current UTXO set, which contains all the currently spendable Bitcoin.
3. The UTXO contains at least 1 bitcoin.

We are particularly interested in the first point as it represents a challenge that can be well addressed by mean of a zero-knowledge proof since we don't want to reveal the public key. The bitcoin protocol hashes the public key to facilitate manual transcription and as a prevention in case in the future it should be possible to reconstruct the private key from public key data due to an unanticipated problem, but it can also be beneficial for privacy reasons.

5.1 Structure of the solution

We use ZoKrates to devise a proof of the mentioned three points in a simplified setting, leveraging the zero-knowledge property of a zk-SNARK scheme like Pinocchio. As explained, we must come up with a ZoKrates program from which a R1CS and then QAP can be built. To address the first point we design a program that takes as inputs a public key, a signature on a message and the message itself, as well as a hash, which should be the hash of the public key. It then verifies the signature validity with the given public key and that hashing the latter yields the input hash. The public key will be privately assigned from the prover and the verification stage will use the generated proof to check that indeed the assigned values pass the program assertions, proving the knowledge of the private key by the prover. This can be concluded since the program

- verifies that hashing the input public key corresponds to the input public key hash
- verifies the signed message with the input public key, which links the public key with the private key

To address the second and the third point we allow ourselves to propose a quite simplistic strategy. Assuming access to the current valid UTXO set, we imagine to make use of a Merkle Tree to represent it. A Merkle Tree is a tree whose leaves are hashes of some data set and the value of every parent node up until the root is the hash of the concatenation of its two children values. It is used to represent a data set in a very efficient way and to ensure its integrity since the modification of something in the data set will result in a different root hash. Proving that some data is included in the tree is quite efficient: following the path from the leaf corresponding to that data up to the root we provide the value of every sibling node needed to compute the value of the parent node. If the final computed value corresponds to the root hash the data is in the tree. Specifically, we build a Merkle Tree with public key hashes belonging to single UTXOs whose stored amount of Bitcoin is at least 1. We assume that such a Merkle tree over UTXOs of at least 1 bitcoin is available to both prover and verifier through, for instance, a standard shared algorithm that can construct it after every new block added to the blockchain. We expand our program design by adding an inclusion proof for the input hash in the proposed Merkle Tree. To address the mentioned extension to the problem and keep the controlled UTXO secret, we now hide the input public key hash and also the input arguments for the Merkle Tree inclusion proof, such that the leaf corresponding to our UTXO also remains secret.

We present the details of this design in the next sections.

5.2 Proving knowledge of private key

We first display how we address the proof of knowledge of a private key given the hash of the corresponding public key. We combine proving knowledge of an hash preimage, which is in fact a typical problem that poses itself very well to the zero-knowledge setting and it is often used as a demonstrative example, with the verification of a signature on a message. Note that in the actual bitcoin implementation ECDSA with the secp256k1 curve parameters is used as the signing algorithm and Hash160, which involves SHA256 hashing followed by a RIPEMD-160 hash, is used as an hash algorithm. In the ZoKrates standard library are available implementations of SHA256 and EdDSA, a signature algorithm based on the Edwards curve, whose purposes are the same and which we will use instead.

The program

The program is proposed in listing 5.1. As it can be seen it returns correctly only if the assertion at line 11 is true, guaranteeing that the signature on the message was actually performed with the correct private key and the value of `HashOfA` really corresponds to the SHA256 hash of the public key. For better understanding: private inputs `R` and `S` composes the signature, private input `A` is the public key and `HashOfA` is the hashed public key (of the claimed UTXO) and finally `M0` and `M1` compose the message hash and are used to verify the signature.

Listing 5.1: ZoKrates program that prove knowledge of a private key given an hashed public key

```
1  def main(field[2] R, field S, private field[2] A, u32[8]
       HashOfA, u32[8] M0, u32[8] M1):
2
3          BabyJubJubParams context = context()
4          bool valid_signature = verifyEddsa(R, S, A, M0, M1,
               context)
5
6          u32[8] pk_x = field_to_u32_8(A[0])
7          u32[8] pk_y = field_to_u32_8(A[1])
8          u32[8] computed_hash = sha256(pk_x,pk_y)
9          bool hashes_match = HashOfA == computed_hash
10
```

```
        assert(valid_signature && hashes_match)
        return
```

Running the setup phase of the Pinocchio protocol for this program generates a QAP with 138313 variables and degree 139264. Being dependent in the number of variables the public evaluation key has a size of around 30 MB whereas the verification key, being dependent only in the input and output variables of the program, has a size of only around 1 KB. To generate a correct witness for the proposed program a prover, owning a public/privare key pair associated with a UTXO, passes as inputs arguments its public key, a message, a signature on that message preformed with its private key and the hashed public key contained in the UTXO.

Inputs generation

To generate a correct input assignment for this program we use python. Specifically, we make use of the ZoKrates PyCrypto library, which implements classes for working with Edwards signatures (EdDSA).

Listing 5.2: Input generation for proving private key ownership

```
sk = PrivateKey.from_rand()
pk = PublicKey.from_private(sk)

raw_msg = "Don't trust. Verify."
msg = sha512(raw_msg.encode("utf-8")).digest()
sig = sk.sign(msg)

assert(pk.verify(sig, msg))
write_signature_for_zokrates_cli(sig, pk, hash_pk(pk), msg, '
    zokrates_inputs.txt')
```

The proof

Generating the proof with ZoKrates yields a file called `proof.json` which contains the 8 group element that compose the proof formatted with the json standard. In additon to that it also contains the values assigned to the public inputs. For this program, the generated proof size is only 2294 bits, less than 300 bytes!

5.3 Inclusion in a Merkle Tree

As explained, to adapt our solution for the extended problem, we combine the previously introduced program with an inclusion proof of the input public key hash in the proposed imaginary Merkle tree. This extension was also investigated as a "symbolic" solution with no real attention to the practicality of it and it was implemented for trees with hardcoded depth three, as making the depth of the tree variable represented a non-trivial challenge to solve with ZoKrates. An example ZoKrates program to perform a Merkle tree inclusion proof was already at our disposal from the standard library, but the real challenge in this task was writing a python program to generate an actual correct witness. We list the ZoKrates program in listing A.1. In addition to the inputs already present in the previous program we have now a few additional arguments to construct this a proof. They consist in: the root hash of the Merkle tree and the value of the siblings nodes needed to compute it, which are declared as private inputs, such that it is not possible for a verifier to identify which leaf is being proven as part of the tree. To determine in which order to join the so far computed digest with the next sibling a "direction selector" is needed, and also passed as a private input.

With python an implementation of a Merkle tree was constructed. We then initialize a Merkle tree using arbitrary SHA256 hashes as leaves, picturing them as the public key hashes extracted from the UTXO set as explained before. In one of the leaf we insert the hash of the public key to which the owned private key corresponds. We then extract all the needed values for the proof and pass them to Zokrates.

Chapter 6

Conclusions

In this report we presented a study of the concepts behind zk-SNARK schemes. We saw how polynomials, homomorphic and pairing based cryptography, the knowledge of exponent assumption and quadratic arithmetic programs are exploited to achieve verifiable computation, potentially with zero-knowledge. An example problem solvable through such a scheme was presented in a simplified setting and addressed using ZoKrates. Many more exciting problems can be solved using a zk-SNARK protocols and many new applications making use of them are emerging.

Given the magnitude of the various topics related with such protocols, a formal and detailed study wasn't in the range of a semester thesis and a lot of abstraction had to be done by treating some concept as blackboxes and only presenting why they are needed and what are the guarantees given. Nevertheless, we hope that an high-level understanding of it could be grasped and a smooth accompaniment in this complex field was deliverd.

Bibliography

[1] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 238–252.

[2] J. Eberhardt and S. Tai, “Zokrates - scalable privacy-preserving off-chain computations,” in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1084–1091.

[3] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: https://doi.org/10.1137/0218012

[4] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again,” Cryptology ePrint Archive, Report 2011/443, 2011, https://eprint.iacr.org/2011/443.

[5] I. Damgård, “Towards practical public key systems secure against chosen ciphertext attacks,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 445–456.

Appendix A
Merkle inclusion extension

Listing A.1: ZoKrates program that prove knowledge of a private key given an hashed public key

```
def main(field[2] R, field S, private field[2] PublicKey, private u32[8] HashOfPubKey, u32[8] M0, u32[8] M1, u32[8] rootDigest, private bool[3] directionSelector, private u32[8] PathDigest0, private u32[8] PathDigest1, private u32[8] PathDigest2):

        // Check that the provided public key was actually used to sign message [M0,M1] to produce signature R,S
        // Check that the provided HashOfPubKey corresponds to the sha256 hash of PublicKey

    BabyJubJubParams context = context()
    bool isSigValid = verifyEddsa(R, S, PublicKey, M0, M1, context)
    bool hashAreEqual = (sha256padded(unpack256u(PublicKey[0]), unpack256u(PublicKey[1])) == HashOfPubKey)

        // Check if HashOfPubKey is actually part of the Merkle Tree:
        // compute the root starting from HashOfPubKey and using the provided PathDigests

        u32[8] currentDigest = HashOfPubKey

        u32[16] preimage = multiplex(directionSelector[0], currentDigest, PathDigest0)
        currentDigest = sha256padded(preimage[0..8],preimage[8..16])

        preimage = multiplex(directionSelector[1], currentDigest
            , PathDigest1)
        currentDigest = sha256padded(preimage[0..8],preimage
            [8..16])

        preimage = multiplex(directionSelector[2], currentDigest
            , PathDigest2)
        currentDigest = sha256padded(preimage[0..8],preimage
            [8..16])


        // All tree assertions must pass for the program to
           return
        assert(isSigValid)
        assert(hashAreEqual)
        assert(rootDigest == currentDigest)

        return
```